FCPP+Miosix: Scaling Aggregate Programming to Embedded Systems

Giorgio Audrito, Federico Terraneo, William Fornaciari

Abstract—As the density of nodes capable of sensing, computing and actuation increases, it becomes increasingly useful to model an entire network of physical devices as a single, continuous space-time computing machine. The emergent behaviour of the whole software system is then induced by local computations deployed within each node and by the dynamics of the information diffusion. A relevant example of this distribution model is given by *aggregate programming* and its minimal set of functional constructs used to manipulate distributed data structures evolving over space and time, and resulting in robustness to changes.
In this paper, we propose the first implementation of the aggregate computing paradigm targeting microcontrollers, by integrating FCPP, a C++ implementation of the paradigm, with Miosix, a modern operating system for microcontrollers with full C++ support. To the best of the author's knowledge, we are the first to present results on the effectiveness of FCPP in an embedded operating system setting as opposed to a simulation environment, thus considering tight memory and computational constraints and accounting for packet losses due to nonidealities of the radio channel. We implemented and tested on a network of WandStem nodes two benchmark applications: a network connectivity checker for network planning and preventive maintenance, and a decentralised contact tracing application. Additionally, we show that common problems in sensor networks such as neighbour discovery, construction of a graph of the network topology, coarse grain clock synchronisation as well as network monitoring and the collection of statistics (such as memory occupation data) can be easily performed thanks to the expressive semantics of aggregate programming.

Index Terms—Embedded Systems, Aggregate Programming, Distributed Systems.

1 Introduction

A prominent challenge in distributed systems programming is achieving reliable operation in networks where communication links are unreliable, such as in Wireless Sensor Networks (WSN) and Internet of Things (IoT) scenarios. This issue is further exacerbated as the size of the network of interconnected devices grows, making it increasingly difficult to pinpoint and debug problems arising out of the interaction of complex distributed embedded systems with limited computational capabilities and memory capacity.

The compound of the aforementioned issues is soon expected to present a significant roadblock in the adoption of the Internet of Things (IoT) and fog computing in general, as these technologies are deployed forming interconnected systems of a real-world scale.

To overcome these issues, new distributed systems programming paradigms have emerged, abstracting from individual device programming and modelling instead an entire network of devices as a single, continuous space-time computing machine where the behaviour of the whole software system emerges out of local computations performed by each node and by the dynamics of information diffusion. Among those approaches, we focus on the Aggregate Programming (AP) paradigm [1] with its companion formal language Field Calculus (FC) [2].

Although this promising paradigm significantly raises the abstraction level at which distributed systems can be programmed, existing implementations are limited to simulation environments only [3], [4], [5]. There is thus a lack of a software platform to demonstrate the benefits of AP in real-world deployments, a matter that we address in this paper by presenting and releasing as open source the integration of FCPP, an optimised implementation of the AP paradigm, with the Miosix OS.

The resulting software framework demonstrates scalability down to distributed embedded systems even when cost or energy constraints demand the use of low power microcontrollers. Moreover, the presented software framework allows to demonstrate how common building blocks of distributed systems such as leader election, reconstruction of the network topology graph, network monitoring and coarse grain clock synchronisation can be conveniently implemented thanks to the expressive semantics of AP.

Summarising, in this paper we provide the following contributions. 1) We present the first software framework for programming real-world distributed embedded systems using AP and release it as open source. 2) We demonstrate the advantages of programming distributed embedded systems using AP by showing how relevant examples of distributed systems problems can be concisely solved using AP. 3) We provide experimental results about the deployment of AP in real-world scenarios and compare them to simulations, including a discussion of the effect of packet losses due to non-idealities of the radio channel. We do so by means of two benchmark applications: a network connectivity checker for network planning and preventive maintenance, and a decentralised contact tracing application.

The paper is organised as follows. Section 2 provides the reader with the necessary background regarding aggregate programming (AP) and the main characteristics of

- *G. Audrito is with Dipartimento di Informatica, Università degli Studi di Torino, 10149 Torino, Italy.*
  E-mail: giorgio.audrito@unito.it
- *F. Terraneo and W. Fornaciari are with Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, 20133 Milano, Italy.*
  E-mail: {federico.terraneo,william.fornaciari}@polimi.it

Manuscript received April 19, 2005; revised August 26, 2015.

the operating system (Miosix) used for the physical validation. Section 3 contains a detailed presentation of FCPP, that is the adopted AP framework, while its integration with the embedded operating system Miosix is described in Section 4. In that section, we discuss the main implementation aspects, related to the use of resources when low end microcontrollers are adopted and the choice of the communication protocol for the validation. Important benefits in terms of abstraction level of the design of a distributed system are pointed out in Section 5, focusing on the implementation of the core network services. Two case studies are presented in Section 6 and 7: the former concerns detection of network vulnerabilities, where a weak point may be prone to disconnection to the rest of the network; the latter considers a problem of contact tracing for the control of disease spreading. For all the considered use cases, both simulation and real data coming from in-field deployment are reported. The final Section 8 draws some conclusions and outlines the future development of our investigation.

