Aggregate Processes as Distributed Adaptive Services
for the Industrial Internet of Things

Lorenzo Testa[1,2], Giorgio Audrito[1], Ferruccio Damiani[1], and Gianluca Torta[1]

[1]Dipartimento di Informatica, University of Turin, Turin, Italy
[2]Concept Reply, Turin, Italy

April 4, 2022

Abstract

The Industrial Internet of Things (IIoT) promises to bring many benefits, including increased productivity, reduced costs, and increased safety to new generation manufacturing plants. The main ingredients of IIoT are the connected, communicating devices directly located in the workshop floor (far edge devices), as well as edge gateways that connect such devices to the Internet and, in particular, to cloud servers. The field of Edge Computing advocates that keeping computations as close as possible to the sources of data can be an effective means of reducing latency, preserving privacy, and improve the overall efficiency of the system, although building systems where (far) edge and cloud nodes cooperate is quite challenging. In the present work we propose the adoption of the Aggregate Programming (AP) paradigm (and, in particular, the "aggregate process" construct) as a way to simplify building distributed, intelligent services at the far edge of an IIoT architecture. We demonstrate the feasibility and efficacy of the approach with simulated experiments on FCPP (a C++ library for AP), and with some basic experiments on physical IIoT boards running an ad-hoc porting of FCPP.

1 Introduction

The Industrial Internet of Things (IIoT), namely the concepts and technologies of IoT applied to (smart) industry, is gaining increasing interest as it promises to improve productivity and safety in the workplace through the collection, analysis and exploitation of large amounts of data from the workshop floor. While there is no single definition of IIoT, the following one (taken from [23]) can be a useful reference for our purposes: "IIoT is the network of intelligent and highly
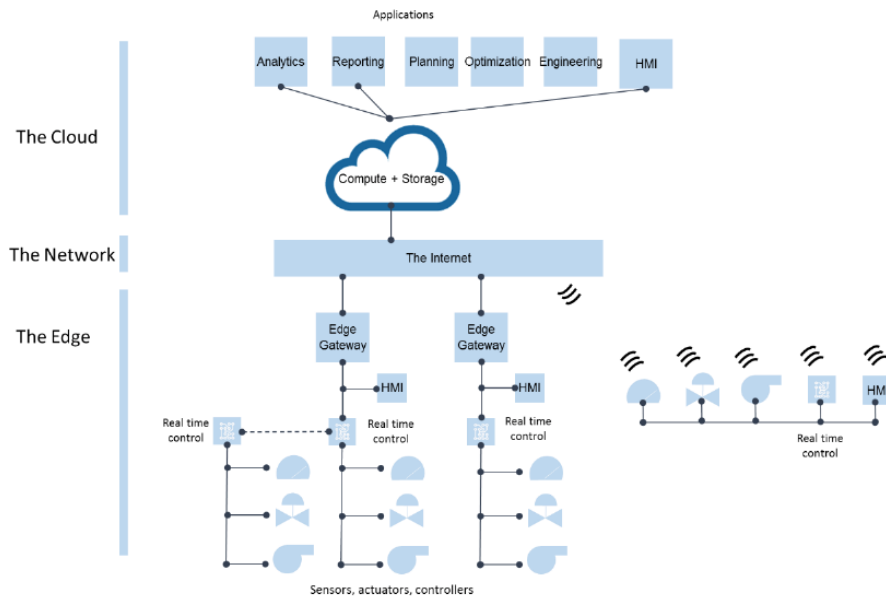
Figure 1: Reference Architecture of the IIoT (from `https://en.wikipedia.org/wiki/Industrial_internet_of_things`).

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."

From this definition we can deduce that connectivity and data collection are the main enabling elements of the IIoT. Furthermore, the reference architectures of the IIoT also stress the need of accessing high performance computational and storage nodes at a higher level, usually in the cloud. Again, we find several slightly different definitions of the IIoT architecture (see, e.g., [23, 35]); all of which, however, share most aspects with one depicted in Figure 1.

We further partition the bottom layer in the figure into two sublayers: the *Far Edge* layer which contains the connected devices in the workshop floor, including: machines, sensors, actuators, real-time controllers, Human-Machine-Interface (HMI) units; and the *Edge gateways* that act as bridges towards the outside world. Then, the information collected by the gateways crosses the *Network* layer to reach the *Cloud* layer, where it is processed by rich applications for analysis, planning, optimization, etc. Of course the flow can also be reversed, so that plans and decisions created at the *Cloud* layer can reach the *Far Edge*.

A well known potential problem with this scheme is that pushing all the high-level functions to the cloud (i.e., far from the workshop floor), can have a negative impact on the reliability, cost, scalability and latency of the system. The field of *edge computing* aims at addressing the problem by moving some
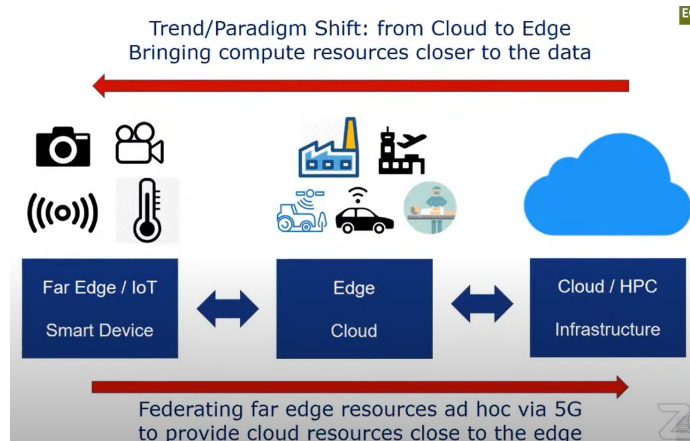
Figure 2: Cloud-Edge-IoT Orchestration.

computations closer to the edge, notably moving (part of) the data processing closer to where the data is produced (see Figure 2, image from the presentation of the Horizon Europe Destination 3: Call 'World leading data and computing technologies 2022'.).[1] The term *edge computing* does not prescribe exactly how close to the *things* (i.e., devices) the computation should be moved, and how much powerful should be the edge computing nodes. Thus, referring to Figure 1, the term can be applied to all the following (and quite diverse) situations: computations running at servers in the *Network* layer that are only a few hops from the *Edge Gateway*s; computations running in the *Edge Gateway*s themselves; and computations running in the *Far Edge* layer, exploiting the constrained memory and computing capabilities of smart devices.

In this paper we focus on the latter meaning of edge computation, whereby offloading some of the computations to the *Far Edge* layer can make them more robust, adaptive and responsive. In particular, we propose an approach that makes it possible to build distributed services for the IIoT running on highly resource-constrained devices. At the core of our approach lies the possibility of running distributed computations – defined as processes in the Aggregate Programming (AP) paradigm [9] and in the foundation of this paradigm provided by the Field Calculus (FC) [5] – on IIoT devices by relying on the FCCP library, a C++ implementation of FC [1]. The AP paradigm is suitable for programming networks of devices forming an open dynamic topology with gossip-based communication. In the IIoT settings, this covers the devices at the *Far Edge* layer including in particular mobile devices, such as wearables carried by workers, or sensors attached to machines and objects that move, such as forklifts and pallets.

---

[1]The presentation (given by Jan Comar at the Horizon Europe Cluster 4 - Digital, Industry & Space Info Day 2021 https://ec.europa.eu/info/research-and-innovation/events/upcoming-events/horizon-europe-info-days/cluster-4_en) can be found at https://www.youtube.com/watch?v=SevWyhwaEwE.

The main contributions of this paper are as follows:

1. we extend the FCPP library with the `spawn` construct enabling *aggregate processes*;

2. we port the extended FCPP library to a resource constrained, IoT physical DecaWave board, equipped with the Contiki NG operating system;

3. we identify several IIoT use cases that can be addressed by applying the AP paradigm to program a network of devices equipped with FCPP;

4. we provide simulations of such IIoT use cases; and

5. we provide preliminary experiments with physical DecaWave boards running FCPP.

