Università degli Studi di Torino

Dipartimento di Informatica

PhD Thesis

AGGREGATE PROGRAMMING FOR THE INTERNET OF THINGS

Advisor

Prof. Dr. Ferruccio Damiani

Dr. Giorgio Audrito

Dr. Gianluca Torta

Ing. Maurizio Griva

Ext. Reviewers

Prof. Dr. Giorgio Delzanno

Prof. Dr. Violet Ka I Pun

Candidate

Lorenzo Testa

May 2024

a.y. 2023 / 2024

I declare that, as author of the document, I'm responsible for its content, and for parts taken from other works credit is explicity given to others by citation or acknowledgement.

Aggregate Programming for the Internet of Things

Lorenzo Testa

May 2024

Contents

Chapter 1

Introduction

1.1 Background and Motivations

This thesis presents the work done within the industry as an employee of Santer Reply for my Industrial PhD program with the University of Turin. In particular Concept Reply, the division of Reply of which I'm part of, is a company focussed on developing end-to-end IoT solutions, from embedded software to frontend/backend applications. Concept Reply has many years of expertise in developing edge to cloud solutions for industrial clients.

In the last years the number, pervasiveness and variety of interconnected devices has been constantly increasing. This network, commonly referred as the Internet of Things (IoT), is composed of numerous and different devices, like industrial machines, vehicular control systems, personal smart devices, drones and all types of sensors. In particular the Industrial IoT (IIoT) applies the IoT concepts to the industry to attempt to increase productivity, improve safety and reduce cost in the new generation manufacturing plants. Developing applications for the IoT requires to handle a network of interconnected devices with many challeges, e.g.: devices with low computational power, devices not able to communicate with the full network, devices not always available and with changing topology, high amount of data produced in real time.

Aggregate Programming [19] (AP) is a paradigm started in 2015 that aims to simplify the development of distributed applications for resource-constrained IoT devices. In order to reduce the complexity of development, the Aggregate Programming paradigm focuses on treating a dynamic network of devices as a cooperative collection instead of focusing on the behavior of single devices. It abstracts from the detail of communication and behavior for the individual heterogeneous nodes of the network but focuses instead on a simple higher level programming API that guarantees resilience to network changes. All the devices of the network cooperate

in the execution of the same program in asyncronous rounds to achieve a common goal. Its applications have been studied in particular for crowd control applications and smart buildings, e.g. efficient crowd dispersal without overcrowding safety paths in case of dangers. The AP paradigm has been implemented in JVM languages and C++ and has been applied to numerous simulated case studies.

1.2 Research question

Although the Aggregate Programming paradigm has been applied to numerous case studies it has never been deployed to real world IoT devices. The goal of this thesis and research question is whether the Aggregate Programming, and in particular its implementation FCPP [8, 17], can be applied to real world IoT devices and used to develop an IIoT case study.

This deployment aims at validating the characteristics of the Aggregate Programming, in particular its suitability for applications on heavily resource constrained devices and primitive communication capabilities, so that in the future it can be applied to bigger devices, e.g. drones.

1.3 Contributions

We developed the support to run FCPP on the DWM1001 general-purpose system on a chip: an IoT module with Ultra Wideband (UWB) and Bluetooth Low Energy (BLE) communication capabilities, a device which was already in use in large scale industrial applications by Reply. This is the first deployment of an Aggregate Programming application on real IoT devices. We based our porting on the open source Contiki-NG [31] operative system after developing the support for it by starting from the existing project Contiki-UWB [23]. For the message exchange we utilized the advertising feature of the BLE transceiver of the module, while to compute the distance between devices (ranging) and use it in our applications we utilized the UWB transceiver obtaining a precision up to 10 centimeters. In particular for the ranging we developed a basic custom protocol to prevent transmissions collisions and to keep the information up-to-date before the rounds of execution of the aggregate programs.

We developed a case study of smart warehousing using the Aggregate Programming paradigm. In this case study the warehouse operators perform loading and unloading of pallets of goods on the paths guided by an aggregate program. The program also provides warnings to prevent collisions between forklifts and collects logs of all the operations performed in the warehouse. For this case study we run simulations with hundreds of devices and a smaller scale real experiment

in laboratory using the DWM1001 modules. This case study has been published in [34].

We are also working on using the Aggregate Programming paradigm on new case studies based on robots and drones, like the Clearpath Jackal.

1.4 Outline

This thesis is organized as follows. Chapter 2 provides an introduction to the concepts of the IIoT and Edge computing. Chapter 3 illustrates the Aggregate Programming paradigm and its formalization: the Field Calculus. Chapter 4 provides the syntax and internal structure of FCPP: a C++ library that implements the Field Calculus. Chapter 5 presents the research project and its goals. Chapter 6 provides a brief introduction on all the existing technologies utilized by the project. Chapter 7 shows the development work done in order to support FCPP on the DWM1001 module. Chapter 8 presents the industrial case study implemented to validate the usage of the Aggregate Programming and the results obtain, both through simulations and experiments. Chapter 9 presents other case studies that have been proposed in other publications but are not yet implemented. Chapter 10 presents the work in progress to develop new case studies on drones and autonomous robots. Finally chapter 11 draws conclusions and delines the future developments.

Chapter 2

Edge Computing and IIoT

2.1 IIoT

The increasing number of interconnected devices in the human environment leads naturally to the challenge of how to program and coordinate them so that they can easility communicate and collaborate. Phones, vehicles and industrial machines are very often equiped with sensors and communication capabilities. Those networks of devices are usually referred to as the Internet of Things (IoT). The IoT paradigm was firstly introduced by Kevin Ashton in 1998 as a concept for connecting things or objects to the Internet [28].

The application of concepts and technologies of IoT to the (smart) industry is commonly called Industrial Internet of Things (IIoT), which can improve the efficience and safety in the workplace thanks to the continous processing of data collected from sensors and machines. In [28] IIoT is more formally defined as: "IIoT is the network of intelligent and highly connected industrial components that are deployed to achieve high production rate with reduced operational costs through real-time monitoring, efficient management and controlling of industrial processes, assets and operational time".

Slighty different definitions of IIoT (e.g. [28], [30]) share most aspects with the one depicted in Figure 2.1. Devices on the edge collect data using *sensors* and perform actions using *actuators*. The Edge devices are connected to the internet, often through Edge Gateways. The data are collected in the Cloud, where computations on the data are performed to provide analytics and send actions to be performed back to the Edge.

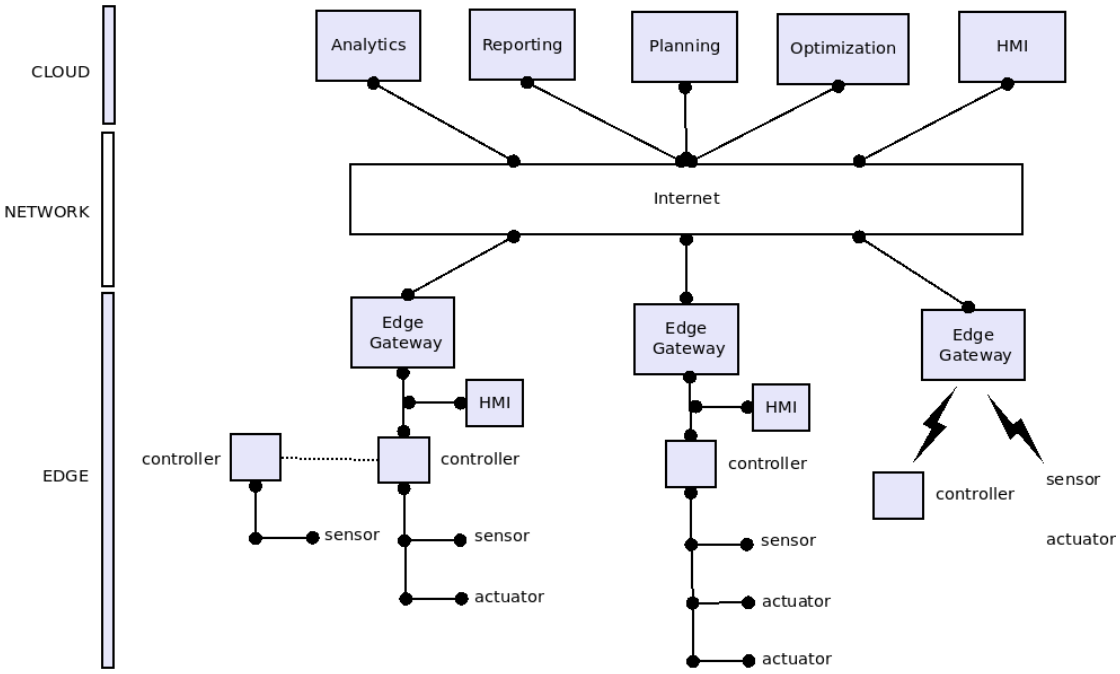

Figure 2.1: Reference Architecture of the IIoT. From [34].

2.2 Edge computing

Performing all the high-level computations on the Cloud presents a series of challenges. The communication Edge-Cloud introduces a network latency, the connection needs to be reliable and scalable and the Cloud architecture has an additional cost of setup and maintainence. The field of *Edge computing* aims at addressing the problem by moving some computations closer to the Edge, notably moving (part of) the data processing closer to where the data is produced (see Figure 2.2). We partition the Edge layer into two sublayers: the *Far Edge* layer, which consists of the devices connected directly with the machines, sensors and actuations; and the *Edge Gateways*, which consists of the devices communicating with the outside world. The Edge computing paradigm doesn't specify exactly on which devices the computation should happen or how powerful the devices should be. Therefore the term can apply to computation running close to the Edge Gateways, on the Edge Gateways themselves or on the Far Edge layer, constrained by the memory and computational capabilities of the smart devices.

In this thesis we will consider in particular the Aggregate Programming paradigm, which is an approach to Edge computing focussed on running computations on the Far Edge using resource-constrained devices, to build IIoT applications.

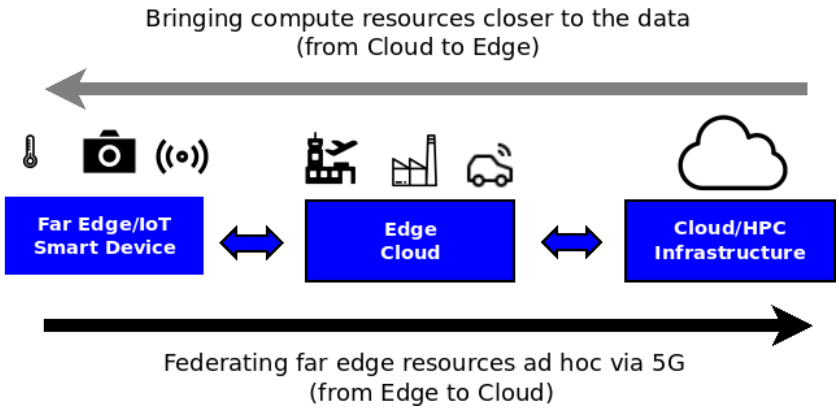

Figure 2.2: Cloud-Edge-IoT Orchestration. From [34].

Chapter 3

Aggregate Programming

The Aggregate Programming (AP) [19, 36] is a paradigm of distributed computation abstracting away from the individual devices to focus on global behavior of the network. AP is based on the notion of *computational field* [30] (or simply called *field*): a global, distributed map from computational devices to computational objects [18] that evolves over time. This notion is formalized by the Field Calculus (FC) [18]: a small formal functional language providing low level constructs to define and manipulate fields, that works as the foundation for the AP paradigm. The AP paradigm aims at solving many of the challenges of developing applications for the Internet of Things. All communication details are hidden and message exchange is implicit in the computation itself, hiding all the coordination complexity from the developer. AP's computation model is naturally resilient to changes to the network topology and resilient to device failures, with provable formal properties like self-stabilization [35], i.e. the ability of a system to recover from arbitrary changes in state.

3.1 Computational Model

The computational model of the field calculus [18] is based on a network of devices that executes a common program `P` in asyncronous rounds. Each device comunicates with neighbour devices following a dynamic (physical or logical) proximity relation, that changes due to mobility, failures and delays. These devices can join or leave the network at any point of the computation. Devices are usually equipped with *sensors*, which provide *inputs* for the program, and *actuators*, which act based on the *output* of the program.

The computation can be considered from two different points of view: local and global. From the local point of view of a single device every round of execution is composed by the following steps:

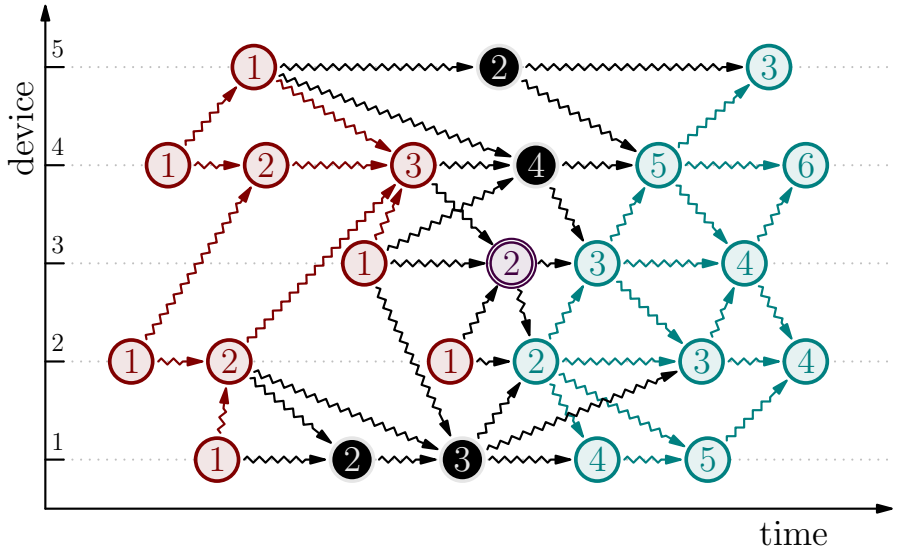

Figure 3.1: Example of a space-time event structure from [9], comprising events (circles), neighbour relations (arrows) and devices (ordinate axis). With respect to the doubly-circled event, the red events are its causal past, the cyan its causal future and the black ones are concurrent.

- all the information from sensors and the device memory are collected;
- from the most recent messages from neighbouring devices a *neighbouring field* is formed;
- the program is executed with the collected information;
- the results of the computation are stored in the device memory and shared to the neighbouring devices as a message;

A device δ is said to "fire" when it runs a round of execution. From the global (or aggregate) point of view the whole computation can be seen as a space-time data structure, called *field evolution* Φ. Every execution is represented by a point in space-time called an *event* ϵ, Φ is then a map from events to computations values. As described in [10] the causal relationship between events can be formalized by an *event structure*.