2 Background and Related Work

This section provides a brief background on the AP topic, to the benefit of the unfamiliar reader, and reviews the literature for what concerns abstractions for distributed systems programming as well as embedded operating systems.

2.1 Aggregate programming

The design of embedded systems has to address peculiar challenges, especially when the system has to be able to react in a partially autonomous way to its environment. Several approaches have been developed to address these challenges, a comprehensive review of the subject can be found in [6]. Among them, we here focus on *Aggregate Programming* (AP) and its companion abstract language *Field Calculus* (FC) [2]. AP is designed to abstract the overall network as a single *aggregate machine*, which operates on collections of distributed data called *computational fields*.

Through AP, it is possible to define and compose distributed computations without explicit management of low-level aspects such as delivery of messages between devices, or the set of devices themselves. This approach greatly simplifies the management of distributed systems, especially in the common case when unreliability must be handled and multiple nodes share a similar behaviour. For completeness, in cases where the behavior of every actor of the system has to be specified individually to achieve the desired goals, approaches other than AP may be more suitable.

In AP a single program P is defined, which is then periodically run on each device i of a network in rounds consisting of the following steps:

- gathering of the context, including:
  - data collected from local sensors,
  - local information stored in previous rounds,
  - messages received from other devices.
- evaluation of the program P, considering the context above as input;
- as a result of the evaluation, some data is stored locally, other is shared with neighbours and further data may be fed to local actuators.

The execution model of AP also does not require synchronization of rounds across the nodes of the network, thus the global computation/transmission pattern is not deterministic. If required by applications, synchronization can however be built on top of AP, as we will show in Section 5.4. The rounds, executed across time and space by different devices, induce a global behaviour at the system level [12], allowing to interpret the network as a single aggregate machine. The communication of data resulting from a program execution is concretely performed through proximity-based broadcasts, in which a device shares all the relevant data (bundling values produced in various points of the program) with whoever other device may be listening to it. In the program, the received values are modelled and used as *neighbouring fields*, i.e., maps from device identifiers to values of some type. This dedicated data type comes with two operations: *mapping*, which applies a function to each value in a field; and *folding*, which reduces the values in a field to a single value via repeated aggregation with a given function. Direct access to field values is not allowed by design, in order to ensure that programs are neighbour-agnostic, hence more resilient to network changes.

In order to achieve its goal of allowing distributed embedded systems programming, the AP paradigm needs to demonstrate scalability to microcontroller-based embedded systems, a necessary requirement to target low cost, battery operated IoT and sensor network hardware.

Software frameworks for AP have been proposed in the past, such as the Protelis [3] and Scafi [4] languages with the Alchemist [5] simulator. These AP frameworks are based on the Java Virtual Machine (JVM) and currently support only simulated environments. In the paper [13] the memory footprint of Protelis+Alchemist was measured and found to be between 2 to 5 MByte per node, while the available RAM in modern low-power microcontrollers such as the ones used in our work is only 128KByte. While we believe that these frameworks could be optimized, a further obstacle in achieving scalability to microcontroller embedded systems is presented by the JVM itself. Although sensor nodes capable of running Java were made in the past, such as the Sun SPOT [14], they were limited to the J2ME CLDC 1.1 profile which is becoming obsolete, and to the best of our knowledge we are not aware of any JVM supporting modern Java that could fit within the resource requirements of microcontrollers.

In this paper, we overcome this limitation by integrating the FCPP aggregate programming framework [13], a resource efficient framework written in modern C++, with the Miosix embedded operating system, showing how this promising paradigm is suitable for the design of distributed embedded systems even when cost or energy constraints demand the use of low power microcontrollers.

2.2 Embedded operating systems

The operating systems scenario in the embedded systems world is more fragmented compared to desktop and cloud environments. The lower entry barrier to the development of an embedded OS compared to a desktop one, coupled with tighter efficiency and code size requirements, favoured the development of operating systems with widely varying features, each targeting specific application classes.