The paper is structured as follows. Section 2 reviews related work. Section 3 briefly recalls AP and its FCCP implementation [1], which is a library providing FC as a C++ internal Domain-Specific Language (DSL). Section 4 describes the fundamental step of porting the FCPP library on a DecaWave board, which provides wireless communication on a resource contrained System on Chip (SOC). Section 5 links the powerful constructs of FC to some types of services that can be implemented in an IIoT setting. Section 6 describes an IIoT scenario of a *Warehouse App*, an AP service run on hardware with the physical characteristics of the DecaWave board; then simulates its operation through the FCPP 3D simulator, and presents preliminary experiments with physical DecaWave boards running a port of the FCPP library. Finally, section 7 concludes the paper discussing further research directions.

2 Related work

2.1 Edge Computing in the IIoT

Distributed edge architectures have been explored in the context of Data Management for the IIoT, which involves generating, aggregating, storing, analysing and requesting data. These are the main kinds of tasks we shall discuss in the present paper for AP powered IIoT edge devices.

In [37], the authors review the role of Big Data Analytics (BDA) in the IIoT setting. They recall that BDA involves several sub-tasks, such as data engineering, preparation, and analytics; and, in additiom, the management and automation of the data pipeline. They propose an architectural model called *Concentric Computing*, whereby the elements of a BDA/IIoT system are placed in concentric circles from the external far edge devices, to the outer and inner edge servers, to the cloud services.

According to Concentric Computing, computational and storage resources must be offered by different devices and systems across the complete set of

cirles. The objectives are expressed in terms of storage, in-network data movement, energy consumption, privacy, security and real-time knowledge availability. Therefore, in many cases priority should be given to devices and systems near data sources to ensure real-time or near real-time intelligence near end points, IoT devices and other data sources in IIoT systems

In [34] the authors propose a Data Management Layer (DML) for the IIoT that is separated from the Network Routing. They focus on the lower (*far-edge*) layer of the IIoT architecture, and consider a typical scenario where a set of source nodes produce data, a set of destination nodes require those data, and a maximum latency L_{max} is tolerated by destinations. They borrow the idea of *proxy* from other data distribution contexts (e.g., Content Distribution Networks [15] and Multimedia Streaming [45]), and dynamically compute a caching scheme on proxy nodes that aims at improving latency while keeping the number of proxies (and, thus, the data redundancy costs) as low as possible.

In [33], the authors explicitly consider user User Equipment (UE) that features 5G connectivity as a means to fill the gap between the intelligence embedded in the tools, shop-floors, and conveyor belts and the cloud services able to process such information. The main problem becomes that of associating each IoT device to a UE in a (approximately) optimal way. The authors find that the greedy association scheme, whereby the IoT devices associate with the UEs with probabilities that are proportional to the number of devices that the UEs can support, performs best. An interesting aspect of this study is the variation of several parameters during (simulated) tests, including the number of IoT devices, the number of UEss, the data arrival probability at the IoT devices, and the evolution of uplink data processes of the UEs.

2.2 Programming ensembles of devices spread over space at the far edge

Different development approaches for systems involving a potentially vast number of heterogeneous devices that need to coordinate to perform collective tasks by relying on proximity-based interactions (as in wireless sensor networks) have been proposed in literature. In the following we classify them into five categories, identified by a survey [8].

- *Foundational approaches* propose compact formalizations aimed at modelling the interaction of groups in complex environments. Most of them extend π-calculus [28]. They include, for instance: models of environment structure (from "ambients" to 3D abstractions) [11, 12, 27]; shared-space abstractions allowing multiple processes to interact in a decoupled way [10, 40]; and attribute-based models featuring declarative specification of the target of communication for dynamically creating ensembles [17].

- *Device abstraction languages* are aimed at allowing programmers to focus on cooperation and adaptation, by making the details of device interactions implicit. They include, for instance: TOTA [25], which supports programming tuples with reaction and diffusion rules; the SAPERE

5

approach [43], which supports similar rules embedded in space and apply semantically; the $\sigma\tau$-Linda model [42], which supports manipulation of tuples over space and time; MPI [26], which allows to declaratively expresses topologies of processes in supercomputing applications; NetLogo [36], which provides abstract means to interact with neighbours according to the cellular automata style; and Hood [44], which features implicit sharing of values with neighbours.

- *Pattern languages* provide adaptive means for composing geometric and/or topological constructions, with little focus on computational capability. They include, for instance: the Origami Shape Language [29], which allows to imperatively specify geometric folds that are compiled into processes identifying regions of space; the Growing Point Language [16], which allows to describe topologies in terms of a "botanical" metaphor with growing points and tropisms; ASCAPE [22], which supports agent communication by means of topological abstractions and a rule language; and a catalogue of self-organisation patterns [19], which organises a variety of mechanisms from low-level primitives to complex self-organization patterns.

- *Information movement languages* are the complement of pattern languages. They provide support summarising information obtained from across spacetime regions of the environment and streaming these summaries to other regions, however, they provide little control over the patterning of that computation. They include, for instance: TinyDB [24], which views a wireless sensor network as a database; Regiment [30], which uses a functional language to be compiled into protocols of device-to-device interaction; and KQML [20], an agent communication language.

- *Spatial computing languages* provide flexible mechanisms and abstractions for spatial aspects of computation. They avoid the limiting constraints of the other categories. They include, for instance: the Lisp-like functional language and simulator Proto [7], for programming wireless sensor networks with the notion of computational fields; and the rule-based language MGS [21], for computation of and on top of topological complexes.

As pointed out in [9], the successes and failures of the above languages, suggest that arraging adaptive mechanisms to be implicit helps to ensure simple and transarent composition of aggregate-level modules and subsystems. This observation is further pursued by a recent survey [39], which overviews AP and its foundation provided by the FC.

3   Aggregate Programming in FCPP

AP [9, 39] is an approach for programming networks of devices by abstracting away from individual devices behaviour and focusing on the global, aggregate

---

aggregate function declaration

F ::= FUN t d(ARGS, t x∗) {CODE return e; }

---

aggregate expression

e ::= x │ ℓ │ t{e∗} │ ue │ e o e │ p(e∗) │ node.c(e∗) │ f(CALL, e∗)

      │ t x = e; e │ [&](t x∗)->t {return e; } │ e ? e : e

---

type   aggregate function

t ::= T │ bt │ tt<t∗, ℓ ∗ >   f ::= b │ d

---

built-in aggregate functions

b ::= old │ nbr │ spawn │ self │ mod_self │ map_hood │ fold_hood │ mux

---

Figure 3: Syntax of FCPP aggregate functions.

behaviour of the collection of all devices. It assumes only local communication between neighbour devices, and it is robust with respect to devices joining/leaving the network, or failing (open dynamic topology). Beside communicating with neighbours, the devices are capable to perform asynchronous computations. In particular, every device performs periodically the same sequence of operations, with an usually steady rate:

1. collection of received messages,

2. computation of a program that is the same for all the devices, and

3. transmission of messages

AP is formally backed by FC [5], a small functional language for expressing aggregate programs, which currently has three incarnations as a full-fledged DSL: the Scala internal DSL/library *ScaFi (Scala Fields)* [13], the Java external DSL *Protelis* [32], and the C++ internal DSL/library *FCPP* [1]. In this paper we focus on the FCPP incarnation, because it is the only one that lends itself to be ported to devices with constrained resources, such as the ones we consider for IIoT (see Section 4).

FCPP is based on an extensible software architecture, at the core of which are *components*, that define abstractions for single devices (*node*) and overall network orchestration (*net*), the latter one being crucial in simulations and cloud-oriented applications. In an FCPP application, the two types *node* and *net* are obtained by combining a chosen sequence of components, providing the needed functionalities in a mixin-like fashion.

Compared to the original presentation of FCPP [1], we have added a fundamental construct for supporting *aggregate processes* [14], namely the built-in *spawn* function (see below). Aggregate processes can be figured as *computational bubbles* that involve a subset of the devices running a given FCPP program; such bubbles can spring out, expand, perform some work, stretch and vanish. Given

7

a computational bubble, a device can be either within the bubble (i.e., participating in the computation and bubble spreading), external to the bubble (i.e., not participating in the computation), or at the border of the bubble (i.e., participating in the computation but not in bubble spreading).