An *event structure* $\mathbf{E}$ is a countable set of events E togheter with a neighbouring relation $\rightsquigarrow \subseteq E \times E$ and a causality relation $< \subseteq E \times E$, such that the transitive closure of $\rightsquigarrow$ forms the irreflexive partial order $<$ and the set $\{\epsilon' \in E | \epsilon' < \epsilon\}$ is finite for all ϵ (i.e., $<$ is locally finite). Every $\rightsquigarrow$ relation represent a message sent from the head neighbour to the tail neighbour with the results of the head

$$
\begin{array}{lcll}
\texttt{P} & ::= & \overline{\texttt{F}}\ \texttt{e} & \text{program} \\
\texttt{F} & ::= & \texttt{def d}(\overline{\texttt{x}})\ \{\texttt{e}\} & \text{function declaration} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{v} \mid (\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}}\ \texttt{e} \mid \texttt{e}(\overline{\texttt{e}}) \mid \texttt{if(e)\{e\}else\{e\}} \mid & \text{expression} \\
 & & \texttt{nbr\{e\}} \mid \texttt{rep(e)\{(x)=>e\}} \mid & \\
 & & \texttt{spawn(e, e, e)} & \\
\texttt{v} & ::= & \ell \mid \phi & \text{value} \\
\ell & ::= & \texttt{c}(\overline{\ell}) \mid \texttt{f} & \text{local value} \\
\phi & ::= & \overline{\delta} \mapsto \overline{\ell} & \text{neighbouring field value} \\
\texttt{f} & ::= & \texttt{b} \mid \texttt{d} \mid (\overline{\texttt{x}}) \overset{\tau}{\texttt{=>}}\ \texttt{e} & \text{function value}
\end{array}
$$

Figure 3.2: Abstract syntax of the field calculus from [18, 22]

computation. Figure 3.1 shows an example of an event structure, showing the relations among events.

3.2 Field Calculus

The Field Calculus is a tiny functional language providing basic constructs to work with fields [18]. In particular the *Higher-order field calculus (HFC)* is an extension of the field calculus, which treats functions as first-class values, that will be considered by this chapter when discussing syntax and semantics. In this chapter the syntax and semantics of HFC are extended to include the `spawn` operator as described in [22] due to its usage in the case study proposed in chapter 8. HFC presents a language with only three basic operators: `branch` expressions which allows to split the network in separated branch of computation that don't exchange data; `nbr` expressions which allows to share a computed value and is the basic mean of communication; and `rep` expressions which allows to memorize a value between rounds of executions and use it in the new computation. The additional `spawn` expressions allow to create aggregate processes: a distributed transient activity in which partecipates a dynamic aggregation of devices. A processes can be started by any devices and is propagated to the neightbours, which in turn can decide to participate or ignore the process, creating a delimited "computation bubble".

Figure 3.2 presents the syntax of the field calculus. The overbar denotes a sequence, e.g. $\overline{\texttt{e}}$ denotes a (possible empty) sequence of expressions $\texttt{e}_1, \texttt{e}_2, \ldots, \texttt{e}_n$, and an empty sequence is denoted by $\bullet$.

A program is then a sequence of function definition followed by a main expres-

sion e, which defines the behavior of the aggregate.

A function declaration defines a function named d with a sequence of variable names $\overline{\texttt{x}}$ and a body of the function consisting in an expression e. The defined functions can be recursive.

An expression can be:

- a variable `x` referring to a function parameter
- a value `v`
- an anonymous function $(\overline{\texttt{x}})\ \overset{\tau}{\texttt{=>}}\ \texttt{e}$ where $\overline{\texttt{x}}$ are variable names for the formal parameter, `e` is the body of the function and τ is a `tag` identifying the function. τ doesn't appear in the source code but is uniquely determined by the function syntactical representation
- a function call $\texttt{e}(\overline{\texttt{e}})$, where `e` evaluates to a function `f`, $\overline{\texttt{e}}$ are the function arguments and evaluates to the function application
- a branching expression $\texttt{if}(\texttt{e}_0)\{\texttt{e}_1\}\{\texttt{e}_2\}$, also called *domain restriction expression*. It is a lazy evaluated expression that divides the computation in two branches: the devices for which $\texttt{e}_0$ evaluates to `True` compute $\texttt{e}_1$, the devices for which $\texttt{e}_0$ evaluates to `False` compute $\texttt{e}_2$
- an `nbr`-*expression*, also called *neightbouring field construction*, $\texttt{nbr}\{\texttt{e}\}$ which evaluates to a local map from neighbouring devices (including the execution device) to their most recent evaluation of the expression `e`
- a `rep`-*expression*, also called *time evolution expression*, $\texttt{rep}(\texttt{e}_0)\{(\texttt{x})\texttt{=>e}\}$, which at each round evaluates to the application of the result of the previous round to the function, using the *initialization expression* $\texttt{e}_0$ in the first round
- a `spawn`-*expression* $\texttt{spawn}(\texttt{e}_b, \texttt{e}_k, \texttt{e}_i)$, which models a collection of aggregate processes. Expression $\texttt{e}_b$ its a function, of informal type $k\texttt{->}a\texttt{->}\langle v, \texttt{bool}\rangle$, that models the processes behaviour. It takes an identifier (process key) k, an input argument a and returns an output value v and a boolean stating whether the process should be maintained alive or stopped. Processes are propagated in the network by all devices that evalutes to `True` and *terminate* when all devices evalutes to `False`. Expression $\texttt{e}_k$ is a field of process keysets to add at each device. If in a device the $\texttt{e}_k$ keyset contains a key that already belongs to the set of active processes, then it will not start a new process with the same key since it has the same identity. Expression e_i is the input field for the processes. The `spawn` expression evaluates to a map from process keys to values.

Each expression will evaluate to a field at the macro-level, mapping when a device executes a computation to values.

A value can be either a *neighbouring field* ϕ or a *local value* ℓ, e.g. a constant field mapping each event to the value 1. Neighbouring field values do not appear in the source code but can only be computed dynamically, usually by built-in operators like `nbr`.

Local values can be either *data value* $\mathtt{c}(\overline{\ell})$, in which `c` is a data constructor and $\overline{\ell}$ are local value arguments, or a function value `f`. Data values can be primitive values like `True` or `False` or structured values like pairs or lists.

A *function value* `f` can be a built-in function `b`, a declared function `d` or a closed anonymous function value $(\overline{\mathtt{x}})\ \overset{\tau}{\mathtt{=>}}\ \mathtt{e}$.

3.2.1 Field Calculus Semantics

The operational semantics of the field calculus is shown in figure 3.3. The derive judgements are in the form $\delta; \Theta; \sigma \vdash \mathtt{e} \Downarrow \theta$ which means that the expression `e` evaluates to the value-tree θ with respect to the value-tree environment Θ, the device δ and the sensor state σ. A *value-environment* Θ is map from device identifiers δ to value-trees, which will contain the value-trees produced by the most recent evaluation of the expression `e` by the neightbours of δ, including δ itself. A *value-tree* θ is either an ordered tree of values tracking the results of all the computed subexpressions or a map from keys to value-trees (this type of value-trees is used only by `spawn`-expressions). The evaluation rules are expressed recursively by evaluating the subexpressions with respect to a new value environment obtained by the subtrees (when present) of the current value-tree environment Θ, this process is called *alignment*. σ is a data structure containing information about the device sensors that will be used by the built-in functions.

The auxiliary function ρ extract the root value of a value-tree, while π extracts a subtree from an ordered value-tree or a value-tree from a key. The function F is a filtering function that selects the value-trees whose root is a pair $\mathtt{pair}(\mathtt{v}, \mathtt{True})$ and extracts `v` as the new root. The functions *name*, *args* and *body* extract respectively the name, formal parameters and body of a function.

Rules [E-LOC] and [E-FLD] define the evaluation of local values and neighbouring field values. Both produce a value-tree with no subtrees, but in case of neighbouring fields the domain of the field is restrict to the aligned devices.

Rule [E-B-APP] and [E-D-APP] model the application of built-in and user-defined (declared or anonymous) functions. In the first case the root of the value-tree is computed by a function $(\!|\mathtt{b}|\!)_{\delta}^{\pi^{\mathtt{b}}(\Theta),\sigma}$ different for each built-in function `b`. In the second case the root is computed by execution the body of the function `f`, the resulting value-tree also has one additional subtree containing the value-tree

resulting from the execution from the body. This is necessary for the alignment of the environment during the execution.

Rule [E-NBR] models the evaluation of `nbr`-expression, which extracts from the value-tree environment the neighbouring values to build a neighbouring field as the root result. In the resulting field the value associated with the executing device is updated by the new result of the execution of `e`. The notation $\phi[\delta \mapsto \mathtt{v}]$ denotes that if δ was part of the domain of ϕ then its corresponded value is updated to `v`.

Rule [E-REP] models the evaluation of `rep`-expression, which extract from the value-tree environment the root of the last computed tree in order to replace `x` in the new evaluation of $\mathtt{e}_2$.

Rule [E-IF] models a branching expression by computing and aligning on only one subtree according to the evaluation of `e`.

Rule [E-SPAWN] models the evaluation of a `spawn`-expression. The subexpressions e_1, e_2 and e_3 are evaluated to compute the trees θ^1, θ^2 and θ^3. Then using the keys from $\rho(\theta^2)$ and the keys from the neightbours a set of active *process keys* $\overline{\mathtt{k}}$ is computed. Using the keyset the function resulting from $\rho(\theta^1)$ is computed against each key and finally filtered by F discarding the terminated processes before being made available to the neightbours.

Figure 3.4 defines the operational semantics for the evaluation of whole networks, i.e. it models the distributed evolution of the computational fields over time. Ψ is map from device identifiers to value-tree environments, modelling the overhall status of the devices at a given time. τ models the topology of the network as a map from device identifiers to sets of identifiers of the neigbouring devices. Σ models the sensors state as map from device identifiers to the local sensor state of the device. *Env* is a pair of topology and sensor state modelling the system's environment. N models the whole network configuration as a pair of an environment *Env* and a status field Ψ.

The network semantics is given in terms of small-steps transitions of the kind $N \xrightarrow{act} N'$ where *act* can be either a device identifier δ representing its firing or the label *env* representing any environment change. The following notation is used. $F(\cdot)$ is a filtering operation that clears old stored values from the value-tree environements in Ψ, usually based on space/time tags attached to value-trees. $\overline{\delta} \mapsto \Theta$ denote the map sending each device identifier in $\overline{\delta}$ to the same value-tree environment Θ. $\Theta_0[\Theta_1]$ denote the value-tree environment with domain $\mathbf{dom}(\Theta_0) \cup \mathbf{dom}(\Theta_1)$ coinciding with Θ_1 in the domain of Θ_1 and with Θ_0 otherwise. $\Psi_0[\Psi_1]$ denotes the status field with the same domain as Ψ_0 made of $\delta \mapsto \Psi_0(\delta)[\Psi_1(\delta)]$ for all δ in the domain of Ψ_1, $\delta \mapsto \Psi_0(\delta)$ otherwise.

Rule [N-FIR] models a computational round at a device δ. From the filtered local value-tree environement $F(\Psi)(\delta)$ determines the single device semantics obtaining the value-tree θ and use it to update the value-tree environment of δ's neighbours.

Rule [N-ENV] models a change of the environment Env to a new well-formed environment Env'. The devices not appearing in the new environment are removed from the status field and new devices are mapped to the empty context.

3.3 Implementations

There are multiple implementations of the aggregate programming paradigm, mainly Protelis, ScaFi and FCPP.

Protelis [33] implements the field calculus as an external Domain Specific Language (DSL) that runs on the Java Virtual Machine (JVM) and provides full interoperability with the Java type-system and API. The textual Protelis programs are translated into a valid representation of the higher order field calculus semantics, then this representation is executed at regular intervals by the Protelis interpreter. Protelis abstracts over the device capabilities and communication system, allowing to use it for both simulations (like the Alchemist simulator [32]) and real world application as long as the JVM is supported by the hardware. Protelis also provides a rich standard library for the application developers. Since Protelis is an external DSL, software developers that desire to use it need to learn its language and integrate it with their JVM code (e.g. Java code).

ScaFi (Scala Fields) [21] its an internal DSL/library for the Scala programming language. Like Protelis runs on the JVM but uses directly the Scala language to define its API. This provide an advantage for developers that are already familiar with the language and takes advantage of the powerful type system and type inference of Scala. This implementation of the field calculus has some semantic differences compared with the HFC, which are formalized in [12] as a new calculus called Neigbours Calculus (NC).

FCPP (FieldCalc++) [8, 17] implements the field calculus as an internal DSL for C++ 14. Unlike Protelis and ScaFi, FCPP doesn't require a JVM allowing programs to run on more resource constrained hardware, like microcontrollers. A more extensive overview of FCPP is provided in Chapter 4.

3.4 Exchange Calculus

In [11] a new calculus called Exchange Calculus (XC) has been recently proposed, to which all FC programs can be translated. The XC is based on a single communication primitive $\texttt{exchange}(\texttt{e}_i, (\underline{n})\ \texttt{=>}\ \textit{return}\ \texttt{e}_r\ \textit{send}\ \texttt{e}_s)$. The exchange operator computes the initial value ℓ_i from $\texttt{e}_i$, then substituites the variable $\underline{n}$ with the field of values received from the neighbours, using ℓ_i as default. The expression $\texttt{e}_r$ determines the value returned by the operator, while the expression $\texttt{e}_s$ computes a

field that associate for each neightbour what value to send to that neightbour for their future computation of the exchange operator. Unlike the `nbr` operator of the field calcules, the `exchange` operator can send a different value to each neightbour, to allow custom interaction.

The exchange calculus is available as an extension of ScaFi and as an extension of FCPP integrated in the main FCPP distribution.

3.5 Comparison with other IIoT paradigms

A comparison of the Aggregate Programming paradigm, in particular using the Exchange Calculus, to other programming models has been recently implemented[1] in [11]. In the comparison the aggregate functions `distanceTo` (a distributed version of the Bellman-Ford algorithm to compute distances) and `channel` (a function which selects a region of a given width connecting a source device with a destination device) have been compared with two publisher-subscriber solutions. The resulting aggregate program was composed of less lines of code and less occurrencies of solution specific constructs. The publisher-subscriber solution also spread the logic over multiple subscription handles compare to the Exchange Calculus thanks to the implicit declaration of data exchange in the aggregate constructs.

More sophisticated IIoT platforms like Thingworx or Ubidots are more focused on the data collection from machines when compared with the Aggregate Programming paradigm and usually run a centralized computation on the data. In contrast AP, thanks to distributing the computation directly on the nodes, allows to develop programs that exploit the network structure itself, e.g. for self-organizing capabilities.

[1] The comparison is publicly available at: `https://github.com/metaphori/aggregate-paradigm-comparison`

Value-trees and value-tree environments:

$\theta \quad ::= \quad \mathtt{v}\langle\overline{\theta}\rangle \mid \overline{\mathtt{v}} \mapsto \overline{\theta}$ value-tree

$\Theta \quad ::= \quad \overline{\delta} \mapsto \overline{\theta}$ value-tree environment

Auxiliary functions:

$\rho(\mathtt{v}\langle\overline{\theta}\rangle) = \mathtt{v}$

$\pi_i(\mathtt{v}\langle\theta_1, \ldots, \theta_n\rangle) = \theta_i \quad \text{if } 1 \leq i \leq n$

$\pi^{\mathtt{f}}(\mathtt{v}\langle\theta_1, \ldots, \theta_{n+1}\rangle) = \theta_{n+1} \quad \text{if } \mathtt{f} \text{ is a built-in function and } \rho(\theta_{n+1}) = \mathtt{f}$

$\pi^{\mathtt{f}}(\mathtt{v}\langle\theta_1, \ldots, \theta_{n+2}\rangle) = \theta_{n+2} \quad \text{if } \mathtt{f} \text{ is a non-built-in function and } \mathit{name}(\rho(\theta_{n+1})) = \mathit{name}(\mathtt{f})$

$\pi^{\mathtt{f}}(\theta) = \bullet \quad \text{otherwise}$

$\pi^{\mathtt{k}}(\overline{\mathtt{v}} \mapsto \overline{\theta}) = \theta_i \quad \text{s.t. } \mathtt{v}_i = \mathtt{k} \text{ if it exists, else } \bullet$

$F(\theta) = \mathtt{v}\langle\overline{\theta}\rangle \quad \text{if } \theta = \mathtt{pair}(\mathtt{v}, \mathtt{True})\langle\overline{\theta}\rangle \text{ else } \bullet$

$$\text{For } \mathit{aux} \in \rho, \pi_i, \pi^{\mathtt{f}}, \pi^{\mathtt{k}}, F: \quad \begin{cases} \mathit{aux}(\delta \mapsto \theta, \Theta) = \delta \mapsto \mathit{aux}(\theta), \mathit{aux}(\Theta) & \text{if } \mathit{aux}(\theta) \neq \bullet \\ \mathit{aux}(\delta \mapsto \theta, \Theta) = \mathit{aux}(\Theta) & \text{if } \mathit{aux}(\theta) = \bullet \\ \mathit{aux}(\bullet) = \bullet \end{cases}$$

$\mathit{name}(\mathtt{d}) = \mathtt{d} \qquad \mathit{args}(\mathtt{d}) = \overline{\mathtt{x}} \quad \text{if } \mathtt{def\ d}(\overline{\mathtt{x}})\ \{\mathtt{e}\} \qquad \mathit{body}(\mathtt{d}) = \mathtt{e} \quad \text{if } \mathtt{def\ d}(\overline{\mathtt{x}})\ \{\mathtt{e}\}$

$\mathit{name}((\overline{\mathtt{x}}) \overset{\tau}{\mathtt{=>}} \mathtt{e}) = \tau \qquad \mathit{args}((\overline{\mathtt{x}}) \overset{\tau}{\mathtt{=>}} \mathtt{e}) = \overline{\mathtt{x}} \qquad \mathit{body}((\overline{\mathtt{x}}) \overset{\tau}{\mathtt{=>}} \mathtt{e}) = \mathtt{e}$

$$\phi_0[\phi_1] = \phi_2 \text{ where } \phi_2(\delta) = \begin{cases} \phi_1(\delta) & \text{if } \delta \in \mathbf{dom}(\phi_1) \\ \phi_0(\delta) & \text{otherwise} \end{cases}$$

Syntactic shorthands:

$\delta; \overline{\pi}(\Theta); \sigma \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta}$ where $|\overline{\mathtt{e}}| = n$ for $\delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 \cdots \delta; \pi_n(\Theta); \sigma \vdash \mathtt{e}_n \Downarrow \theta_n$

$\rho(\overline{\theta})$ where $|\overline{\theta}| = n$ for $\rho(\theta_1), \ldots, \rho(\theta_n)$

$\overline{\mathtt{x}} := \rho(\overline{\theta})$ where $|\overline{\mathtt{x}}| = n$ for $\mathtt{x}_1 := \rho(\theta_1) \ \ldots \ \mathtt{x}_n := \rho(\theta_n)$

Rules for expression evaluation: $\boxed{\delta; \Theta; \sigma \vdash \mathtt{e} \Downarrow \theta}$

$$\frac{}{\delta; \Theta; \sigma \vdash \ell \Downarrow \ell\langle\rangle}\ [\text{E-LOC}] \qquad \frac{\phi' = \phi|_{\mathbf{dom}(\Theta) \cup \{\delta\}}}{\delta; \Theta; \sigma \vdash \phi \Downarrow \phi'\langle\rangle}\ [\text{E-FLD}]$$

$$\frac{\delta; \overline{\pi}(\Theta); \sigma \vdash \overline{\mathtt{e}}, \mathtt{e} \Downarrow \overline{\theta}, \theta \qquad \mathtt{b} = \rho(\theta) \qquad \mathtt{v} = (\!|\mathtt{b}|\!)_{\delta}^{\pi^{\mathtt{b}}(\Theta), \sigma}(\rho(\overline{\theta}))}{\delta; \Theta; \sigma \vdash \mathtt{e}(\overline{\mathtt{e}}) \Downarrow \mathtt{v}\langle\overline{\theta}, \theta\rangle}\ [\text{E-B-APP}]$$

$$\frac{\begin{array}{c}\delta; \overline{\pi}(\Theta); \sigma \vdash \overline{\mathtt{e}}, \mathtt{e} \Downarrow \overline{\theta}, \theta \qquad \mathtt{f} = \rho(\theta) \text{is not a built-in} \\ \delta; \pi^{\mathtt{f}}(\Theta); \sigma \vdash \ \mathit{body}(\mathtt{f})[\mathit{args}(\mathtt{f}) := \rho(\overline{\theta})] \Downarrow \theta'\end{array}}{\delta; \Theta; \sigma \vdash \mathtt{e}(\overline{\mathtt{e}}) \Downarrow \rho(\theta')\langle\overline{\theta}, \theta, \theta'\rangle}\ [\text{E-D-APP}]$$

$$\frac{\Theta_1 = \pi_1(\Theta) \qquad \delta; \Theta_1; \sigma \vdash \mathtt{e} \Downarrow \theta_1 \qquad \phi = \rho(\Theta_1)[\delta \mapsto \rho(\theta_1)]}{\delta; \Theta; \sigma \vdash \mathtt{nbr}\{\mathtt{e}\} \Downarrow \phi\langle\theta_1\rangle}\ [\text{E-NBR}]$$

$$\frac{\begin{array}{l}\delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 \\ \delta; \pi_2(\Theta); \sigma \vdash \mathtt{e}_2[\mathtt{x} := \ell_0] \Downarrow \theta_2\end{array} \qquad \ell_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \mathbf{dom}(\Theta) \\ \rho(\theta_1) & \text{otherwise} \end{cases}}{\delta; \Theta; \sigma \vdash \mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}_2\} \Downarrow \rho(\theta_2)\langle\theta_1, \theta_2\rangle}\ [\text{E-REP}]$$

$$\frac{\delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta_1 \qquad \ell_0, \ell_1 = \begin{cases} \pi_1(\Theta), \mathtt{e}_1 & \text{if } \rho(\theta_1) = \mathtt{True} \\ \pi_2(\Theta), \mathtt{e}_2 & \text{if } \rho(\theta_1) = \mathtt{False} \end{cases} \qquad \delta; \ell_0; \sigma \vdash \ell_1 \Downarrow \theta}{\delta; \Theta; \sigma \vdash \mathtt{if}(\mathtt{e})\{\mathtt{e}_1\}\{\mathtt{e}_2\} \Downarrow \rho(\theta)\langle\theta_1, \theta\rangle}\ [\text{E-IF}]$$

$$\frac{\begin{array}{l}\delta; \pi_i(\Theta); \sigma \vdash \mathtt{e}_i \Downarrow \theta^i \text{ for } i \in 1{,}2{,}3 \\ \mathtt{k}_1, \ldots, \mathtt{k}_n = \rho(\theta^2) \cup \bigcup\{\mathbf{dom}(\pi_4(\Theta(\delta'))) \text{ for } \delta' \in \mathbf{dom}(\Theta)\} \\ \delta; \pi^{\mathtt{k}_i}(\pi_4(\Theta)); \sigma \vdash \rho(\theta_1)(\mathtt{k}_i, \rho(\theta_3)) \Downarrow \theta_i \text{ for } i \in 1, \ldots, n\end{array}}{\delta; \Theta; \sigma \vdash \mathtt{spawn}(\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3) \Downarrow F(\overline{\mathtt{k}} \mapsto \rho(\overline{\theta}))\langle\theta^1, \theta^2, \theta^3, F(\overline{\mathtt{k}} \mapsto \overline{\theta})\rangle}\ [\text{E-SPAWN}]$$

Figure 3.3: Big-step operational semantics adapted from [18, 22].

System configurations and action labels:

Ψ	$::=$	$\overline{\delta} \mapsto \overline{\Theta}$	status field
τ	$::=$	$\overline{\delta} \mapsto \overline{I}$	topology
Σ	$::=$	$\overline{\delta} \mapsto \overline{\sigma}$	sensors-map
Env	$::=$	τ, Σ	environment
N	$::=$	$\langle Env; \Psi \rangle$	network configuration
act	$::=$	$\delta \mid env$	action label

Environment well-formedness:

$WFE(\tau, \Sigma)$ holds iff $\mathbf{dom}(\tau) = \mathbf{dom}(\Sigma)$ and $\tau(\delta) \subseteq \mathbf{dom}(\Sigma)$ for all $\delta \in \mathbf{dom}(\Sigma)$.

Transition rules for network evolution: $\boxed{N \xrightarrow{act} N}$

$$\text{[N-FIR]}\quad \frac{Env = \tau, \Sigma \quad \tau(\delta) = \overline{\delta} \quad \delta; F(\Psi)(\delta); \Sigma(\delta) \vdash \mathtt{e_{main}} \Downarrow \theta \quad \Psi_1 = \overline{\delta} \mapsto \{\delta \mapsto \theta\}}{\langle Env; \Psi \rangle \xrightarrow{\delta} \langle Env; F(\Psi)[\Psi_1] \rangle}$$

$$\text{[N-ENV]}\quad \frac{WFE(Env') \qquad Env' = \tau, \overline{\delta} \mapsto \overline{\sigma} \qquad \Psi_0 = \overline{\delta} \mapsto \emptyset}{\langle Env; \Psi \rangle \xrightarrow{env} \langle Env'; \Psi_0[\Psi] \rangle}$$

Figure 3.4: Small-step operational semantics for network evolution from [18].

Chapter 4

FCPP

FCPP (FieldCalc++) [8, 17] is a C++14 library that implements the field calculus as an internal DSL. Its main advantage compared to existing implementation is that it doesn't need a JVM but can be compiled directly for the hardware it runs on. FCPP is also optimized for performance, leveraging C++ metaprogramming, and optimized message size.

4.1 Architecture

FCPP architecture is organized logically in three layers visible in Figure 4.1.

The first layer is the data layer, which defines the data structure for memory efficient representation of data (like tuples and vectors) and for representing neighbouring fields.

The second layer defines the functionalities available and how the programs are executed with a component based architecture that allows it to be customizable for different application scenarios. Each component has a structure visibile in Figure 4.2, in a *mixin*-like fashion [20]. The component receives configuration options as template parameters `Ts`, then the components subclass receives as template parameters both the final composition of all components `F` (so it can be used in its typings) and the parent component to extend `P`. Inside the functionalities of each component are further separated in two classes: the *node* class describes the abstractions for a single device, while the *net* class defines the overall network orchestration. The combination of the components that need to be used for a program is usually defined using the macro `DECLARE_COMBINE(name, component1, ..., componentn)` which defines a combination of components named `name` that can be used to construct a *net* object that will run the program. Various combinations of components are pre-defined for common use cases, e.g. running batch simulations.

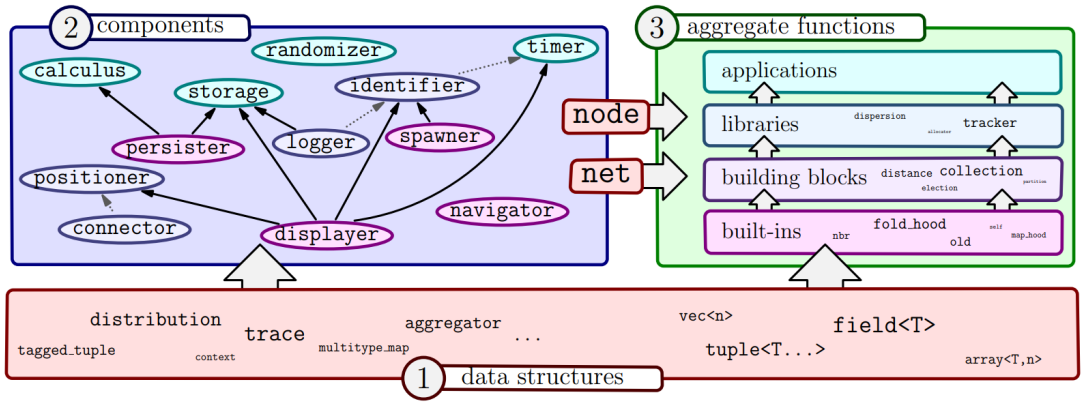

Figure 4.1: Layers of the FCPP architecture, from [8]. In the second layer the components are further categorized by color, with cyan indicating the general purpose components, violet pertaining to simulations or cloud-based applications components and magenta the simulation specific components. Dependencies between components are displayed using arrows. In particular solid arrows represent a hard dependency, meaning that the pointed components must be an ancestor of the other, while dotted arrows represent soft dependencies, meaning that the pointed component is not required but if present it must be an ancestor of the other.

```
template <class... Ts>
struct my_functionality {
    template <class F, class P>
    struct component : public P {
        class node : public P::node { ... };
        class net : public P::net { ... };
    }
}
```

Figure 4.2: Structure of FCPP components.

aggregate function declaration

$\texttt{F} ::= \texttt{FUN}\ T\ \texttt{d}(\texttt{ARGS}, T\ \texttt{x}*)\ \{\texttt{CODE return e;}\}$

aggregate expression

$\texttt{e} ::= \texttt{x} \mid \ell \mid T\{\texttt{e}*\} \mid \texttt{ue} \mid \texttt{e o e} \mid \texttt{p}(\texttt{e}*) \mid \texttt{node.c}(\texttt{e}*) \mid \texttt{d}(\texttt{CALL}, \texttt{e}*)$
$\mid T\ \texttt{x} = \texttt{e};\ \texttt{e} \mid [\&](T\ \texttt{x}*)\texttt{->}T\{\texttt{return e;}\} \mid \texttt{e ? e : e}$

type	aggregate function
$T ::= t \mid \texttt{bool} \mid tt\langle T*, \ell* \rangle$	$\texttt{d} ::= \texttt{b} \mid \texttt{d}$

built-in aggregate functions

$\texttt{b} ::= \texttt{old} \mid \texttt{nbr} \mid \texttt{spawn} \mid \texttt{self} \mid \texttt{mod\_self} \mid \texttt{map\_hood} \mid \texttt{fold\_hood} \mid \texttt{mux}$

Figure 4.3: Syntax of FCPP aggregate functions, from [8].

The third layer define the aggregate functions available for the aggregate programs as a DSL. This layer provides the basic coordination constructs like *nbr* and *rep*. In particular it provides the built-in aggregate functions `old` and `nbr`. The `old` function acts like the *rep* construct with many overloads to simplify the usage in common situations. The `nbr` function acts like the *nbr* construct or the *share* construct from [9], based on the overloading used. FCPP also defines many built-in operators to manipulate fields, like `self` to extract from a field the value for the current node, `map_hood` to apply a point-wise function to the values of a field or `fold_hood` to reduce the values of a field into a single value using a commutative and associative binary function. The syntax of aggregate functions in FCPP is given in Figure 4.3. Every FCPP program is a valid C++ program, meaning that all the features of C++ are available. The keywords `FUN`, `ARGS`, `CODE` and `CALL` in Figure 4.3 are macros that define the extra code necessary to define aggregate functions while hiding the implementation details from the developers.