A fundamental aspect is that every process instance is automatically propagated by all the participating (internal) devices to their neighbours. Therefore, when a device generates a process, it just needs not to leave it immediately in order to propagate it to its neighbours; and so on. Unless nodes explicitly indicate that they are willing to leave the process, the process will tend to expand to every reachable node. On the other hand, when a device leaves a process, even if it happens to be the process creator, it is up to the other nodes still in the process to decide whether they also want to leave (eventually leading up to the termination of the whole process) or not. It is also possible, however, to explicitly initiate a propagating shutdown of the process (through a special status, see below).

The syntax of aggregate functions in FCPP is given in Fig. 3. It should be noted that, since FCPP is a C++ library providing an internal DSL, *an FCCP program is a C++ program* (so all the features of C++ are available). We use $*$ to indicate an element that may be repeated multiple times (possibly zero).

An *aggregate function declaration* consists of keyword FUN, followed by the return type t and the function name d, followed by a parenthesized sequence of comma-separated arguments t x (prepended by the keyword ARGS), followed by an *aggregate expression* e (within brackets and keywords CODE return). *Aggregate expressions* can be either:

- a *variable* identifier x, or a C++ *literal value* ℓ (e.g. an integer or floating-point number);

- an *object* of type t built through a class constructor call $t\{e*\}$ with arguments e;

- an *unary operator* u (e.g. $-$, $\sim$, !, etc.) applied to e, or a *binary operator* e o e (e.g. $+$, $*$, etc.);

- a *pure function call* p(e$*$), where p is a basic C++ function which does not depend on node information nor message exchanges

- a *component function call* node.c(e$*$), where c is a function provided by some component, depending on node information but not on messages;

- an *aggregate function call* f(CALL, e$*$), where f can be either a defined aggregate function name d or an aggregate built-in function b (see below);

- a *let-style statement* t x = e_1; e_2, declaring a variable x of type t with value e_1 referable in e_2;

- a *conditional branching* expression e_{guard} ? $e_\top$: $e_\bot$, such that $e_\top$ is evaluated and returned if e_{guard} evaluates to `true`, while $e_\bot$ is evaluated and returned if e_{guard} evaluates to `false`.

8

The `old` and `nbr` built-in functions, constitute the fundamental constructs of Field Calculus; in FCPP, they are overloaded to several different signatures:

- $old(CALL, v_0, v)$ with v_0, v of type t returns the value fed as second argument v in the *previous round* of computation (thus introducing one round of delay), defaulting to v_0 if no such value is available;

- $old(CALL, v)$ is a shorthand for $old(CALL, v, v)$;

- $old(CALL, v_0, f)$ computes the result of applying f to the value of the whole `old` function at the previous computation cycle (using v_0 if no such value is available);

- $nbr(CALL, v_0, v)$ with v_0, v of type t returns the *neighbouring field* of values fed as second argument v in the previous round of computation of *neighbour nodes*, defaulting to v_0 for the current node if no such value is available for it;

- $nbr(CALL, v)$ is a shorthand for $nbr(CALL, v, v)$;

- $nbr(CALL, v_0, f)$ (whose logic is described in [2] as the *share* operator), computes the result of applying f to the neighbouring field of values of the whole `nbr` function at the previous computation cycle of neighbour nodes (using v_0 for the current node if no such value is available).

The newly implemented `spawn` built-in function has the following signature:

$$spawn(CALL, p, ks, v_0, ...)$$

and spawns an *aggregate process* corresponding to function p for every *key* in the container ks, passing the *values* of the (possibly empty) sequence $v_0, ...$ as further input to each of them. The aggregate process function p takes as arguments a key and a sequence of values, and returns a pair consisting of a result and a process status. Currently, FCPP supports overloads of `spawn` for two different types of process status:

1. `status`, that is one of the following constants:

   (a) `terminated`: the node wants to shutdown the computation (propagating to its neighbours)

   (b) `external`: the node is not part of the computation

   (c) `border`: the node is at the border of the computation (see above)

   (d) `internal`: the node is within the computational bubble

   (e) $*$`_output` (where $*$ is one of `terminated`, `external`, `border`, or `internal`): the node is in status $*$ and the output of function p should be returned by `spawn`

2. `bool`, with `true` and `false` corresponding to the `internal_output` and `border_output` values of type `status`

9

The `spawn` function itself returns an unordered map from the keys of the processes with an *_output state to their output values, so that such output values can be used for further computations in the current round.

The other built-in aggregate functions currently available are:

- `self`(CALL, ϕ), which given a value ϕ of `field<t>` type returns the value $\phi(i)$ taken by the neighbouring field ϕ for the current node (of identifier $i = $ `node.uid`);

- `mod_self`(CALL, ϕ, v), which given a value ϕ of `field<t>` type, returns the same value with $\phi(i)$ changed to v, where $i = $ `node.uid`;

- `map_hood`(CALL, $f, v*$) which applies f point-wise to a sequence of local or field values $v*$;

- `fold_hood`(CALL, f, ϕ) which *folds* the values in the range of ϕ of `field<t>` type through the commutative and associative binary operator f of type (t,t)->t, reducing them to a single value of type t;[2]

- `fold_hood`(CALL, f, ϕ, v) which folds ϕ as above, using v instead of the value of ϕ for the current device: in other words, it is equivalent to `fold_hood`(CALL, f, `mod_self`(CALL, ϕ, v);

- `mux`(CALL, e_c, e_t, e_f), which evaluates all the expressions e_c, e_t, e_f and returns the value of either e_t or e_f based on the Boolean value of e_c; note how `mux` differs from the conditional branching expression described above which evaluates only the branch selected by the condition.

4 Port of FCPP on a DecaWave Board

A fundamental step for using FCPP in real-world IIoT scenarios is porting it to a suitable platform, including hardware and operating system. We have chosen the DWM1001C module produced by Decawave, which is currently used by Reply[3] to offer solutions to some of its industrial customers.

The DWM1001C module integrates the Nordic Semiconductor nRF52832 general-purpose system on a chip (SoC), the Decawave DW1000 Ultra Wideband (UWB) transceiver and the STM LIS2DH12TR 3-axis accelerometer. The nRF52832 SoC offers a 64MHz ARM Cortex-M4 CPU with floating-point unit, a Bluetooth Low Energy (BLE) transceiver with Bluetooth 5 support, a 512 KB flash memory and a 64 KB RAM. Decawave also offers a developer board (DWM1001-DEV) with a battery connector, a charging circuit, eight LEDs, two buttons, a USB connector and a J-Link on board debug probe for debugging and logging.

---

[2]In Field Calculus, a neighbouring field always has at least a value for the current node; thus, folding is well-defined.

[3]The company which co-operated to the present study: `https://www.reply.com`.

The UWB transceiver allows the module to perform communication over an higher range than the BLE transceiver and to compute the distance between two modules (*ranging*) with a precision of up to 10 centimeters. In particular, in our experiments we measured a range of communication in open air of around 70 meters with UWB and of around 25 meters with BLE, with a ranging error of under 30 centimeters with UWB. We also measured the energy consumption by the UWB transceiver to be around three times higher than the BLE transceiver.

We based our FCPP porting on the existing work of the D3S Research Group of the University of Trento[4], which includes a port of the Contiki operating system[5] on the DWM1001C and a UWB driver with ranging API. Contiki is an open source operating system for microcontrollers in the IoT, which provides high level APIs for cooperatives threads (proto-threads, see [18]), timers and networking (with protocols for each network stack layer). We upgraded the porting to the newer Contiki NG,[6] which provides a partial support for the C++ programming language required by FCPP, and we improved the C++ support by integrating the C++ clock API and standard output with the Decawave hardware. Contiki NG offers a low code footprint of just about 100kB and the possibility to configure memory usage to be as low as 10kB. The access to the UWB and BLE features of DWM1001C is granted, respectively, by a port of the UWB driver for Contiki developed by the D3S Research Group (see above), and by the Soft Device for BLE offered by the Nordic SDK for the nRF52832 SoC.

In order to connect FCPP with the UWB and BLE drivers, we had to extend it. In particular, the `hardware_connector` component included in the library handles the communications between physical (hardware) nodes. The constructor of its `node` class receives an object whose type is the class implementing the communication functions on a specific hardware. For our present purposes, we have created two classes (FCPP drivers), one for UWB and one for BLE. In order to work with the `hardware_connector` component, they just need to expose:

- a constructor taking a `node` which represents the current device;

- a constructor taking a `node` and a data structure for configuring the driver;

- a `send` method taking a vector of characters to send; and

- a `receive` method that returns a vector of messages from neighbours.

The UWB FCPP driver exploits the native UWB driver to broadcast all communications between devices over the CSMA protocol with the UWB transceiver, reaching a greater distance of communication compared to BLE at the expense of an higher energy consumption. The UWB configuration allows to send messages of size up to 116 bytes.

---

[4]See `https://github.com/d3s-trento/contiki-uwb`.

[5]See `https://github.com/contiki-os/contiki`.

[6]See `https://github.com/contiki-ng/contiki-ng`.

The BLE FCPP driver exploits the native BLE driver to broadcast the messages using the Bluetooth 5 extended advertisement. Moreover, it uses the UWB transceiver to perform the ranging between devices for computing their distances. The main goal of the protocol for intermixing BLE communication with ranging is that of keeping the UWB transceiver in sleep mode for as long as possible to reduce energy consumption. More precisely, the main steps are as follows:

- when a node sends a message through the BLE native driver, 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 node then prepares to do ranging with the neighbours in the list at the beginning of its next round; and

- when a node receives a message through the BLE native driver, and it appears in the prefix list, it prepares to do ranging with the sender at the beginning of the next round of the sender.

Thanks to the synchronization information exchanged by piggybacking the BLE messages, the nodes can turn on their UWB transceivers just for the time needed for performing the ranging operations. The BLE configuration allows to send messages of over 200 bytes, with the actual size depending on the ranging configuration, i.e., on the maximum size of the prefix devoted to synchronize ranging, which is not available for the regular payload.

As mentioned above, a serious constraint of the DWM1001C module is the quantity of RAM, limited to 64kB. Thanks to the small footprint of Contiki NG and of FCPP, it has been possible to leave approximately 16kB of stack space and 16kB of heap space to be used by applications built on FCPP.

5 AP Services in an Industrial IoT

5.1 Overall Hardware Architecture

Figure 4 shows the schematic architecture that we assume for the IIoT scenarios we address. It is derived from the one in Figure 1, but is specifically tailored to the assumptions and focus of this paper.

We assume two networks, based on different technologies and covering different layers:

1. a *Far Edge net* $\mathcal{N}_{EDGE}$ composed of low-end computational nodes that communicate point-2-point based on proximity. Corresponds to the *Edge* layer in Figure 1 but assumes specifically point-2-point wireless communication; and

2. an *Intranet and Cloud net* $\mathcal{N}_{INET}$ that connects more powerful systems within the factory among them and, possibly, with external cloud systems, using standard internet technologies and protocols. Corresponds to the
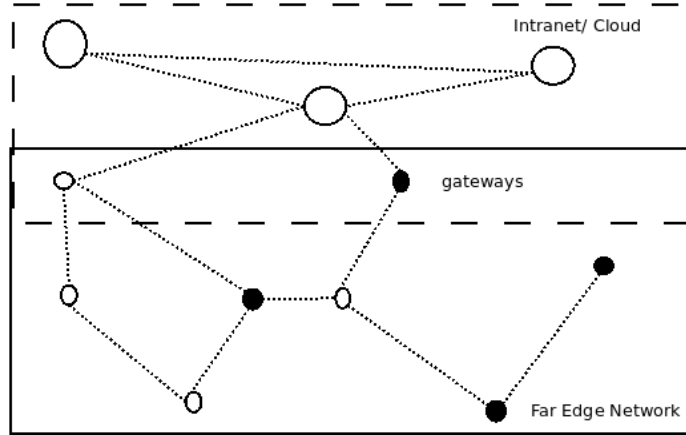
Figure 4: Schema of the Architecture for the IIoT Service Scenarios.

Network and *Cloud* layers in Figure 1, that we do not need to distinguish for our purposes.

We envision the following types of computational nodes:

1. *fixed* nodes N_F of the $\mathcal{N}_{EDGE}$ net, associated with fixed equipment such as machines, controllers, or fixed sensors (depicted as empty circles in Figure 4);

2. *mobile* nodes N_M of the $\mathcal{N}_{EDGE}$ net, associated with mobile equipment (e.g., forklifts) and users (depicted as full circles in Figure 4);

3. *gateway* nodes N_G that are nodes that can communicate with both the $\mathcal{N}_{EDGE}$ nodes and the $\mathcal{N}_{INET}$ nodes, thus making it possible to route data between the two nets; and

4. *cloud* central system nodes N_C that can provide: long term storage; expensive data analysis and ML; connection to a Digital Twin; etc.

It should be noted that the set of nodes N_M can change dynamically quite often, since, e.g., users enter/leave the factory or moving machines are switched on/off. Moreover, gateway nodes $g \in N_G$ may be either fixed or mobile.

Sensors and actuators can be associated with both mobile and fixed nodes. For convenience, we spell two important categories:

1. *distance* sensors S_{DIST}, that can detect the distance between nodes of the $\mathcal{N}_{EDGE}$; and

2. *parameter* sensors S_{PARAM}, that measure the values of relevant parameters at their location.

We do not detail how the nodes of $\mathcal{N}_{EDGE}$ communicate with their associated sensors and actuators: they may, e.g., be directly connected through a common physical board, or use a short-range wireless technology such as BLE. We
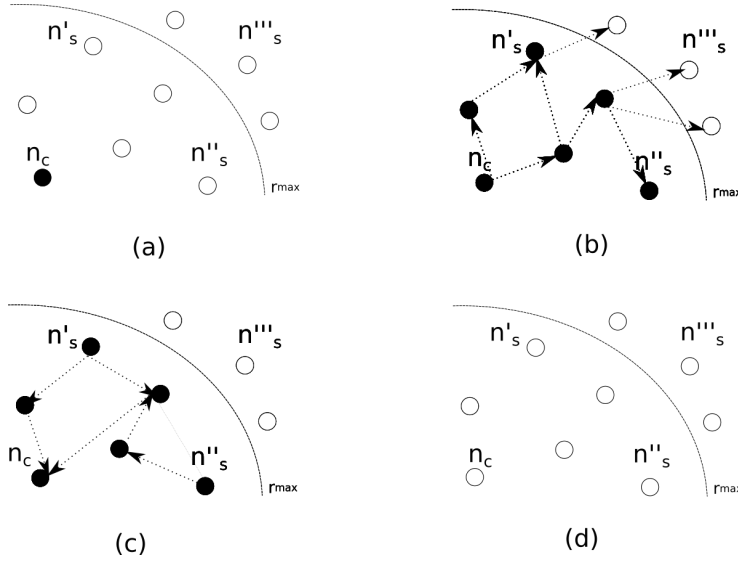
Figure 5: Life of an aggregate process spawned by a client node (a-d). See explanation in the text.

just assume that $\mathcal{N}_{EDGE}$ can retrieve data from associated sensors and issue commands to associated actuators.

5.2 Services Software Architecture

AP-based services consist of the execution of FCPP programs by the nodes of the $\mathcal{N}_{EDGE}$. An important role, however, is played by Aggregate Processes and how they are generated, managed, and terminated.