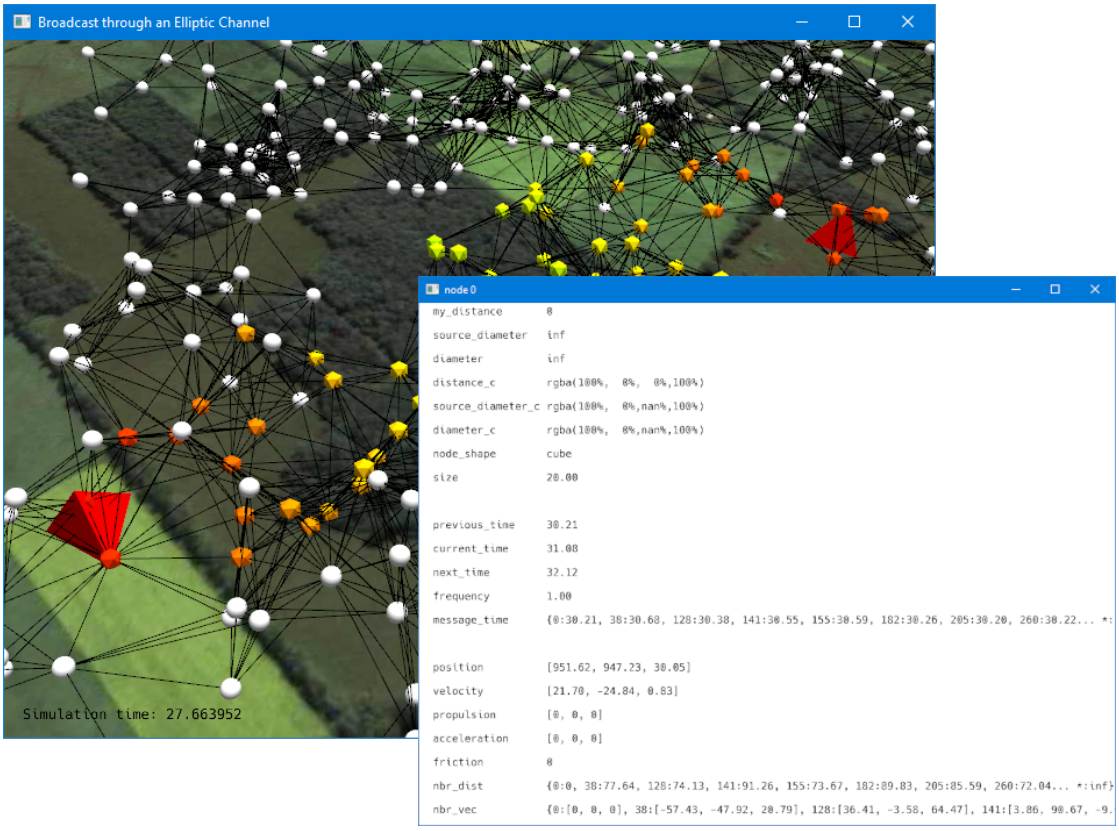

Figure 4.4: Screenshot of the FCPP simulator in action, from the FCPP website.

4.2 Simulator

FCPP includes a simulator with a user interface capable simulating 2D or 3D environment with obstacles and 3D visualization of the simulation [16]. Figure 4.4 shows an example screenshot from the simulator. The simulator window is able to show all the nodes in the simulation and their connections. The simulation can be sped up, slowed down or paused by the user, and at any time each node can be inspected to obtain more information on its state. It is also able to run in single-threaded or multi-threaded modes. The user can interact with the GUI, implemented with the graphical library OpenGL, to navigate, zoom and rotate the visualization. Batch simulations without the GUI are also supported. Thanks to the modular architecture of FCPP the simulator is fully configured as any other component, in particular the `batch_simulator` and `interactive_simulator` combinations of components provide all the components needed to run respectively a batch and a graphical simulator. For both of them all aspects of the simulation can be configurated, like the number and behavior of the nodes, the duration of

```
#include "lib/fcpp.hpp"
FUN double abf(ARGS, bool source) { CODE
    return nbr(CALL, INF, [&] (field<double> d) {
        double v = source ? 0.0 : INF;
        return min_hood(CALL, d + node.nbr_dist(), v);
    });
}
```

Figure 4.5: Example aggregate function that estimates distances from devices where source is true using the Adaptive Bellman Ford algorithm [15], from [8].

```
MAIN() { ... }
DECLARE_OPTIONS(example_options, fcpp::options::program<main>, ...)
int main() {
    using net_t = fcpp::component::batch_simulator<example_options>::net;
    net_t network{};
    network.run();
    return 0;
}
```

Figure 4.6: Example setup to run a simulated FCPP aggregate program.

the simulation, the communication range, the background texture and the visual aspect of the nodes based on their internal status.

4.3 Usage

Figure 4.5 shows an example aggregate function implemented in FCPP that estimates the distance from devices where `source` is true. In the example `node` is a variable implicitly provided by the `ARGS` macro. The function `nrb_dist` its a built-in that returns a `field<double>` of the estimated distances from neighbour nodes. The function `min_hood` is a build-in function that returns the minimum value in a field.

Figure 4.6 provides an high level overview of the setup needed to run an FCPP program. More specific examples can be found in the FCPP GitHub repository. To write a full aggregate program, e.g. a program that uses the example `abf` aggregate function, a main function should be defined using the `MAIN` macro like `MAIN() { ... }`. The main function can call any desired aggregate function or operator. The configuration options, e.g. how often the rounds are executed or which main program to execute, need to be collected using the `DECLARE_OPTIONS` macro like `DECLARE_OPTIONS(example_options, ...)` which groups all the configurations

in a variable named as the first argument, i.e. `example_options` in this example. In the Figure 4.6 the options declare that the program to execute is contained in the structure called `main` generated by the `MAIN` macro. In the example the main function of the C++ program constructs a `net` object in a variable named `network` using the `batch_simulator` component type, i.e. a combination of components specialized in running batch simulation of aggregate programs predefined using the `DECLARE_COMBINE` macro. The function `run` of the `net` instance runs the simulation until its end. Ready to use sample projects are provided in a repository[1] with documentation on how to use them and on what they do.

[1] https://github.com/fcpp/fcpp-sample-project

Chapter 5

Research Project

The goal of the research project was to apply the Aggregate Programming paradigm to industrial case studies using real world IoT platforms for the first time. We identified an IoT module offering sensors and low range communication technologies called DWM1001, which Reply was already using in an RTLS (real time location system) application for an industrial client. This contributions achieve for the first time the ability to run aggregate programs on real world IoT devices. Reply also helped us identify a case study of smart warehousing based on their experience in the industry.

The researh project consisted of the following steps:

- Learning of how to use the hardware and tools. Thanks to the previous usage in Reply we had a foundation of tools already in use to build applications for the DWM1001 module. A toolchain was already developed to build applications based on the operative system Contiki OS, an operative system focused on resource-constrained embedded IoT systems. We also had available APIs to exchange messages using the Ultra Wideband (UWB) transceiver and to compute the distance between two devices.

- Development of support for C++ and Contiki-NG on the DWM1001 module. The Contiki version used by the toolchain was old and no longer maintained, so we updated the toolchain to the latest version of Contiki-NG. The build also didn't support C++, which was needed in order to use FCPP.

- Porting of FCPP on the DWM1001 module using the Ultra Wideband and Bluetooth Low Energy radios. We updated FCPP to fully support the customizations needed by the plaftorm and then we implemented drivers to run FCPP programs on the DWM1001 with Contiki. We supported exchanging messages by either BLE or UWB.

- Development of a case study of smart warehousing. In this case study we suppose that each forklift and each pallet in the warehouse are equipped with a DWM1001 module. All the devices cooperate to (i) collect logs of the operation performed in the warehouse, (ii) provide routes to retrieve goods or deposit new pallets and (iii) provide warnings to prevent collisions between moving forklifts. We run simulations of this case study.

- Development of a prood-of-concept experiment in Reply's laboratory of the proposed case study. We used DWM1001 modules to run a small scale experiment with which we validated that the program is actually able to run on the hardware with satisfying results.

The source code for the C++ support on the DWM1001 and for the porting of FCPP is currently proprietary software owned by Reply and not publicly available, while the source code for the case study is open source.

The following chapters illustrate in detail all the steps of the research project.

Chapter 6

Existing Technologies

A key component of the research project was to enable applications written with the Aggregate Programming paradigm to run on real world devices, in particular the ones found in industrial use cases. We identified an IoT device called DWM1001 that they were already using in industrial plants for running applications based on distributed data and distributed computations. One of the key features of this module is a Ultra Wideband (UWB) transceiver, i.e. support for a radio technology which allows for short-range communication and can be used to compute the distances between devices with a precision of few centimeters [23]. This module has a small size and weight at a price point under 30 euro per unit, making it ideal to be used as a wearable device (e.g. in badges). In particular we supported running FCPP applications on DWM1001 using the Contiki-NG operative system (OS). Supporting different IoT OSes would require to develop new integrations with each OS execution model for the parallel execution of the aggregate programming rounds and the handling of the message exchange.

6.1 DWM1001

The DWM1001 [3] is a module produced by Qorvo, composed of the Nordic Semiconductor nRF52832 general-purpose system on a chip (SoC), the Qorvo DW1000 Ultra Wideband (UWB) transceiver, Bluetooth and UWB antenna, and the STM LIS2DH12TR 3-axis accelerometer. It has a size of 19.1mm x 26.2mm x 2.6mm [2].

The DWM1001 module is sold in two variations: the DWM1001C, i.e. a version of the module certified to ETSI, FCC and ISED regulations and available ready to use; and the DWM1001-DEV, i.e. the development board. During the development our software was tested on the DWM1001-DEV development board due to ease of use and availability in Reply.

The DWM1001-DEV module compared to the DWM1001C includes the following additional features:

- an additional USB connection for reprogramming, debugging and power supply. The J-Link software is able to use this connection to load the built *.hex* files on the module;
- an on board J-Link debugger. This debugger allows to asyncronously collect logs on the connected PC using the J-Link software. It is also possible to connect the standard C or C++ debugger *gbc* to put stops in the execution or read the content of variables at runtime. This feature was particularly useful during the development to detect memory issues resulting in hard faults;
- a reset button, not configurable by the user for different actions;
- a user programmable button, which sends an interupt on which any action can be performed;
- a Raspberry Pi compatible header, not used in our development;
- 8 LEDs, one showing if the battery is charging, one showing if the on-board J-Link is active, one showing if the UWB radio is transmitting, one showing if the UWB radio is in listening mode and the 4 remaining LEDs for custom information from the user software.

The development module can be powered by batteries in the range from 3.6V to 5.5V, while the DWM1001 recommends an input voltage from 2.8V to 3.6V just for the module itself. In particular during our development we powered the development board and the custom Reply board (see Chapter 6.1.1) with 3.7V 950mAh batteries.

Qorvo also provides a ready to use RTLS software, allowing the modules to be run as *tags* or *gateways*, which collect information about the distances between devices. This software has not been included in our experimentation due to it not being much customizable and closed source.

The nRF52832 [6] SoC integrated in the DWM1001 offers a 64 MHz ARM Cortex-M4 CPU with floating-point unit (FPU), a 512 KB flash memory, a 64 KB RAM and a Bluetooth Low Energy (BLE) 2.4 GHz transceiver. The BLE transceiver is configurable at different transmission power levels, from -20 to +4 dBm, data rates of 1Mbps and 2Mbps, and includes support for the Bluetooth 5 protocol. Nordic Semiconductor offers also a Software Development Kit (SDK), called nRF5 SDK, which provides many C libraries to easily interact with the hardware capabilities of the SoC. The Bluetooth APIs are a separated part of the SDK called SoftDevice and are provided as a precompiled and linked binary

Figure 6.1: Reply Custom Board with DWM1001C

software. In particular the SoftDevice S132 provides support for the SoCs of the nRF52 series like the nRF52832 SoC.

The DWM1000 [1] module is a Ultra Wideband low-power transceiver compliant with the IEEE 802.15.4-2011 UWB specification. It supports six channel of comunication (with only the channel 5 certified to perform according to specifications), three different data rates (110 kbps, 850 kbps and 6.8 Mbps) and a ranging (i.e. computing the distance between two devices by excending messages) accuracy as low as 10cm. It supports sending packets of size up to 1023 bytes, with an advertised range of communication of up to 290 meters.

The DWM1001 module was choosen for this research project in virtue of being at that time one of the few low-cost Ultra Wideband industry ready module available on the market. The UWB radio technology is particularty important for real time location systems (RTLS) thanks to the capability of computing distances and triangulate positions with a high accuracy (up to 10cm with the DW1000) compared to the Bluetooth Low Energy technology [26].

6.1.1 Reply Custom Board

At the time of the start of the research project the DWM1001 module was in use in Reply for an RTLS system. Hundreds of devices were being used in a project for a big italian company in the beverage sector to track employees coming in close contact during the COVID-19 pandemic inside the industrial plants.

The DWM1001 module was integrated in a custom board developed specifically for the project mounted inside the employee badge. The custom board included a buzzer and an additional 32KB flash memory. During the research project the custom board was used to validate experimentally a case study, due to the higher number of available modules in the laboratory compared to the DWM1001-DEV modules.

6.2 Contiki

Contiki [31, 24] is an open source operative system (OS) since 2006 written in the C programming language and focused on running on resource-constrained embedded IoT systems. It advertises a code footprint in the order of 100 kB and a runtime RAM usage under 10 kB. The Contiki OS is based on an event-based kernel, where different cooperative processes are event handlers that always run to completion without maintaining a separate execution stack. Every process in Contiki is based on the concept of Protothreads. Protothreads [25] are a programming abstraction that allows to write event based programs in a thread-like style. Protothreads are lightweight since they don't require their own stack and are implemented in Contiki using C macros that translate to regular C switch statements. Contiki also includes APIs for efficient memory allocation in blocks, dynamic load of programs, network communication with different pre-implemented protocols, syncronization primitives, timers, LEDs and data structures. The Contiki project was formerly known as Contiki OS but since 2017 development of the original project was interupped and forked in the Contiki-NG (Next Generation) project in order to break retro compatibility. Contiki NG keeps the support for multiple platforms but focusses more on targeting ARM Cortex-M platforms.

6.2.1 Contiki Processes

In Contiki the user code is run in two execution contexts. Processes are run in a cooperative context, meaning that processes run sequentially, are never interrupted and must yield control intentionally to other processes (e.g. when waiting for an event). Interrupts (i.e. signals received from the processor or sensors) and real-time timers handlers are run in a preemptive context, meaning that they will stop the execution of the current process until they have completed.

The Contiki scheduler maintains in memory for each process a lightweight structure called *process control block*, which contains all the information needed to resume the execution of the process after a context switch. The active processes are kept in a linked list and executed when their waiting condition is met, usually by an asyncronous event created by another process or an interrupt. The processes ready for execution are choosen to run using a round robin policy by Contiki, with the possibility to schedule a round of execution with higher priority when needed by a process. Processes can keep a state between rounds of execution using static variables.

Figure 6.2 shows an example of a simple Contiki program. In this example the program is composed of a single process called `example_process` that never terminates. First all processes are declared using the `PROCESS` macro in order to allocate

```
#include "contiki.h"
PROCESS(example_process, "Example Process Name");
AUTOSTART_PROCESSES(&example_process);
PROCESS_THREAD(example_process, ev, data) {
    static struct etimer et;
    PROCESS_BEGIN();
    while (1) {
        etimer_set(&timeout, CLOCK_SECOND * 5);
        PROCESS_YIELD_UNTIL(etimer_expired(&timeout));
        LOG_INFO("Timer expired\n");
    }
    PROCESS_END();
}
```

Figure 6.2: An example of a Contiki program.

a static structure that will host the state of the process. The `AUTOSTART_PROCESSES` macro can be used only once and defines the list of processes to start when Contiki starts running on the device. Then, using the macro `PROCESS_THREAD` the code of the process is defined, wrapped around the `PROCESS_BEGIN` and `PROCESS_END` macros. In this example the process runs an infinite loop in which sets a timer 5 seconds long using the APIs of Contiki, waits until the timer is expired, prints a log message and restarts the loop. The macro `PROCESS_YIELD_UNTIL` returns control to the Contiki scheduler, which will not run this process again until an event will be sent to it. Every time an event will wake up the process it will check if the condition given to `PROCESS_YIELD_UNTIL` is satisfied, in which case the execution will restart from the instruction following the yield. The stack of the process is not restored between executions, meaning that the timer variable *et* needs to be declared as `static`. Other processes are free to run while this process is waiting.