Consider a node n_c (*client*) that must reach another node n_s (*server*) for receiving a service or an information. A simple schema would be to have n_c broadcast its request into the $\mathcal{N}_{EDGE}$, then collect the replies from the available service providers $n'_s, n''_s, \ldots$, and finally choosing to adopt, e.g., the reply of the closest one, and discard the others. The problem with this schema, is that, in order for it to work, every node in the $\mathcal{N}_{EDGE}$ should be executing an FCPP program that includes the logic for the broadcast from n_c and the generation and collection of replies from service provider nodes. This is obviously not practical, since the client can be any node in $\mathcal{N}_{EDGE}$ and there may be many possible types of requests that must be handled and answered according to different logics.

This is where aggregate processes come into play as a fundamental building block of AP systems. In a situation like the one described above:

- The *client* node n_c creates a process, by passing a suitable key (e.g., its

node id id_c) to a `spawn` that appears in the FCPP program being executed:

$$\texttt{spawn}(\texttt{CALL}, p, \{id_c\}, v_0, ...)$$

Note that, up to the current round the `spawn` has always been executed by node n_c as part of the program, but since it was given an empty set of keys, it immediately returned as a no-operation (if it didn't execute processes generated by other clients). Figure 5 (a) depicts n_c as a full circle to indicate that it participates to the new process.

- The function executed by the new process is the one specified as the p argument to `spawn`, and of course depends on how exactly the interaction between n_c and the service provider(s) should happen; it receives the key of the process id_c and the additional parameters $v_0, ...$

- Typically, n_c wants to place a limit r_{max} to the maximum radius of the process expansion, and this can be achieved by passing r_{max} (shown as a dotted arc in Figure 5) as one of the additional parameters $v_0, ...$ of `spawn` that are forwarded to p.

- The process created by n_c starts to spread automatically to the neighbours of n_c, and on to more and more nodes $n_p \in \mathcal{N}_{EDGE}$. Figure 5 (b) shows as full circles the nodes within r_{max} to which the process propagates.

- Consider a node n_p to which the process has just been propagated:

  - n_p executes the p function, which, exactly as in n_c, receives the key of the process id_c and the other parameters including r_{max}.

  - The body of function p should be such that node n_p gets to know whether it is beyond the maximum radius r_{max} from node n_c (e.g., by participating to a *gradient* aggregate computation, see section 5.4).

  - If so, n_p should immediately leave the process (i.e., return an `external` or `false` status). In Figure 5 (b), the nodes beyond r_{max} remain empty circles although the nodes within the bubble try to propagate the process to them.

  - Otherwise, it should determine whether it can (try to) provide the service requested by n_c; if yes, it should send its reply to n_p (e.g., by participating as a producer to a *collection* aggregate computation, see section 5.4). Figure 5 (c) shows the replies originating from the server nodes n'_s, n''_s flowing towards n_c through the $\mathcal{N}_{EDGE}$.

  - When n_c decides to terminate the process, it initiates the shutdown by returning a `terminated` status, which propagates to the other nodes within the bubble. Figure 5 (d) depicts all the nodes, including n_c, as empty circles to indicate that they have all left the process.

5.3 Safety Services

An important set of AP-based services that can be implemented in the architecture of Figure 4 are those that improve the safety of the industrial environment and, in particular, of the people that populate it at any given time.

The typical schema for the AP implementation of a safety service is the following:

1. let N_{SAFE} be the subset of nodes of the $\mathcal{N}_{EDGE}$ net involved in the safety service;

2. nodes N_{SAFE} continuously exchange relevant information (i.e., their velocity) with neighbours, and receive data from their sensors;

3. in particular, if the node is equipped with a distance sensor $s \in S_{DIST}$, the built-in funtion `nbr_dist` returns a field of type `field<real_t>` associating each neighbour UID (of type `device_t`) with its distance from the current node (of type `real_t`);

4. the sensor data is evaluated w.r.t. the internal status and with the safety-related data collected from the neighbours;

5. if a danger is detected, the node can react immediately (e.g., stop moving) and initiate an information sharing process to share the detected danger with other $\mathcal{N}_{EDGE}$ and/or $\mathcal{N}_{INET}$ nodes (see sections 5.4, 5.5).

It should be noted that even a simple safety process as the one just described can take advantage of the intelligence provided by AP. For example, the safety threshold of the distance between two nodes can depend on whether the two nodes are not moving, moving towards each other, or moving away from each other. This can be easily achieved with AP.

5.4 Information Sharing for Intelligent Decisions

The sharing of information among $\mathcal{N}_{EDGE}$ nodes (without involving the $\mathcal{N}_{INET}$ nodes) can provide many useful, robust and low-latency services. In particular, when executed as an Aggregate Process, it can support the request of a service by a node and the collection of the replies from potential servers, as explained in section 5.2.

Information sharing can typically follow two alternative ways: the first one assumes that the nodes that directly sense (or, more generally, own) relevant information *spread* it to the rest of the net; the second one assumes that nodes that need information, *collect* it from the other nodes possibly *accumulating* it into a suitable data structure. The spreading and collection of information are fundamental blocks of the AP paradigm, and are sometimes named, respectively, G-block and C-block in the literature [38]. Referring again to the service request/reply described in section 5.2, the client node can spread its request with a G-block, and collect the replies with a C-block.

```
GEN(P,T) T broadcast(ARGS, P const& distance, T const& value) { CODE
  return nbr(CALL, value, [&] (field<T> x) {
    return get<1>(min_hood(CALL,
                           make_tuple(nbr(CALL, distance), x),
                           make_tuple(distance, value)));
  });
}
```

Figure 6: The broadcast function implementation in FCPP.

The typical schema for the AP implementation of information spreading in our reference setting is the following:

1. let N_{CONS} be the subset of nodes of the $\mathcal{N}_{EDGE}$ net interested in the information produced by a node n_{PROD}

2. node n_{PROD} produces the information and broadcasts it (see below) to all the $\mathcal{N}_{EDGE}$ nodes

3. the nodes outside N_{CONS} are used just to propagate the information, while the nodes in N_{CONS} also pick-it up and use it for their computations, possibly including making decisions, and posting a reply

Figure 6 shows the implementation of the `broadcast` function in FCPP. First of all, we note that it is a templated function, which parametrizes the types `P`, `T` of the `distance` and `value` arguments: in this case, the `FUN` keyword is substituted with the alternative $GEN(T^*)$ listing the parameter types. The `distance` parameter represents the (estimated) distance of the node from the source of the information, i.e. it is 0 for the node that generates the information, and can be estimated with a suitable aggregate function for the other nodes in $\mathcal{N}_{EDGE}$. The FCPP library offers several functions for the distance estimation, ranging from a basic function `abf_hops` counting the number of hops from the source with the Bellman-Ford algorithm, to sophisticated functions such as `bis_distance` [4], or `flex_distance` [6]. Clearly, a more accurate estimate can be achieved if the underlying system provides S_{DIST} sensors for measuring the distance between each node and its neighbours.

The function body uses the form of `nbr` that takes a lambda function that processes the field `x` of most-recent values exchanged with neighbours to derive the new value for the current node. The lambda applies `min_hood` to pairs (d, v) for each neighbour δ, where d is the distance of δ from the source and v is the value held by δ; the result of `min_hood` is thus a pair (d', v') corresponding to the neighbour closest to the source. Finally, the use of `get` extracts the value v', and such a value is returned by `broadcast` and will be associated with the current node in the field `x` when `broadcast` is computed again in the next round.

The AP implementation of information collection and accumulation in our reference setting typically has the following schema:

17

```
GEN(P, T, U, G)
T sp_collection(ARGS, P const& distance, T const& value,
                      U const& null, G&& accumulate) { CODE
  return nbr(CALL, (T)null, [&](field<T> x){
    device_t parent = get<1>(min_hood(CALL,
                                      make_tuple(nbr(CALL, distance),
                                                 nbr_uid(CALL)) ));
    return fold_hood(CALL, accumulate,
                          mux(nbr(CALL, parent) == node.uid, x, (T)null),
                          value);
  });
}
```

Figure 7: The single-path collection function implementation in FCPP.