6.2.2 Contiki for DWM1001

We decided to use Contiki for the research problem since it was already in use by Reply on the DWM1001 module. In particular we used the open source Contiki-based software package for the DecaWave EVB1000/DWM1001 platforms [23], also called Contiki-UWB, developed by the D3S Research Group of the University of Trento.

The software package included the porting of Contiki OS for the DWM1001 platform, support for USB logging with `printf`, APIs for the UWB radio and an implementation of a two-way ranging algorithm using UWB. The platform specific code utilize the Nordic nRF5 SDK to interact with the nRF52832. The software is also configured to allow to send log messages via USB using the SEGGER Real

Time Transfer (RTT) APIs available for the J-Link debugger integrated in the DWM1001-DEV board. The Bluetooth APIs have not been integrated with the Contiki network stack.

Contiki NG, i.e. the latest version of Contiki, was not supported at the time by the Contiki UWB software. The previous Reply project also utilized the Janus [27] software. Janus is a Contiki-UWB based application that computes a distributed table of distances between devices with high precision and low energy consumption. Janus was not utilized in the research project due to being a paid and closed source project.

The software package is based on the GNU ARM Embedded Toolchain [4], in order to compile the code targetting the ARM Cortex-M4 CPU of the nRF52832. It also requires the Nordic nRF5 SDK as an internal dependency and the SEGGET J-Link software to flash the software on the DWM1001 module. Contiki is included as-is using a Git submodule without customizations.

Applications written for Contiki UWB are composed by: a Makefile that configures a set of variables needed by the Contiki build files and then includes the Contiki UWB Makefile, an optional Makefile that specifies the target device, a C header file called *project-conf.h* that defines some project dependent variables as C macros, one or more *.c* files containing the definition of the Contiki processes.

Building the application generates a *.hex* file that can be flashed on any DWM1001 device and will be execute when the device starts. Multiple example applications are provided in the repository.

Chapter 7

Technologies Developed

7.1 Contiki-NG support

Since 2017 Contiki OS is no longer under development and a fork called Contiki-NG has been officially supported instead. This was done by the developers in order to break retro-compability and drop support for old 8-bit or 16-bit platforms.

One of the first steps we decided to take before working on the FCPP porting was to upgrade the support for the DWM1001 module from Contiki OS to Contiki-NG, in order to have access to the latest APIs and bug fixes. The Contiki-NG support was implemented in a fork of Contiki UWB called Contiki-NG UWB.

The migration work consisted of the following steps:

- update the Makefile configurations to the build system of Contiki-NG. In Contiki-NG the Makefiles have been reorganized to facilitate code reuse and many platform specific configuration were already loaded by a new Makefile common for all ARM Cortex-M4 CPUs;
- update the platform specific code to use the new API, in particular the platform specific code didn't need anymore to define the C main function. Instead in a new *platform.c* file, replacing the old *contiki-main.c* file, four new C functions needed to be implemented: `platform_init_stage_one`, `platform_init_stage_two`, `platform_init_stage_three` and `platform_main_loop`, each responsible to perform the correct initialization at the right time according to the Contiki specifications and to implement the main execution loop;
- remove usage of old network APIs that were removed in Contiki-NG. One of the big changes from Contiki OS was the new network API, more generic to support different means of communication and protocols. Old non-standard protocols were also removed;

- migration of the custom logging macros to the new standard logging API of Contiki-NG;
- update Contiki UWB examples to compile with the new Makefiles and API;
- update SEGGER RTT [5] (i.e. the API used by Contiki UWB to deliver log messages through a USB connection) to the latest version, which provided better performance and stability for the debugging process.

Support to organize files in subfolders in the user applications was also added in the Makefile.

Contiki-NG UWB application are developed in the same way as Contiki UWB applications and various examples are included in the repository.

7.2 C++ support

In order to run FCPP on DWM1001 we needed first to update the build to support the compilation of C++ programs. The Contiki-UWB build is configured to use the GNU ARM Embedded Toolchain [4], i.e. a collection of open source tools for C and C++ programs, including the GNU C Compiler for devices with an ARM architecture, like the DWM1001.

The platform specific Contiki-UWB Makefiles were updated to included the *.cpp* files and to compile them with the standard GNU++14. In order to support the C++ features utilized by FCPP some platform specific code needed to be implemented. In particular in order to support the C++ standard output stream we redirected it to a custom stream which sends the characters to the SEGGER RTT output. We also implemented a custom implementation of the clock API in the std::chrono namespace that uses the nRF5 SDK API. All the C++ support features that we implemented can be enable using the `UWB_CONTIKI_CPP` flag in the user Makefile.

7.3 Multithreading

Since FCPP was based on a multithreaded execution model, with one thread handling the message exchanges and one thread running periodically the aggregate program, we explored how to support it on the single threaded architecture of the nRF52832.

The Nordic nRF5 SDK offers an experimental library called Task Manager. The Task Manager library allows to define a number of parallel tasks with a fixed stack size (configured at compile time in the Makefile) and run with a cooperative

round robin scheduler. The library internally splits the available RAM in chunks of equal sizes and assigned each chunk to a tasks. Tasks can be dynamically created and terminated but a fixed number of maximum concurrent tasks must be set at compile time to organize the memory. We implemented the C++ thread and mutex APIs using the Task Manager library, with one additional always active task executing the Contiki-NG kernel main loop, meaning that all the Contiki processess will run in a single task.

Thanks to the multithreading support we run FCPP using two tasks, one responsible of the program execution and switching task after each round or when idle, one responsible to the processing of the received messages. Unfortunatly the memory division didn't leave enough space to each task to satisfy the memory requirements of FCPP for non trivial applications.

Since the usage of the DWM1001 was a requirement from Reply we had no choice but to update FCPP to optionally avoid the usage of thread and mutex APIs and we run the FCPP parallel execution using Contiki processes. Removing the multi-threading requirement from FCPP was also a useful improvement for the software, since it will allow it in the future to be ported to more single-threaded platforms similar to the DWM1001 without losing the FCPP core features.

7.4 Ranging

Various use cases for the Aggregate Programming paradigm require the devices to be able to compute the distance from their neightbours, e.g. RTLS systems that triangulate their position from the distances from anchors and neightbours. The task of computing the distance between two devices is called ranging.

The Ultra Wideband (UWB) radio technology is able to compute the distance from a simple message exchange with a precision up to 10 centimeters according to our experiments. In particular we measured using the DWM1001-DEV module a range of communication in the offices aisles of around 70 meters (while we achieved only 25 meters with BLE) and a ranging error always under 30 centimeters. The ranging algorithm was already implemented in the Contiki UWB project and consists in particular of the Double-sided Two-way Ranging (DS-TWR) algorithm [7]. The device (D1) initiating the ranging broadcasts a message with the current timestamp and the address of the device that it wants to range with (D2). When the target device D2 receives the message it transmit the response and D1 computes the distance based on the time consumed by the exchange. Then D1 sends one more message so that also D2 can compute the distance. The protocol is managed by a Contiki process, after the received messages are consumed and stored by interrupts, that maintains the state of the ranging exchanges.

The Single-sided Two-way Ranging (SS-TWR) algorithm was also already im-

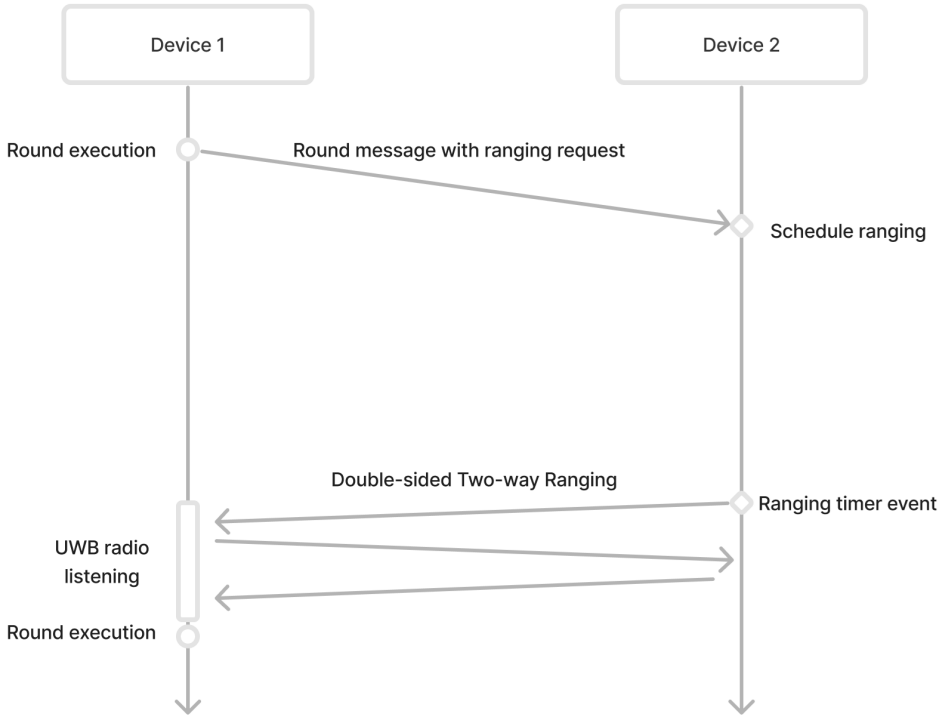

Figure 7.1: Message exchange over time to perform ranging. Device 1 puts in its round message the information that it wants to perform ranging with device 2. When device 2 receives the message it schedules an event to perform the ranging at the right time. Device 1 turns on the UWB radio right before the next round of execution in order to perform the ranging.

plemented. Like for the DS-TWR the device (D1) initiating the ranging broadcasts a message with the current timestamp and the address of the device that it wants to range with (D2). When the target device D2 receives the message it transmit the response and D1 computes the distance based on the time consumed by the exchange. Unlike the DS-TWR the device D1 doesn't send an additional message, meaning that only D1 will know the distance from D2 but not vice versa. Since for the Aggregate Programming paradigm all devices are interested in knowing the most accurate distance possible from all neighbours we decided that using the DS-TWR algorithm was more suitable since it introduces less overhead compared to performing two SS-TWR to inform both devices of their distance.

7.4.1 Ranging for FCPP

To perform the ranging during the execution of a FCPP program we implemented a protocol that determines when and how often the ranging is performed. The protocol aims at minimizing the energy consumpion, in particular by keeping the UWB radio in sleep mode as often as possible. More precisely, the devices perform the following steps:

- when a node sends a message at the end of its round of execution, it adds a prefix with a list of neighbour nodes it wants to invite to do a ranging session at the beginning of the next round. The maximum number of devices is limited by a fixed value in order to keep enough space for the actual message content and the devices are choosen by preferring the devices with the oldest information.
- the node then prepares to do ranging with the neighbours in the list right before the beginning of its next round, so that the values will be as updated as possible. The radio is turned on just for the time necessary to perform in sequence all the ranging operations requested.

To handle the ranging requests the devices also perform the following steps:

- when a node receives a message with the round execution data of another device it checks if its address appears in the prefix list of the message. If it does it schedules a timer to perform the ranging. It will know the correct instant to schedule in virtue of all devices having the same round interval.
- when a ranging timer expires it performs the ranging with the device that requested it.

This protocol doesn't account for collisions. In case of collisions the ranging will fail and the device will be choosen again in the future since devices with the oldest ranging data are preferred by the algorithm. Figure 7.1 shows the message exchange to perform a ranging operation from the point of view of the device requesting the ranging (Device 1).

7.5 FCPP porting on DWM1001

To implement a FCPP driver to support the DWM1001 platform we first needed to make some changes to the FCPP codebase, generalizing some API to meet the needs of our platform. In particular we needed to implement the following changes. We added an option to completly avoid the usage of C++ exceptions in FCPP since they where not supported by the hardware. We added an option to use a

custom clock implementation to allow FCPP to get the time from the DWM1001 clock. We added an option to use a custom output stream in order to have a more efficient implementation using internally the Nordic nRF5 SDK APIs. We added a field of the current neightbours identifiers in order to use it in the ranging implementation. Finally we generalized the definition of the `hardware_connection`, i.e. the FCPP component responsible for the handling of the message exchanges, in order to support our implementation.

The support to run FCPP program on DWM1001 was implemented in a separate project called FCPP-Contiki, which uses internally Contiki-NG UWB as a Git submodule and requires the same tools. Applications for FCPP-Contiki have the same project structure of applications for Contiki-NG UWB and examples are included in the repository. We also implemented some wrappers around the Contiki API needed by the driver implementations since Contiki didn't include C++ compatible header files, in particular for the network API.

7.6 Example FCPP-Contiki program

Figure 7.2 shows a basic FCPP configuration for running an FCPP-Contiki application. Like in a regolar FCPP program the configuration is declared with the `DECLARE_OPTIONS` macro, in this case using the predefined `base_fcpp_contiki_opt` as the base options, which pre-configures the BLE driver and the output stream. The usage of the `FCPP_CONTIKI` macro is mandatory and generates all the code needed by the contiki process to run the aggregate program. It takes in input the previously declared configuration and additional optional settings, for example the driver settings for the radio power or the number of ranging performed per round.

Figure 7.3 shows a basic Contiki process that starts the execution of the aggregate program. It declares a process called `app_process` and declares with the `AUTOSTART_PROCESSES` that it will be executed when the device is turned on. This process performs only the action of calling the `start_fcpp` function, which in turn will start a new pre-implemented Contiki process responsible of executing the aggregate program. The aggregate program, not shown in the Figure, is declared using the regular FCPP macro `MAIN`.

7.7 UWB and BLE drivers

We implemented three different drivers for FCPP:

- the BLE driver is the default driver and uses the BLE radio for the aggregate messages and the UWB radio for ranging;

```
#include "fcpp-contiki-api.hpp"
#include "lib/fcpp_program.hpp"

using namespace fcpp;
using namespace component::tags;
using namespace coordination::tags;

#define ROUND_PERIOD 1 // time in seconds between transmission rounds

// Dictates that messages are thrown away after 5 seconds.
using retain_type = retain<metric::retain<5, 1>>;
// Dictates that rounds are happening every 1 seconds (den, start, period).
using schedule_type =
    round_schedule<sequence::periodic_n<1, ROUND_PERIOD, ROUND_PERIOD>>;

using storage_t = tuple_store<>;

DECLARE_OPTIONS(opt,
    base_fcpp_contiki_opt,
    program<coordination::main>,
    retain_type,
    schedule_type,
    exports<>,
    storage_t
);

FCPP_CONTIKI(component::deployment<opt>, common::make_tagged_tuple<>())
```

Figure 7.2: An example FCPP configuration for FCPP-Contiki.

- the UWB driver uses the UWB radio for the aggregate messages and doesn't perform ranging;

- the Simulation driver doesn't support any radio communication and is used only to run the FCPP simulator directly on the hardware.