1. let N_{PROD} be the subset of nodes of the $\mathcal{N}_{EDGE}$ net involved in the production of information that must be collected by a node n_{CONS};

2. each node in N_{PROD} produces a piece of information and aggregates it into the data that is collected towards n_{CONS} through the other $\mathcal{N}_{EDGE}$ nodes (see below);

3. when the aggregated data reaches n_{CONS} it is picked-up and used.

Note that, in the second step, it is possible to aggregate the information while it flows from the N_{PROD} nodes to the n_{CONS}. For instance, imagine that a client node n_c has spawned a process to ask for a service, and it then collects the answers of servers $n'_s, n''_s, \ldots$ consisting of pairs `pair<real,real>` where the second element is the value of the reply, and the first element is the degree of confidence that the server had in that reply. As the replies flow towards n_c, each node in $\mathcal{N}_{EDGE}$ will propagate only the pair with the highest first element. In this case, the aggregation function is thus the *max* function, but any other aggregation function could be used in the process.

Figure 7 shows the implementation of the `sp_collect` function in FCPP. The FCPP library also offers more sophisticated multi-path collection functions, namely `mp_collect` and `wmp_collect` [3], but the simpler `sp_collect` serves well our current explanation purposes. The `sp_collect` templated function has the following type parameters: `P`, of the `distance` parameter, representing the (estimated) distance of the node from the consumer of the information; `T`, of the `value` parameter, representing the aggregate value awaited by the consumer; `U`, of the `null` parameter, the identity element of the accumuation function; and `G`, of the `accumulate` parameter, representing the accumulation function that should take two `T` values and aggregate them into a a single output `T` value.

Similarly to `broadcast`, the function body uses the form of `nbr` that takes a lambda function that processes the field `x` of most-recent values exchanged with neighbours to derive the new value for the current node. First, the lambda applies `get<1>` to the result of a `min_hood` to get the node `uid` of the neighbour closest to the consumer; we store the result in a `parent` variable, to stress the

fact that the flow of information follows a tree[7] from the furthest nodes to the consumer itself.

Then, the `accumulate` function is exploited by `fold_hood` to aggregate a field of `T` values into a single result of type `T`. Such a value is returned by `sp_collection` and will be associated with the current node in the field `x` when `sp_collection` is computed again in the next round. Using the `mux` function, the field aggregated by `fold_hood` associates to each neighbour δ' of the current node δ a `T` value as follows: if δ' has determined that δ is its `parent`, the most recent value sent by δ' (contained in the `x` field); otherwise the identity element `null` of `accumulate`. In other words, the current device δ aggregates all and only the values received from neighbours that chose it as their parent.

5.5 Interaction with the Cloud

The sharing of information between the $\mathcal{N}_{EDGE}$ nodes and the $\mathcal{N}_{INET}$ nodes, and especially between the edge and the N_C nodes of the $\mathcal{N}_{INET}$ that host cloud services, is different than the $\mathcal{N}_{EDGE}$-level sharing described in the previous section in the following main respects:

- all the communications must necessarily go through the gateway nodes N_G;

- the amount of data collected by cloud nodes can be much higher, in general, than that required by services provided completely within the $\mathcal{N}_{EDGE}$;

- typical benefits of the AP paradigm such as low latency, robustness, and privacy, which apply to the $\mathcal{N}_{EDGE}$ services, may not apply to services that also require information to cross the $\mathcal{N}_{INET}$.

Given these characteristics, the following additional mechanisms can be suitable:

- distribute the workload among N_G gateway nodes as far as possible;

- ensure some redundancy in the data transmitted to the N_C cloud nodes (for improving both latency and robustness).

In FCPP, the distribution of workload can be easily done by partitioning the nodes in $\mathcal{N}_{EDGE}$ in as many regions as there are gateway nodes in N_G. A design pattern specifically created to achive these goals in a fully distributed fashion through FC itself is the SCR (Self-Organising Coordination Regions) pattern described in [31]. The pattern supports distributed selection of the leaders, but in case all the N_G nodes are used, the selection becomes trivial. Also the formation of the regions can be trivial in its simplest form, whereby each node decides to belong to the region associated with the the closest N_G node.

---

[7]It is easy to see that such a tree is a Single Source Shortest Path (SSSP) tree with the consumer as source, on the (unweighted) network graph.

As for ensuring redundancy, a natural approach is to extend the SCR pattern in such a way that regions overlap, i.e., each node belongs to two or more regions. Again, the criteria adopted by a node to select the regions to join can have varying degrees of complexity depending on the context. If the N_G nodes can be partitioned a-priori into two or more subgroups or *types*, a simple approach consists of each node joining one region per type.

6 Experimental Validation

6.1 Case Study: Warehouse App

To validate the proposed approach experimentally, we consider a scenario inspired by use-cases currently being investigated in Reply of smart warehouse management. We assume that warehouse workers move around a series of aisles with forklifts moving at a maximum speed of 10 km/h (a standard for forklifts). Pallets containing goods are arranged in a regular grid along aisles, while some empty pallets are available in a common loading zone, where every load and unload operation is performed. We assume that the 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. As a byproduct, a digital representation of the warehouse status (e.g., a digital twin) is usually inaccurate or impossible. Furthermore, workers may occasionally run into each other at aisle joints, damaging goods and slowing down the warehouse operations.

In order to overcome these issues, we propose a warehouse management app realising the following services:

1. *preventing accidental collisions*, by warning workers whenever another forklift is approaching with a speed greater than a threshold, within a given safety radius;

2. *providing route information* towards either empty spaces and goods matching a given query, presented to the interested warehouse worker by turning on led lights on neighbouring smart devices;

3. *collecting logs* of relevant events (loading/unloading of goods and collision warnings) towards central points that are connected to the cloud.

We assume that the app runs on a network of DWM1001C modules, where each pallet and forklift has an associated module. Pallet modules have lower power (to save battery life), and only present output in the form of small led lights (for routing). Instead, forklift modules are connected with a simple application on the workers' personal smartphone, which allows the worker to provide some basic input (i.e., logging loading and unloading details, issuing routing

requests), shows her some basic output (i.e., collision warnings and additional route information), and stores locally the collected logs, uploading them on the cloud as soon as possible.

Service 1 is realised as a simple aggregate process, that is spawned by every forklift module and extends until reaching the safety radius. In this area, the distance towards the closest other forklift is gathered: if this distance decreases faster than the threshold, a warning is issued.

Service 2 is also realised through aggregate processes. One process computes routes towards empty spaces (more precisely, pallets that detect an empty space around them), others compute routes towards goods satisfying given queries, as they are issued. Each process expands naturally into the whole network, and is terminated everywhere once the routing request is cancelled.

Service 3 is realised through two simultaneous aggregate collection processes, to enable redundancy and greatly reducing the chances of information losses. Forklift modules are used as collection sinks, since they can upload data on the cloud: based on their unique identifier, half of them are assigned to sink group 1 and the other half to sink group 2. The logs produced are collected twice, towards the closest sink in group 1 and towards the closest sink in group 2. The collection algorithm used is a custom version of *multi-path collection*, designed to keep the network load low for the specific log collection task. Firstly, hop-count 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.

6.2 Simulations

Firstly, we simulated the operations of a smart warehouse empowered by the proposed app through the FCPP simulation framework [1] for aggregate computing.[8] A screenshot of the simulation is shown in Figure 8. The simulated warehouse consists of 22 rows × 3 columns of aisles; each of them composed by 15 slots in the horizontal direction × 3 slots in the vertical direction. The bottom part is dedicated to a loading zone (left) and office space (right).