In our experiments we powered the module with 1000mAh 3.7V batteries. We measured 12 hours and 15 minutes of battery life when keeping the UWB radio always off and the BLE radio always listening for advertisement at 0dBm. In the same conditions setting the radio power down to -12dBm increased the battery life to more than 50 hours. Turning the UWB radio on 5 times a second for the duration of 16 milliseconds to perform randing, while keeping the BLE radio always listening for advertisement at -12dBm, reduced the battery life to 6 hours and 24 minutes.

```
#include "contiki.h"
#include "fcpp-runner.h"

PROCESS(app_process, "App");
AUTOSTART_PROCESSES(&app_process);
PROCESS_THREAD(app_process, ev, data) {
    static struct etimer et;
    PROCESS_BEGIN();
    PROCESS_PAUSE();
    start_fcpp();
    PROCESS_END();
}
```

Figure 7.3: An example Contiki process to run a FCPP program.

Figure 7.4: Structure of the BLE advertisement payload. One byte contains a time correction to account for the resent messages delay. Two bytes contain the CRC16 code for the error detection. A variable number of bytes contains the UWB address of the device, based on the settings. A single bytes indicates the number N of devices it wants to range with, then for N times the size of an UWB address specifies the address of the devices it wants to range with. Finally the remaining bytes are occupied by the FCPP round message.

7.7.1 BLE driver

The BLE driver uses the Bluetooth 5 extended advertisement to broadcast the messages. All the devices are always listening for advertisement in order to collect all messages. To reduce the risk of collisions all messages are sent a configurable number of times, defaulting to 3, with an additional time correction byte at the beginning of the message so that the recipients know with what delay it was sent. Two additional bytes are dedicated in the message to detect transmission errors using a Cyclic Redundancy Check (CRC) 16 alghoritm implemented by the nRF5 SDK. Additionally a variable message space is consumed by the ranging algorithm based on how many devices it wants to perform ranging with. Using a single extended advertisement message a payload of up to 228 bytes can be included minus the space dedicated to the ranging information, meaning that with the default settings the driver is able to dedicate over 200 bytes for the aggregate program. Figure 7.4 shows the full structure of the broadcasted message.

The BLE transceiver can be configured at different transmission power for different energy consumption and different distances reached by the transmission. The valid power settings for the nRF52832 are -40 dBm, -30 dBm and all values ranging from -20 to +4 dBm in 4 dB step, with 0 dBm as default. In the Reply offices aisles we measured a range of 26 meters for +4 dBm, 25 meters for +0 dBm, 16 meters for -8 dBm and 30 centimeters for -40 dBm. We measured an average energy consumption when listening for BLE advertisement of around 33.70 mA.

All interactions with the BLE radio are implemented using the SoftDevice APIs, which is a set of Bluetooth API from the Nordic nRF5 SDK.

7.7.2 UWB driver and Simulation Driver

The UWB driver uses the UWB radio to broadcast the messages. The UWB transceiver is always in listening mode for all devices, except when transmitting, in order to collect all messages. We measured an average energy consumption when listening for UWB messages of around 110 mA. Due to the high energy consumption and the need to develop ad-hoc algorithms to prevent collisions (compared to BLE advertisement) we didn't developed further the UWB driver. In future work we might bring this driver to feature parity with the BLE driver after implementing some protocol to reduce collision and possibly to avoid keeping the radio always on.

The Simulation driver runs the FCPP simulator directly on the DWM1001 module. It was useful to test the memory consumption of the aggregate programs and to verify that the programs can be run without runtime problems.

Chapter 8

Smart Warehouse Case Study

8.1 Smart Warehouse

To validate the viability of running aggregate programs on the DWM1001 module we developed a use case of smart warehouse management proposed by Reply and published in [34]. In this case study[1] we consider warehouses organized in a series of aisles on which pallets of goods are stored. The warehouse workers move around the aisles with forklifts, which are usually able to reach a speed of 10 km/h. The workers perform loading and unloaded of pallets from a common loading zone to the correct aisles. It is not uncommon for workers to run into each other while operating the forklifts, damaging the goods and slowing down the operations.

The aim of the case study is to propose an IoT system to improve the safety and efficiency of warehouses by deploying DWM1001 devices running aggregate programs. We assume that the case study warehouse is managed with a high turnover, so that goods are placed as close as possible to the relevant point of operation for them, without a fixed placement based on the good type. High turnover allows for greater efficiency in principle, but it also suffers from performance degradation as the warehouse starts to fill up: workers may need to perform long searches for a required good, or even to find an empty space for a new pallet. We also assumed that each pallet contains a single type of goods.

8.2 Goals

We developed a warehouse management aggregate application that aims at performing the following services:

[1] The source code for the case study is publicly available at: `https://github.com/fcpp-experiments/warehouse-case-study`

- *prevent accidental collision*, by warning workers whenever another forklift is approaching with a speed greater than a threshold, within a given safety radius;
- *provide route information* towards either empty spaces or goods matching a given query to the interested warehouse worker;
- *collect logs* of relevant events (loading/unloading of goods and collision warnings) towards central points that are connected to the cloud.

We assumed that every pallet and every forklift has an associated DWM1001 module. Pallet modules are only able to present output in the form of small LED lights to provide routing information to the operators. Forklift modules can be connected to a simple application on the workers' personal smartphone, which allows the worker to provide some basic input (i.e. starting loading operation and routing requests) and visualize basic outputs (i.e. collision warnings and routing information). The devices mounted on the forklift are called wearables in the source code.

8.3 Implementation

The *prevent accidental collision* service is implemented by the `collision_detection` aggregate function visible in Figure 8.1. The aggregate function is configurable based on the desired detection radius and a threshold on how fast the distance between devices can decrease. The service spawns an aggregate process for each forklit that determines the speed at which the nearest forklift is coming closer. The process uses the `distance_waypoint` aggregate function, visible in figure 8.2 to compute the distance to the closest wearable devices, even when direct communication with it is not possible, and compares it with the distance of the previous round. If this computed speed is higher than the threshold it generates a log that signals a risk of collision, on which the device running the aggregate function can then act upon to warn the user. When the speed become again lower than the threshold it generates a new log to signal that there is no longer the risk.

The `find_space` and `find_goods` aggregate functions, visible in Figure 8.3 implement the *provide route information* service.

In the `find_space` function each pallet determines if there is an empty space around it by counting the number of neighbours within the `grid_step` distance, i.e. a parameter that tells the function how close are arranged the pallets on the shelves. Then using the `distance_waypoint` function the nodes compute the closest neighbour leading to the closest pallet with an empty space available nearby. In this

```
FUN std::vector<log_type> collision_detection(ARGS, real_t radius,
        real_t threshold, times_t current_clock, real_t comm) { CODE
    bool wearable =
        node.storage(tags::node_type{}) == warehouse_device_type::Wearable;
    std::unordered_map<device_t, real_t> logmap = spawn(CALL,
        [&](device_t source){
            auto t = distance_waypoint(CALL, node.uid == source, 0.1*comm);
            real_t dist = self(CALL, get<0>(t));
            real_t closest_wearable = nbr(CALL, INF, [&](field<real_t> x){
                return min_hood(CALL, mux(get<0>(t) > dist, x, INF),
                    wearable and node.uid != source ? dist : INF);
        });
        real_t v = 0;
        if (isfinite(closest_wearable))
            v = (old(CALL, closest_wearable) - closest_wearable) /
                    (node.current_time() - node.previous_time());
        return make_tuple(dist < radius ? v : -INF, dist < radius);
    }, wearable ? common::option<device_t>{node.uid}
            : common::option<device_t>{});
    std::vector<log_type> logvec;
    real_t vn = max(logmap[node.uid], real_t(0));
    real_t vo = old(CALL, vn);
    if (vn > threshold and vo <= threshold)
        logvec.emplace_back(LOG_TYPE_COLLISION_RISK_START, node.uid,
            discretizer(current_clock), vn);
    if (vo > threshold and vn <= threshold)
        logvec.emplace_back(LOG_TYPE_COLLISION_RISK_END, node.uid,
            discretizer(current_clock), vn);
    return logvec;
}
```

Figure 8.1: Implementation of the collision prevention service.

case study we considered for simplicity only pallets with less than two neighbours as pallets with a free space in order to account for pallets located in the corners of the shelves. For practical applications this aggregate function could be improved by triangulating the relative position of the neightbours.

The `find_goods` function takes as input the type of good that the device is looking for, which will be always the constant `NO_GOODS` for pallets or a specific good type for wearable when they are looking to collect that type of good from the warehouse. An aggregate process is spawned for each wearable looking for a good which uses `distance_waypoint` to find the closest pallet on the path to the good.

The output of both functions is then used by the main application function `warehouse_app`, visible in Figure 8.5, to turn on the LED of the correct pallets,

```
FUN tuple<field<real_t>, device_t> distance_waypoint(ARGS,
    bool source, real_t distortion) { CODE
        return nbr(CALL, INF, [&] (field<real_t> d) {
            real_t dist;
            dist = min_hood(CALL,
                d + node.nbr_dist(), source ? -distortion : INF);
            dist += distortion;
            mod_self(CALL, d) = dist;
            device_t waypoint = get<1>(min_hood(CALL,
                make_tuple(d, node.nbr_uid())));
            return make_tuple(make_tuple(d, waypoint), dist);
        });
}
```

Figure 8.2: Implementation of the utility function `distance_waypoint`, which computes the distance of every neighbour from a source, and the best waypoint towards it.

showing the path to the operator.

The *collect logs* service is implemented by the `single_log_collection` aggregate function visible in Figure 8.4. Forklift modules are used as collection sinks, since they can upload data on the cloud, and are split into two groups that simultaneously collect all the logs to enable redundancy and greatly reducing the chances of information losses. Based on their unique identifier, half of the forklifts are assigned to sink group 1 and the other half to sink group 2. Firstly, hopcount distances towards sinks are produced. Then, every device computes its partial log collection, by including every log that appears farther than it from the sink, but not also closer than it from the sink. This second condition ensures that logs stop being propagated when they are already closer to the sink, greatly reducing the communication load.

Finally the `warehouse_app`, visible in Figure 8.5, aggregate function combines all the services, saves in the device storage the information needed to interact with the user and computes statistics over the computation.

8.4 Simulation

To validate the implementation of the services, we simulated a smart warehouse empowered by the proposed app through the FCPP simulation framework for aggregate computing. The simulated warehouse consists of 22 rows x 3 columns of aisles. In each aisle there are 15 slots in the horizontal direction x 3 slots in the vertical direction in which the pallets can be placed.

```
FUN device_t find_space(ARGS, real_t grid_step, real_t comm) { CODE
    bool is_pallet = node.storage(tags::node_type{}) ==
        warehouse_device_type::Pallet and
        node.storage(tags::loaded_goods{}) != no_content and
        node.storage(tags::pallet_handled{}) == false;
    int pallet_count = fold_hood(CALL, [&](tuple<real_t, uint8_t> t, int c){
        return c + (get<0>(t) < 1.2 * grid_step and get<1>(t));
    }, make_tuple(node.nbr_dist(), nbr(CALL, uint8_t{is_pallet})), 0);
    bool source = is_pallet and pallet_count < 2;
    auto t = distance_waypoint(CALL, source, 0.1*comm);
    return get<1>(t);
}
FUN device_t find_goods(ARGS, query_type query, real_t comm) { CODE
    using key_type = tuple<device_t,query_type>;
    std::unordered_map<key_type, device_t> resmap =
        spawn(CALL, [&](key_type const& key){
            bool found = match(get<1>(key), node.storage(tags::loaded_goods{}))
                and node.storage(tags::pallet_handled{}) == false;
            auto t = distance_waypoint(CALL, found, 0.1*comm);
            device_t waypoint = get<1>(t);
            return make_tuple(waypoint, get<0>(key) != node.uid ?
                status::internal : query == no_query ?
                status::terminated : status::internal_output);
    }, query == no_query ? common::option<key_type>{}
        : common::option<key_type>{node.uid,query});
    return resmap.empty() ? node.uid : resmap.begin()->second;
}
```

Figure 8.3: Implementation of the route information service.

A screenshot of the simulation is shown in the Figure 8.6. In the bottom of the warehouse there are two special zone: the first one on the left is a loading zone which contains the pallets that the operators will move in the aisles and in which the operators will unload the pallets taken from the aisles; the second one, marked as black in the Figure 8.6, is an office space not accessible by the operators. In the simulation pallets are represented as cubes with a 1m side, while the slots in the aisles are of 1.5m each side, leaving 0.5m space between adjacent pallets. The content of a pallet is displayed by the colour of its middle band with white representing no content. The pallets are pre-loaded with 100 different types of goods, randomly generated accortding to a Zipf distribution [37] so that few goods are very common and many are uncommon. Forklifts are represented as spheres, with the middle band coloured according to the good that the forklift is currently searching or loading on an empty pallet. A black middle band represents that the forklift is currently neither searching or loading any specific good type, e.g. while

```
FUN std::vector<log_type> single_log_collection(ARGS,
        std::vector<log_type> const& new_logs, int parity) { CODE
    bool source = node.uid % 2 == parity and
        node.storage(tags::node_type{}) == warehouse_device_type::Wearable;
    field<uint8_t> nbrdist = nbr(CALL, std::numeric_limits<uint8_t>::max(),
        [&](field<uint8_t> d){
            uint8_t nd = min_hood(CALL, d,
                std::numeric_limits<uint8_t>::max());
            nd = source ? 0 : nd + 1;
            mod_self(CALL, d) = nd;
            return make_tuple(std::move(d), nd);
    });
    uint8_t dist = self(CALL, nbrdist);
    std::vector<log_type> r = nbr(CALL, std::vector<log_type>{},
        [&](field<std::vector<log_type>> nl){
            std::vector<log_type> uplogs   = sum_hood(CALL,
                mux(nbrdist > dist, nl, std::vector<log_type>{}));
            std::vector<log_type> downlogs = sum_hood(CALL,
                mux(nbrdist < dist, nl, std::vector<log_type>{}));
            return (uplogs - downlogs) + new_logs;
    });
    return source ? r : std::vector<log_type>{};
}
FUN std::vector<log_type> log_collection(ARGS,
        std::vector<log_type> const& new_logs) { CODE
    std::vector<log_type> r0 = single_log_collection(CALL, new_logs, 0);
    std::vector<log_type> r1 = single_log_collection(CALL, new_logs, 1);
    return r0.empty() ? r1 : r0;
}
```

Figure 8.4: Implementation of the log collection service.

it is idle or while placing a pallet in the aisles after loading its content. For pallets the lateral bands are: yellow if it has a LED turned on, meaning that is part of a route; red during handling, meaning that is being carried by a forklift; and gray if the device is idling. For forklifts the lateral bands are: yellow if it has a LED turned on, meaning that there is a collision risk; red while changing the content of a nearby paller during a loading or unloading operation; gray in all other cases, meaning that is either idle or moving to a location to perform an operation of insert or retrieve of a pallet.

The forklift randomly perform a task among:

- *retrive a pallet containing a specific good*: starting from the loading zone the forklift picks up a matching pallet following the path provided by the routing algorithm, then it brings it back to the loading zone and marks the pallet as

```
FUN device_t warehouse_app(ARGS, real_t grid_step, real_t comm_rad,
        real_t safety_radius, real_t safe_speed) { CODE
    bool is_pallet = node.storage(tags::node_type{}) ==
        warehouse_device_type::Pallet;
    times_t current_clock = shared_clock(CALL);
    node.storage(tags::global_clock{}) = current_clock;
    std::vector<log_type>& logs = node.storage(tags::new_logs{});
    logs = {};
    logs = logs + load_goods_on_pallet(CALL, current_clock);
    logs = logs + collision_detection(CALL,
        safety_radius, safe_speed, current_clock, comm_rad);
    node.storage(tags::coll_logs{}) = log_collection(CALL, logs);
    device_t space_waypoint = find_space(CALL, grid_step, comm_rad);
    device_t goods_waypoint = find_goods(CALL,
        node.storage(tags::querying{}), comm_rad);
    device_t waypoint = is_pallet ? node.uid :
        node.storage(tags::querying{}) == no_query ?
            space_waypoint : goods_waypoint;
    node.storage(tags::led_on{}) = any_hood(CALL,
        nbr(CALL, (real_t)waypoint) == node.uid, false);
    statistics(CALL, current_clock);
    return waypoint;
}
```

Figure 8.5: The main aggregate function that implements the uses all the services.

empty.

- *insert a pallet with a specific good*: starting from the loading zone the forklift fills an empty pallet with the choosen good and picks it up, then it brings it to an empty space in the aisles following the path provided by the routing algorithm.

The tasks, and the associated goods types, are generated randomly for each forklift while is idle.

We run the simulation with an initial setup of 6 forklifts, 10 empty pallets and 500 pallets in the aisles. The simulated warehouse size is of 89 meters x 95 meters with an height of 10 meters. The maximum range of communication was set to 25 meters for the forklifts and 9 meters for the pallets, with a probabilistic message loss of 50% of messages lost at 80% of range radius.

In Figure 8.6, we can see few empty pallets in the loading zone (gray and white cubes), and several hundreds of loaded pallets in the aisles (coloured cubes in the grid). In the loading zone, two forklifts are currently idling (gray and black spheres), while one is currently unloading a pallet (bottom left corner, with the central color white meaning that is setting the pallet to having no goods loaded and

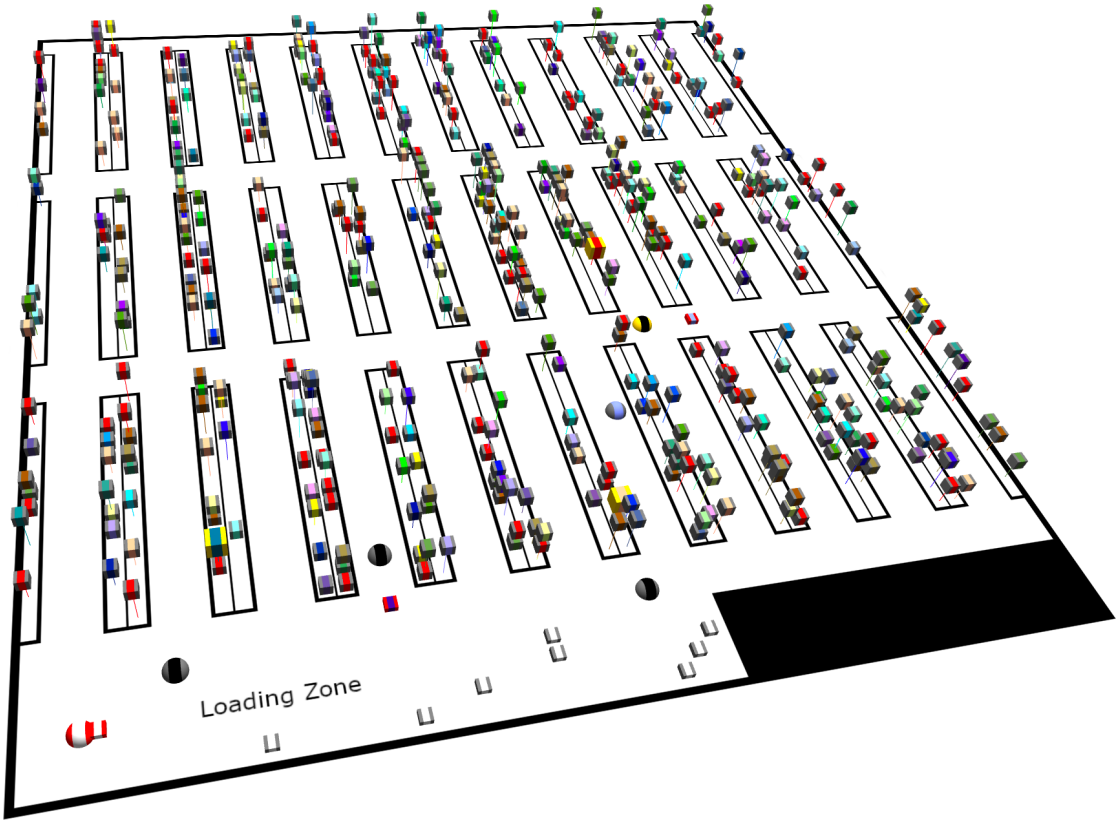

Figure 8.6: Screenshot of the simulation execution.

red bands meaning that is activly setting the content of a pallet in that moment). A forklift (bottom of the 4th vertical aisle, gray and black sphere followed by a red and violet cube) has recently loaded a pallet, and is bringing it to the closest available space. The pallet has red bands, meaning that it is being handled by a forklift, and a violet middle band representing the good type that has already been loaded on it. The forklift has gray bands, meaning that is not currently changing the content of any pallet, and a black color representing that is not currently searching for any good. Another forklift (bottom of the 7th vertical aisle, gray and cerulean sphere) is currently looking for a specific good (identified by the cerulean colour), following LED lights (currently, the yellow and red cube further up in the same aisle). The last forklift (middle-bottom horizontal corridor, yellow and black sphere followed by a red and cerulean cube) is bringing back a pallet to the loading zone for unloading. Since these last two forklift are quickly approaching the same intersection, a collision warning is triggered (the external yellow bands of the last forklift).

We collected the data visible in Figure 8.7 over the course of 500 seconds of

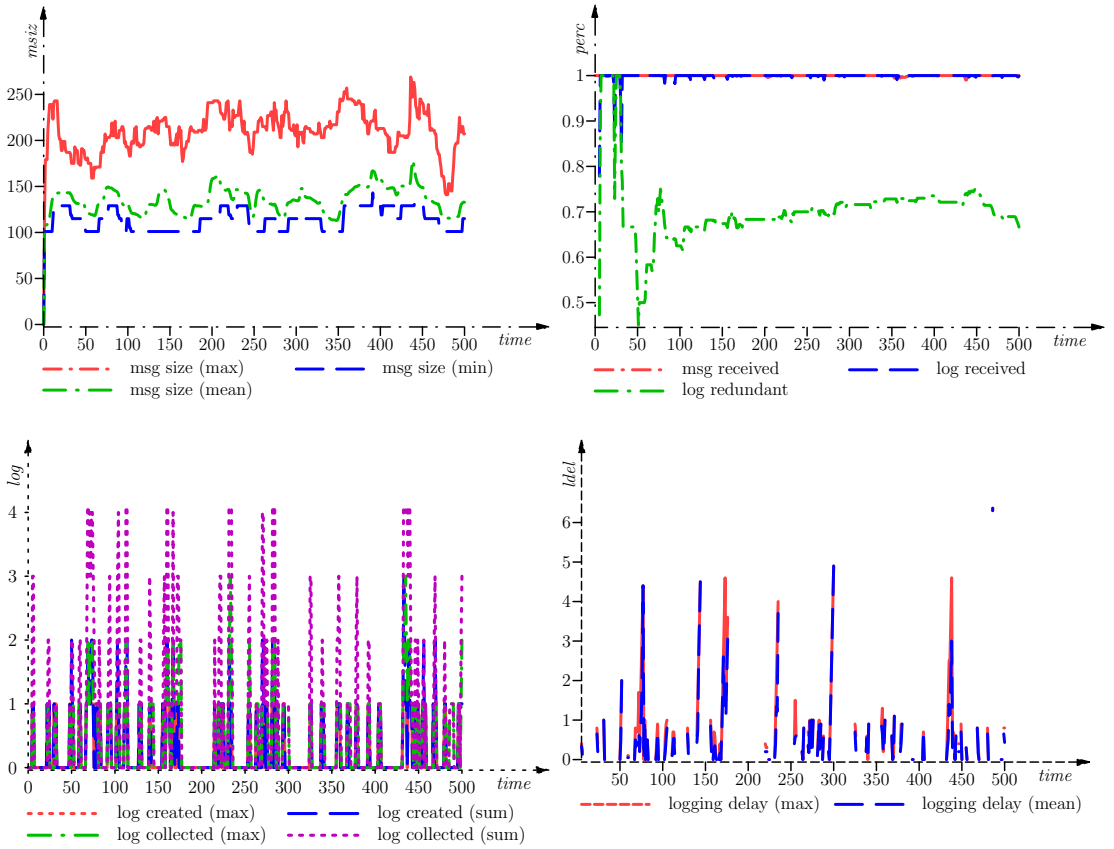

Figure 8.7: Plots of simulated performance over time: message size in bytes (top-left); percentage of messages delivered, log received at least once and received twice (top-right); logs created and collected (bottom-left), average delay of log collection (bottom-right).

simulated time. The size of messages is usually below 150 bytes, with peaks below 250 bytes. In the messages 20 bytes were dedicated to the simulation logics, which means that almost every message is small enough to be sent with the BLE driver. Every created log is eventually received, but only 70% of them are received by both sink group, while another 30% is only received in one sink group, proving the effectiveness of the redundancy strategy. Across the simulation, at most 2 logs are created simultaneously (and never more than one in a single device), while a single sink may collect up to 3 logs in a single round (with 4 total logs collected in the same round by sinks overall). The average delay between the log creation and collection is very small, being mostly below 5 seconds (missing parts in the bottom-right graph correspond to times when no log was collected, so that no delay was computable).

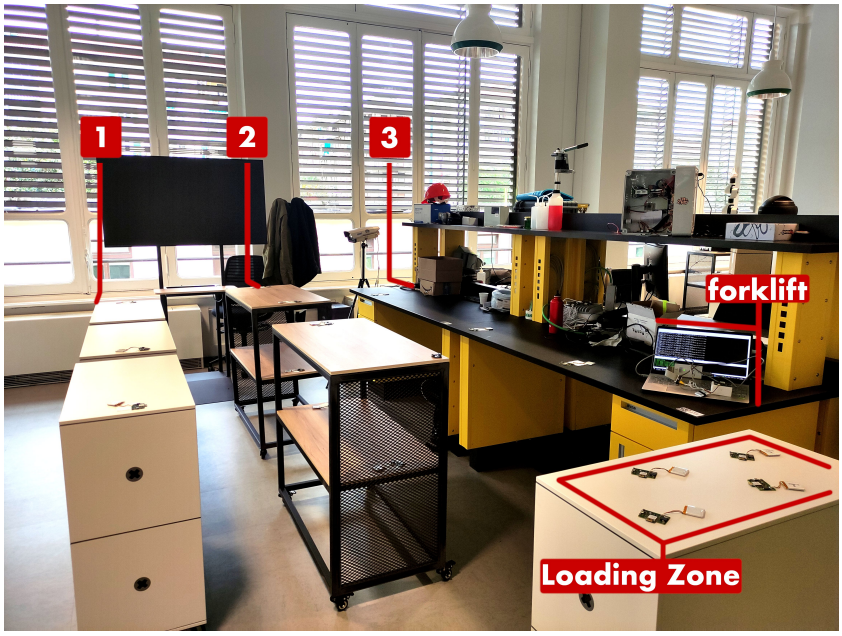

Figure 8.8: Layout of the devices in the Proof-of-Concept deployment.

8.5 Proof-of-concept Experiment

To validate the results obtained in the simulation we run the same program on a batch of twenty DWM1001 modules, in particular using the Reply's custom board. To support the custom board we implemented new APIs to interact with the button, the buzzer and the flash memory of the module.

The devices were physically deployed in the Reply's laboratory as visible in Figure 8.8. One module was configured as the personal device of a forklift operator, and kept connected to a PC to collect the logs in real time (Figure 8.8 right). Four other modules were configured as being attached to empty pallets in the *loading zone*, realised by a separate cabinet (Figure 8.8 bottom right). The remaining 15 modules were configured as being attached to pallets having three different kinds of goods already loaded (of different abundance). These modules were arranged in a grid-like configuration with a step of about $1m$ to mimic warehouse aisles, consisting of three parallel rows and three different levels of elevation (Figure 8.8, aisles numbered from 1 to 3).

We configured FCPP to use the BLE driver with a transmission power of -12dBm to reduce the range of communication so that not all devices were able to communicate with every module. The messages where broadcasted three times each round at interval of 10 milliseconds to account for collisions. We also configured to 2 the maximum number of ranging request in each round by each device

to limit collisions and battery usage. We stored in the flash memory the state of the computation at the end of each round, so that in case of restarts of a device the computation can keep going on.

We tested a scenario in which a single operator searches for a pallet containing a randomized kind of good, followed by its unloading near the empty pallets. In this scenario the operator interacts with the modules using the button available on them, while the modules provide feedbacks using the vibrations from the buzzer. The operator starts the request to find a kind of good by pressing the button on the personal module, then follows the vibration of the loaded pallets until reaching one containing the requested goods. At this point, the module on that pallet starts to vibrate intermittently and the operator delivers it back to the starting area and marks the pallet as empty by pressing the button on the pallet's module. This operation was repeated three times without resetting the devices.

At the end of the test the FCPP logs from the personal module of the operator were collected on the PC to verify that the network presented the expected behavior. Thanks to the resilience of the AP paradigm, the network was left in a consistent state, with all operations consistently performed, in spite of a quite high rate of message delivery failure, and devices occasionally restarting due to memory or other issues.

Chapter 9

Other Case Studies

To validate the advantages of the Aggregate Programming paradigm we proposed a few more case studies, published in [13, 14], involving safety and monitoring of smart buildings. Those case studies have not yet been implemented but will be subject of future work.

In particular we proposed the following case studies:

- Office building application for the Department of Architecture, Built Environment and Construction Engineering (DABC) building of Politecnico di Milano [13]. This is a four-story building of about 4300 square meters of gross floor area, hosting administrative and university staff offices, research laboratories, and meeting rooms. This building already hosts an IoT network with camera sensors to monitor real-time occupancy and user flows by linking virtual agents to the users, visible in figure 9.1. We proposed an AP system to optimize the space management in real-time and to recognize dynamically the best emergency egress paths in case of emergencies.

- School building application for the school of Melzo [13]. This case study was first analyzed in a research project started during the first months of the Covid-19 pandemic, with the main goal of ensuring safe school reopenings, visible in figure 9.2. The existing research project developed an online tool called "Spazio alla Scuola" aimed at supporting school managers in defining maximum room capacities and time needed to safely entering and exiting school buildings. We proposed to apply the AP technology to enable real-time monitoring of the respect of social distancing measures and to enable detection of safe escape routes in case of hazardous events.