Pallets are represented as cubes with a $1m$ side (spaced $1.5m$ from each other), while forklifts are represented as spheres. The content of a pallet is displayed by the colour of its middle band: white represents no content, and various colours are used for 100 different types of goods (randomly generated according to a Zipf distribution [46], so that few goods are very common, and many are uncommon). The middle band of a forklift is coloured analogously according to the good that the forklift is currently searching or loading (if any, black otherwise). For both pallets and forklifts, the lateral bands are grey if the device is idling, yellow if it has a led turned on (pallet routing or forklift signalling a collision risk), and red during handling (loading/unloading for fork-

---

[8]Code available at: `https://github.com/fcpp-experiments/warehouse-case-study`
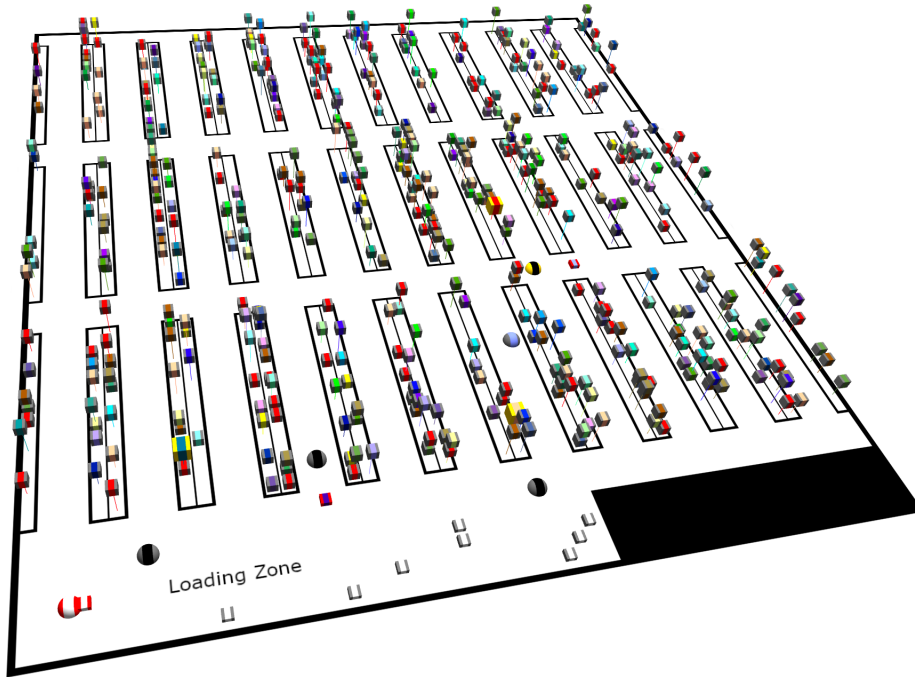
Figure 8: Screenshot of the simulation execution.

lifts, being carried by a forklift for pallets). Forklifts randomly perform either a retrieve task of a specific good (picking up a matching pallet, and bringing it to the loading zone for unload), or a insert task (where an empty palled is filled up and then brought to an empty space in the aisles). Tasks are generated randomly during idling times.

In Figure 8, 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, coloured red and white). 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. Another forklift (bottom of the 7th vertical aisle, grey 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).

Figure 9 presents few performance indicators of the proposed app through the course of the first 500 seconds of simulated time. The size of messages is
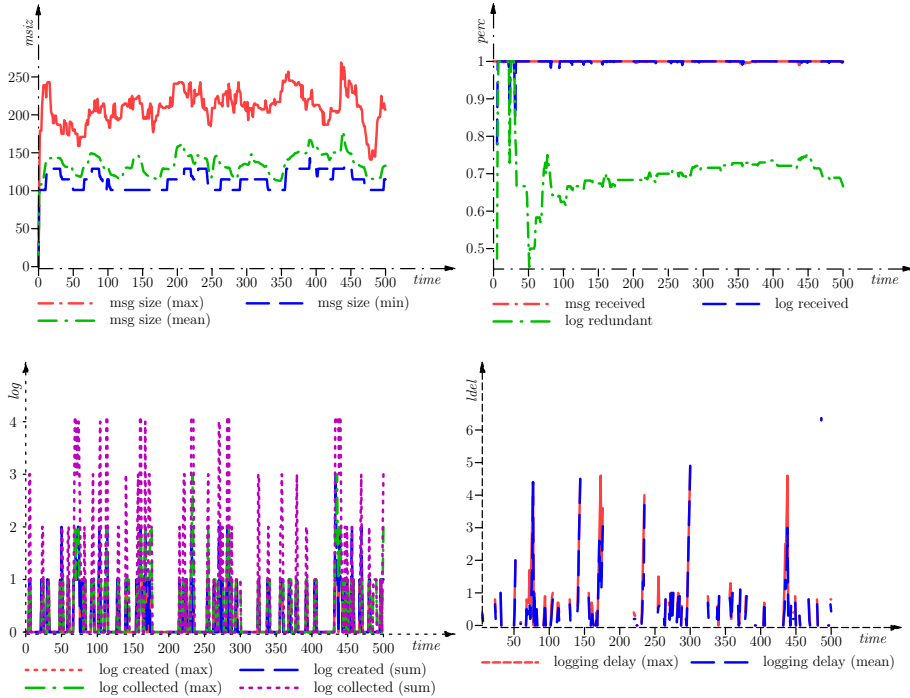
Figure 9: Plots of simulated performance over time: message size in bytes (top-left); percentage of messages delivered, log received at least once and received twice (top-right); logs created and collected (bottom-left), average delay of log collection (bottom-right).

usually below 150 bytes, with peaks below 250 bytes: almost every message is small enough to be sent, as the message limit of the modules is currently of 222 bytes (and 20 bytes are used by the simulation logics, and would not be used in a deployment). Every log that is created is eventually received, but only about 70% of them are received twice in both sink groups, 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).

6.3   Physical Deployment: a Proof-of-Concept

We validated the aggregate program implemented for the simulation on a batch of seven DWM1001-DEV modules. Six modules were configured as if they were attached to pallets, four of them having some goods already loaded and two
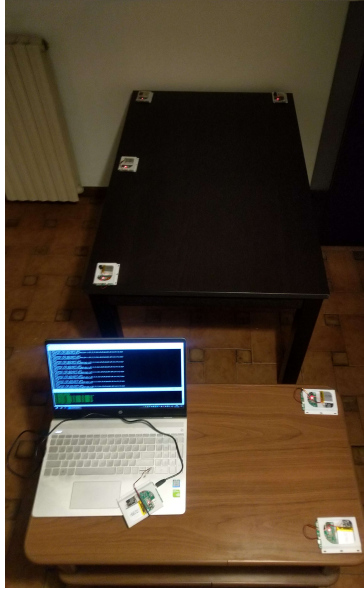
Figure 10: Layout of the devices in the Proof-of-Concept

empty, while one module was configured as the personal device of a forklift operator. The devices were arranged as visible in Figure 10, with the modules associated to loaded pallets on the darker table, the module of the forklift operator connected to a PC to read the logs in real time and the modules associated to empty pallets near the PC. The BLE transmission power was configured to -16dBm to reduce the range of communication due to the small space available.

Two scenarios of actions performed by the operator were tested: the loading of goods on an empty pallet followed by its placement near the already loaded pallets, and the search for a pallet containing a predermined kind of good followed by its unloading near the empty pallets. In both scenarios the operator interacts with the modules by using the button available on the personal module while receiving a feedback from the LEDs. In the former scenario the operator marks the nearest pallet as loaded with some good by using the button on the personal module, then finds an empty space near the previously loaded pallets by following the ones with the LEDs turned on. In the latter scenario the operator starts the request to find a kind of good by pressing the button on the personal module, then follows the turned on LEDs on the loaded pallets until reaching one containing the requested goods. The LEDs on that pallet start to blink 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.

At the end of each test the FCPP logs, from both the personal module and the module representing the pallet handled by the operator, were collected on the PC to verify that the network presented the expected behavior. In all tests

the network was left in a consistent state but all the computation were slower than expected (with reaction times of 5s to 10s) due to an high number of dropped messages in both communication and ranging.

7 Conclusions

We have considered intelligent services for the IIoT, deployed on far-edge devices placed at fixed locations in the workshop floor or attached to people and moving machines, and sharing information with cloud servers through edge gateways.

Previous work on offloading part of the computational and storage needs to the far edge of the IIoT mostly focusses on data management and proposes architectural schemes to satisfy specific needs of efficient storage distribution or edge-to-cloud communication (c.f. Section 2). Compared to such existing work, we show that by exploiting the `spawn` construct of FCPP it is possible to implement any kind of service that a set of client nodes may require from a set of potential server nodes by querying the far-edge network (including safety, information retrieval, path planning). Additionally, the AP paradigm guarantees "for free" that the system is open and adaptable, thus allowing node failures, mobile nodes, and nodes joining/leaving the network at any time.

A valuable contribution of the paper is the actual deployment of AP-based applications on physical IoT boards with highly constrained resources. Without the highly optimized FCPP library it would have been impossible to achieve these results. Alternative libraries implementing AP based on the JVM [41], would introduce a memory footprint that exceeds by far the resources of the target devices.

We want to further pursue the research described here in several directions. First of all, we would like to experiment with the physical deployment in a full-size, real world factory setting, to assess the scalability and reliability of our systems when faced with a noisy, highly dynamic environment. In particular, we envision the need for a better synchronization between FCPP computation rounds and the ranging operations, and for retransmission protocols, since with the current deployment many messages are lost. The deployment of a larger number of devices in an area corresponding to a real warehouse should also provide valuable feedback for tuning our system.

We would also like to exploit the ranging capabilities offered by the DecaWave boards to implement an efficient and accurate cooperative RTLS (Real Time Location System) based on FCPP. Such a service would open the way for many other interesting services to be offered at the far edge. In particular, we envision the possibility of designing "intelligent" triangulation algorithms for 3D position estimation, to be exploited in routing and discovery services.

Finally, we are considering to apply our approach to scenarios involving large numbers of mobile robotic agents that need to coordinate in an indoor space (e.g., factory, warehouse) to achieve global goals.

Acknowledgements

References

[1] 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.

[2] G. Audrito, J. Beal, F. Damiani, D. Pianini, and M. Viroli. The share operator for field-based coordination. In *Coordination Models and Languages (COORDINATION)*, volume 11533 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 2019.

[3] G. Audrito, S. Bergamini, F. Damiani, and M. Viroli. Effective collective summarisation of distributed data in mobile multi-agent systems. In *18th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1618–1626. IFAAMAS, 2019.

[4] G. Audrito, F. Damiani, and M. Viroli. Optimally-self-healing distributed gradient structures through bounded information speed. In *Coordination Models and Languages (COORDINATION)*, volume 10319 of *Lecture Notes in Computer Science*, pages 59–77. Springer, 2017.

[5] G. Audrito, M. Viroli, F. Damiani, D. Pianini, and J. Beal. A higher-order calculus of computational fields. *ACM Trans. Comput. Log.*, 20(1):5:1–5:55, 2019.

[6] J. Beal. Flexible self-healing gradients. In *ACM Symposium on Applied Computing (SAC)*, SAC '09, pages 1197–1201. ACM, 2009.

[7] J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, 2006.

[8] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013.

[9] J. Beal, D. Pianini, and M. Viroli. Aggregate programming for the Internet of Things. *IEEE Computer*, 48(9):22–30, 2015.

[10] L. Bettini, V. Bono, R. De Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The Klaim project: Theory and practice. In *Global Computing*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer, 2003.

[11] L. Cardelli and P. Gardner. Processes in space. In *6th Conference on Computability in Europe*, volume 6158 of *Lecture Notes in Computer Science*, pages 78–87. Springer, 2010.

[12] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[13] R. Casadei, M. Viroli, G. Audrito, and F. Damiani. Fscafi : A core calculus for collective adaptive systems programming. In *ISoLA (2)*, volume 12477 of *Lecture Notes in Computer Science*, pages 344–360. Springer, 2020.

[14] 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.

[15] Y. Chen, R. H. Katz, and J. D. Kubiatowicz. Scan: A dynamic, scalable, and efficient content distribution network. In *International Conference on Pervasive Computing*, pages 282–296. Springer, 2002.

[16] D. Coore. *Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer*. PhD thesis, MIT, Cambridge, MA, USA, 1999.

[17] R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A language-based approach to autonomic computing. In *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 25–48, 2013.

[18] 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.

[19] J. L. Fernandez-Marquez, G. D. M. Serugendo, S. Montagna, M. Viroli, and J. L. Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, 12(1):43–67, 2013.

[20] T. Finin, R. Fritzson, D. McKay, and R. McEntire. Kqml as an agent communication language. In *29th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 456–463. ACM, 1994.

[21] J.-L. Giavitto, C. Godin, O. Michel, and P. Prusinkiewicz. Computational models for integrative and developmental biology. Technical Report 72-2002, Univerite d'Evry, LaMI, 2002.

[22] M. Inchiosa and M. Parker. Overcoming design and development challenges in agent-based modeling using ascape. *Proceedings of the National Academy of Sciences of the United States of America*, 99(Suppl 3):7304, 2002.

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

[24] S. R. Madden, R. Szewczyk, M. J. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.

[25] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. on Software Engineering Methodologies*, 18(4):1–56, 2009.

[26] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.2*, 2009.

[27] R. Milner. Pure bigraphs: Structure and dynamics. *Information and Computation*, 204(1):60 – 122, 2006.

[28] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I. *Information and Computation*, 100(1):1–40, 1992.

[29] R. Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, Cambridge, MA, USA, 2001.

[30] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Workshop on Data Management for Sensor Networks*, pages 78–87, 2004.

[31] D. Pianini, R. Casadei, M. Viroli, and A. Natali. Partitioned integration and coordination via the self-organising coordination regions pattern. *Future Gener. Comput. Syst.*, 114:44–68, 2021.

[32] D. Pianini, M. Viroli, and J. Beal. Protelis: practical aggregate programming. In *30th ACM Symposium on Applied Computing (SAC)*, pages 1846–1853. ACM, 2015.

[33] S. Rao and R. Shorey. Efficient device-to-device association and data aggregation in industrial iot systems. In *2017 9th International Conference on Communication Systems and Networks (COMSNETS)*, pages 314–321, 2017.

[34] T. P. Raptis and A. Passarella. A distributed data management scheme for industrial iot environments. In *2017 IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 196–203, 2017.

[35] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund. Industrial internet of things: Challenges, opportunities, and directions. *IEEE Transactions on Industrial Informatics*, 14(11):4724–4734, 2018.

[36] E. Sklar. Netlogo, a multi-agent simulation environment. *Artificial life*, 13(3):303–311, 2007.

[37] M. H. ur Rehman, I. Yaqoob, K. Salah, M. Imran, P. P. Jayaraman, and C. Perera. The role of big data analytics in industrial internet of things. *Future Generation Computer Systems*, 99:247–259, 2019.

[38] M. Viroli, G. Audrito, J. Beal, F. Damiani, and D. Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Transactions on Modeling and Computer Simulation*, 28(2):16:1–16:28, 2018.

[39] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini. From distributed coordination to field calculus and aggregate computing. *J. Log. Algebraic Methods Program.*, 109, 2019.

[40] M. Viroli, M. Casadei, S. Montagna, and F. Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Transactions on Autonomous and Adaptive Systems*, 6(2):14:1 – 14:24, 2011.

[41] M. Viroli, R. Casadei, and D. Pianini. Simulating large-scale aggregate mass with alchemist and scala. In *Federated Conference on Computer Science and Information Systems (FedCSIS)*, volume 8 of *Annals of Computer Science and Information Systems*, pages 1495–1504. IEEE, 2016.

[42] M. Viroli, D. Pianini, and J. Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In *Coordination Models and Languages (COORDINATION)*, volume 7274 of *Lecture Notes in Computer Science*, pages 212–229. Springer, 2012.

[43] M. Viroli, D. Pianini, S. Montagna, G. Stevenson, and F. Zambonelli. A coordination model of pervasive service ecosystems. *Science of Computer Programming*, 110:3 – 22, 2015.

[44] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Mobile Systems, Applications, and Services (MobiSys)*. ACM Press, 2004.

[45] K.-L. Wu, P. Yu, and J. Wolf. Segmentation of multimedia streams for proxy caching. *IEEE Transactions on Multimedia*, 6(5):770–780, 2004.

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