- Construction site application for an on-going project of demolition and construction of a school complex located in Inveruno [13], visible in figure 9.3. We proposed to apply an AP system to mitigate the risks on the construction

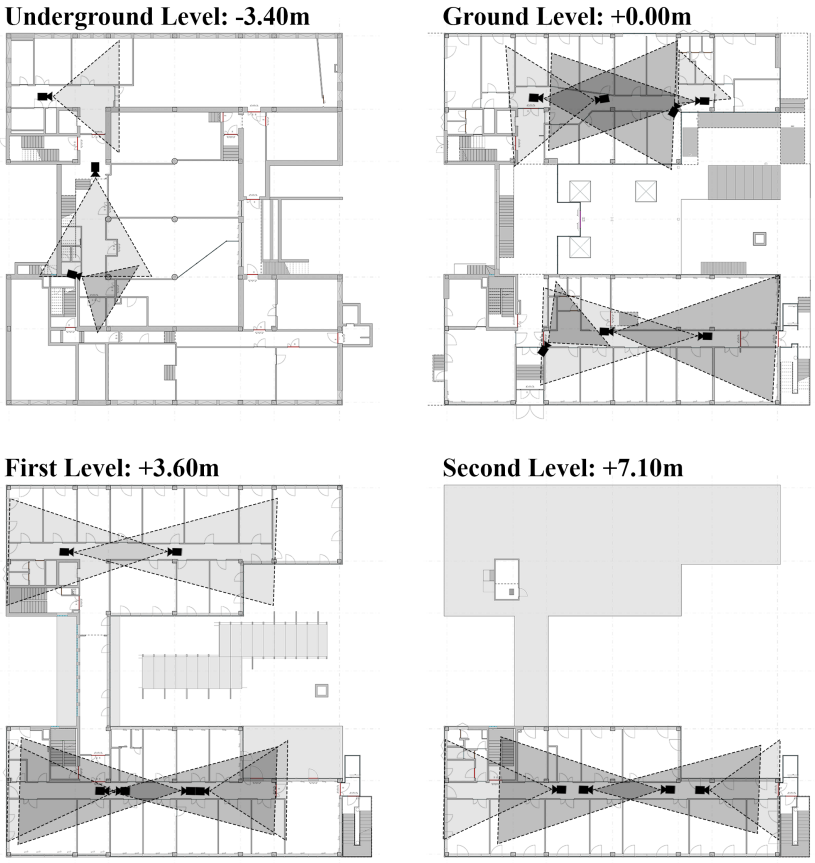

Figure 9.1: Spatial distribution of existing camera sensors in the ABC Department of Politecnico di Milano. From [13].

site, like preventing access to areas during dangerous activities or preventing being stuck by a moving vehicle.

Futher work needs to be done in the future to simulate and implement the proposed case studies.

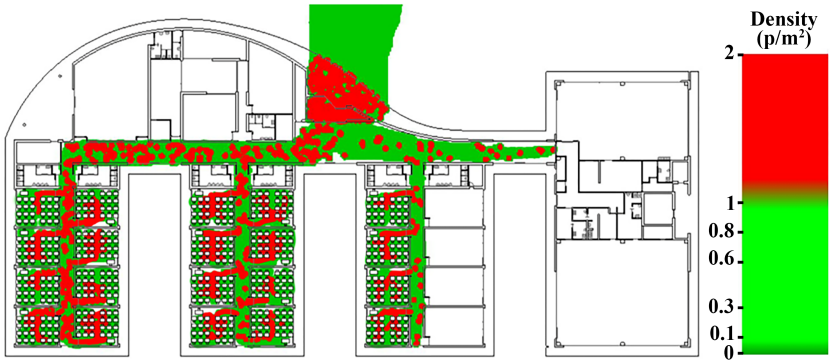

Figure 9.2: Density map resulting from crowd simulations on the primary school of Melzo. From [13].

Figure 9.3: Construction site layout of the primary and secondary school of Inveruno. From [13].

Chapter 10

Ongoing Development

With the porting of FCPP on the DWM1001 module we achieved the first deployment of an aggregate program on IoT devices, which opened to road to further research and case studies currently under development. The company where I work, Santer Reply, is currently working with Synesthesia and the department of Computer Science of the University of Turin to develop new case studies for the Aggregate Programming involving fleets of autonomous robots or drones. The team involves several people from both companies and the University. In the project named New Generation ROBOtics by cooperative fleet (RoboNG) the goal is to guide the operations of robots, equipped with sensors and machine vision systems, using aggregate programs to achieve an higher level of coordination, reliability and resilience.

The FCPP program platform has been integrated with the ROS 2 [29] (Robot Operating System 2) software platform. ROS 2 is an open source middleware to develop applications for robot systems, which provides API in the C++, Python and Java programming languages. As shown in figure 10.1, ROS 2 provides an architecture based on *nodes* that communicate to perform actions. The nodes communicate using (i) publish subscribe mechanisms on topics, (ii) direct requests with direct responses or (iii) actions, i.e. long-running procedures with continous feedback and the ability to cancel the goal. ROS 2 provides also ready to use algorithms and development tools for simulation, visualization and debugging.

FCPP has been made able to interact with this architecture through the file system. It creates goal files and feedback files from the topics and made them available to the AP FCPP engine. With the files the engine updates the internal device data in the aggregate program, then it emits an actions file that will be converted in commands to send to the robots using the ROS 2 APIs.

Currently the team is working to apply this architecture to applications on the following robots:

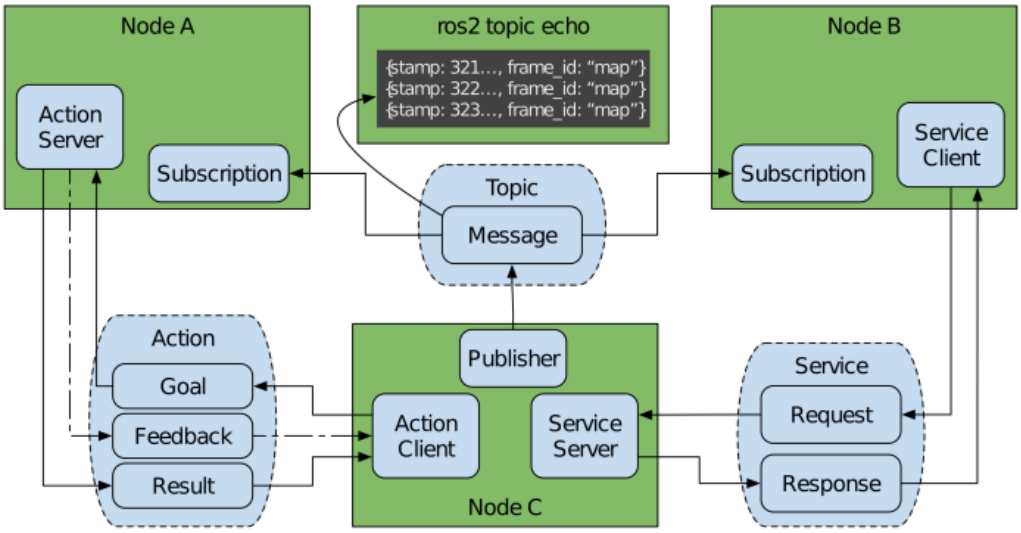

Figure 10.1: ROS 2 node interfaces: topics, services, and actions. From [29].

- Clearpath Robotics Jackal, visible in figure 10.2, is a small robot (508 x 430 x 250 mm) equipped with an on-board computer, GPS, ROS 2 support and both Bluetooth and Wi-Fi connectivity.
- Create 3, visible in figure 10.3, is based on the Roomba design and equipped with sensors and user buttons. The software is based on ROS 2.

The team will soon start to work also with the drone Crazyflie, visible in figure 10.4. Crazyflie is a lightweight flying drone with BLE and accelerometer/gyroscope sensors. The team is also working on equipping the robots with LiDAR sensors and cameras to perform image detection (using AI techniques) and obstacles detection.

In particular we are starting to work on a case study in collaboration with the airport of Turin. In this case study we will deploy Jackals to detect Foreign Object Debris (FOD) on the pavement of the airport to guarantee the safety of the environment. The Jackal robots will coordinate to explore all the area and analize in real time the data from the cameras and LiDAR. An operator will then be able to see the collected data on a laptop in real time and verify the reports. Thanks to the Aggregate Programming the system will be able to adapt to different number of robots, starting the operations from not pre-established points and handle problems in the robot operations.

Figure 10.2: Jackal by Clearpath Robotics. From `https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle` .

Figure 10.3: Create 3 by iRobot Education. From `https://iroboteducation.github.io/create3_docs/hw/overview` .

Figure 10.4: Crazyflie by Bitcraze. From `https://www.bitcraze.io/products/crazyflie-2-1` .

Chapter 11

Conclusions and Perspectives

11.1 Results obtained

With the implementation of the support for FCPP on the DWM1001 module we enabled aggregate programs to run on this IoT module for the first time. This porting also validated the FCPP architecture, proving that its modularity allows to support new platforms even when they require massive customizations. Thanks to the changes developed for FCPP in order to support the DWM1001 module we increased the configurability of the software, which will make easier to support other hardwares in the future by developing new drivers. Future integrations of FCPP with other OSes different from Contiki will also be able to use this porting as reference on how to integrate the aggregate program execution with the execution model of the OS. Implementing the case study proved both that the Aggregate Programming is a viable approach to implement industrial distributed applications and that aggregate programs are enough lightweight to run on low resources hardware with small sized messages. The current proof of concept implementation of the case study also paved the way for the new case studies under development with robots and drones described in chapter 10.

11.2 Future Work

In the future, the DWM1001 support needs to be further stabilized, by preventing the occasional crashes and reducing the memory footprint. Better syncronizating algorithms need to be implemented in order to reduce the ranging and BLE collisions, so that less messages are lost, and to reduce energy consumption. The case study can be further improved by implementing a cooperative RTLS (real time location system) in FCPP, so that each device can determine its position in the

warehouse to improve the services. Finally larger scale experiments need to be performed with the devices to evaluate the performance of the system in a real world scale. The case studies described in chapter 9 need to be further developed and implemented. The case study with the airport of Turin needs to be fully implemented and more use cases need to be developed for the available robots and drones.

References

[1] DWM1000 - Qorvo. https://www.qorvo.com/products/p/DWM1000. Accessed: 2024-04-29.

[2] DWM1001 Data Sheet. https://www.qorvo.com/products/d/da007950. Accessed: 2024-04-29.

[3] DWM1001C - Qorvo. https://www.qorvo.com/products/p/DWM1001C. Accessed: 2024-04-29.

[4] GNU ARM Embedded Toolchain. https://developer.arm.com/downloads/-/gnu-rm. Accessed: 2024-04-29.

[5] J-Link RTT – Real Time Transfer. https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/. Accessed: 2024-04-29.

[6] nRF52832 - Versatile Bluetooth 5.2 SoC - nordicsemi.com. https://www.nordicsemi.com/Products/nRF52832. Accessed: 2024-04-29.

[7] IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs). *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, pages 1–314, 2011.

[8] G. Audrito. FCPP: an efficient and extensible field calculus framework. In *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 153–159. IEEE, 2020.

[9] G. Audrito, J. Beal, F. Damiani, D. Pianini, and M. Viroli. Field-based coordination with the share operator. *Logical Methods in Computer Science*, 16, 2020.

[10] G. Audrito, J. Beal, F. Damiani, and M. Viroli. *Space-Time Universality of Field Calculus*, pages 1–20. 01 2018.

[11] G. Audrito, R. Casadei, F. Damiani, G. Salvaneschi, and M. Viroli. The eXchange Calculus (XC): A functional programming language design for distributed collective systems. *Journal of Systems and Software*, 210:111976, 2024.

[12] G. Audrito, R. Casadei, F. Damiani, and M. Viroli. Computation Against a Neighbour: Addressing Large-Scale Distribution and Adaptivity with Functional Programming and Scala. *Logical Methods in Computer Science*, Volume 19, Issue 1, Jan. 2023.

[13] G. Audrito, F. Damiani, G. Di Giuda, S. Meschini, L. Pellegrini, E. Seghezzi, L. C. Tagliabue, L. Testa, and G. Torta. RM for users' safety and security in the built environment. In *Proceedings of the 5th ACM International Workshop on Verification and MOnitoring at Runtime EXecution*, VORTEX 2021, page 13–16, New York, NY, USA, 2021. Association for Computing Machinery.

[14] G. Audrito, F. Damiani, S. Rinaldi, L. C. Tagliabue, L. Testa, and G. Torta. *Aggregate Programming for Customized Building Management and Users Preference Implementation*, pages 147–172. Springer International Publishing, Cham, 2023.

[15] G. Audrito, F. Damiani, M. Viroli, and E. Bini. Distributed real-time shortest-paths computations with the field calculus. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 23–34. IEEE, 2018.

[16] G. Audrito, L. Rapetta, and G. Torta. Extensible 3d simulation of aggregated systems with fcpp. In M. H. ter Beek and M. Sirjani, editors, *Coordination Models and Languages*, pages 55–71, Cham, 2022. Springer Nature Switzerland.

[17] G. Audrito and G. Torta. FCPP to aggregate them all. *Science of Computer Programming*, 231:103026, 2024.

[18] G. Audrito, M. Viroli, F. Damiani, D. Pianini, and J. Beal. A higher-order calculus of computational fields. *ACM Trans. Comput. Logic*, 20(1), jan 2019.

[19] J. Beal, D. Pianini, and M. Viroli. Aggregate programming for the internet of things. *Computer*, 48:22–30, 09 2015.

[20] H. I. Cannon. *Flavors: A Non-hierarchial Approach to Object-oriented Programming*. Institute of Technology, 1981.

[21] R. Casadei and M. Viroli. Towards Aggregate Programming in Scala. In *First Workshop on Programming Models and Languages for Distributed Computing*, PMLDC '16, New York, NY, USA, 2016. Association for Computing Machinery.

[22] R. Casadei, M. Viroli, G. Audrito, D. Pianini, and F. Damiani. Aggregate processes in field calculus. In *International Conference on Coordination Languages and Models*, pages 200–217. Springer, 2019.

[23] P. Corbalán, T. Istomin, and G. P. Picco. Poster: Enabling Contiki on Ultra-wideband Radios. In *Proceedings of the International Conference on Embedded Wireless Systems and Networks*, EWSN'18, 2018.

[24] A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*, pages 455–462. IEEE, 2004.

[25] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, page 29–42, New York, NY, USA, 2006. Association for Computing Machinery.

[26] T. Istomin, E. Leoni, D. Molteni, A. L. Murphy, G. P. Picco, and M. Griva. Janus: Efficient and accurate dual-radio social contact detection, 2021.

[27] T. Istomin, E. Leoni, D. Molteni, A. L. Murphy, G. P. Picco, and M. Griva. Janus: Dual-radio accurate and energy-efficient proximity detection. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 5(4), dec 2022.

[28] W. Z. Khan, M. Rehman, H. M. Zangoti, M. K. Afzal, N. Armi, and K. Salah. Industrial Internet of Things: Recent advances, enabling technologies and open challenges. *Computers & Electrical Engineering*, 81:106522, 2020.

[29] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66), 2022.

[30] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4), jul 2009.

[31] G. Oikonomou, S. Duquennoy, A. Elsts, J. Eriksson, Y. Tanaka, and N. Tsiftes. The Contiki-NG open source operating system for next generation IoT devices. *SoftwareX*, 18:101089, 2022.

[32] D. Pianini, S. Montagna, and M. Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7, 01 2013.

[33] D. Pianini, M. Viroli, and J. Beal. Protelis: Practical aggregate programming. *Proceedings of the ACM Symposium on Applied Computing*, pages 1846–1853, 01 2015.

[34] L. Testa, G. Audrito, F. Damiani, and G. Torta. Aggregate processes as distributed adaptive services for the industrial internet of things. *Pervasive and Mobile Computing*, 85:101658, 2022.

[35] M. Viroli, G. Audrito, J. Beal, F. Damiani, and D. Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Trans. Model. Comput. Simul.*, 28(2), mar 2018.

[36] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini. From distributed coordination to field calculus and aggregate computing. *Journal of Logical and Algebraic Methods in Programming*, 109:100486, 2019.

[37] G. K. Zipf. *Human behavior and the principle of least effort: An introduction to human ecology.* Addison-Wesley, 1949.