A prominent class of embedded operating systems are *Real-Time* Operating Systems (RTOS). RTOS have in common a design optimised for predictable latency in time-sensitive operations such as interrupt response and context switches, making them suitable for time-critical applications. Excluding this common trait, there is however still great variability also within this class. On one side of the RTOS spectrum we can find small real-time operating systems focusing on simplicity and reduced code size, such as FreeRTOS [15] and ChibiOS/RT [16], while on the other end we can find RTOSes targeting multicore architectures with full POSIX compliance such as VxWorks [17], QNX [18] and RTAI [19]. Other classes of embedded operating systems exist, targeting specific domains such as Wireless Sensor Networks [20], [21] and IoT [22], [23].

Miosix [24][1] is an RTOS focusing on compliance to POSIX as well as to the C and C++ standard libraries, while at the same time being suitable for resource-constrained microcontrollers. Miosix has been ported to over 40 different hardware targets, with ARM Cortex, RISC-V and ARM7 microcontrollers. It is written almost entirely in C++, and scalability is achieved through a modular design and compile-time configuration, making it possible to exclude certain components, such as the filesystem module, from a build when the feature is not required. The kernel provides multiple schedulers, including a priority scheduler, an Earliest Deadline First (EDF) scheduler and a scheduler based on control theory [25], selectable at compile time.

The userspace module is also a compile-time component that can be disabled to save code size, making it possible to build monolithic firmwares where the applications and the kernel live in the same address space, similar to a unikernel. As a consequence, Miosix provides POSIX compliance also in kernelspace, and the kernel itself takes advantage of the data structures and algorithms provided by the standard C and C++ libraries.

Miosix has been used for the design of networked real-time embedded systems, especially in the research community, where it served as a testbed for the design of clock synchronisation algorithms [26], [27] and real-time wireless mesh protocols [28]. Due to its support for wirelessly connected platforms, as well as its C++ support, it was chosen in this work to host the FCPP aggregate programming framework.

3 FCPP: AN OPTIMIZED AP RUNTIME

FCPP[2] is an implementation of the AP paradigm as a C++ library. This base language has been selected both for its performance, and for its broad support for multiple different architectures. FCPP is portable to any platform providing support for C++14. Thanks to the enhancements we performed as part of this work, the porting procedure has been streamlined to the point of only requiring to write the code to interact with the wireless transceiver provided by the selected platform, as this is the only dependency other than the C++ standard. The FCPP library provides:

- a domain specific language for aggregate programs;
- a component-driven design, easing the extension to multiple application scenarios, including microcontroller deployments, simulations, data processing;
- a compile-time optimised implementation, thanks to meta-programming techniques [29];
- fine-grained parallelism support;
- simulation tools for distributed systems.

Among the possible application scenarios, the one initially supported by FCPP was the simulation of distributed systems running aggregate programs. With respect to alternative tools for the same purpose (Protelis [3] and Scafi [4] languages with the Alchemist [5] simulator), FCPP provides some further simulation features (3D environments, basic physics, probabilistic wireless connection models), while granting a massive increase in efficiency (about 100-fold [13]). Furthermore, the extensible component-based architecture allows to cover additional scenarios that were not previously addressed, including:

1) deployments on distributed systems of microcontrollers, requiring a tiny memory footprint and low computing complexity;
2) graph-based big data processing, requiring efficiency and fine-grained parallelism in order to effectively scale with the available resources.

In the remainder of this paper, we will present a novel implementation of the first of those scenarios, featuring the Miosix operating system. The latter scenario is currently under development, with some preliminary features already available in FCPP [30].

Figure 1 depicts the high-level architecture of the FCPP library, consisting in three main layers:

1) *Data structures*, which are both used for the implementation of aggregate functions in the third layer; and for the specification of components in the second layer either in their internal details or in their external parameter specification.
2) *Components* defining abstractions representing single devices (the *node* class) and the overall network (the *net* class, particularly useful in simulations and data processing). These two classes are produced leveraging template meta-programming by composition of a given list of components [31], each providing a specific functionality, in a *mixin*-like fashion [32], [33]. The component system enables the reusability of several specific functionalities across radically different application scenarios (microcontrollers, simulations, data processing).
3) *Aggregate functions*, which implement the abstract concepts of aggregate programs as templated functions with a `node` parameter. The FCPP library includes both the basic *built-in* functions traditionally included in aggregate languages, as well as a large library of general reusable algorithms (*building blocks*) with proven performance guarantees. Further more specific algorithms may be provided by external libraries, and be used to compose the FCPP-based applications to be run.

Figure 1 (top left) shows the relations between components, displaying whether a component needs another as

1. https://miosix.org
2. https://fcpp.github.io

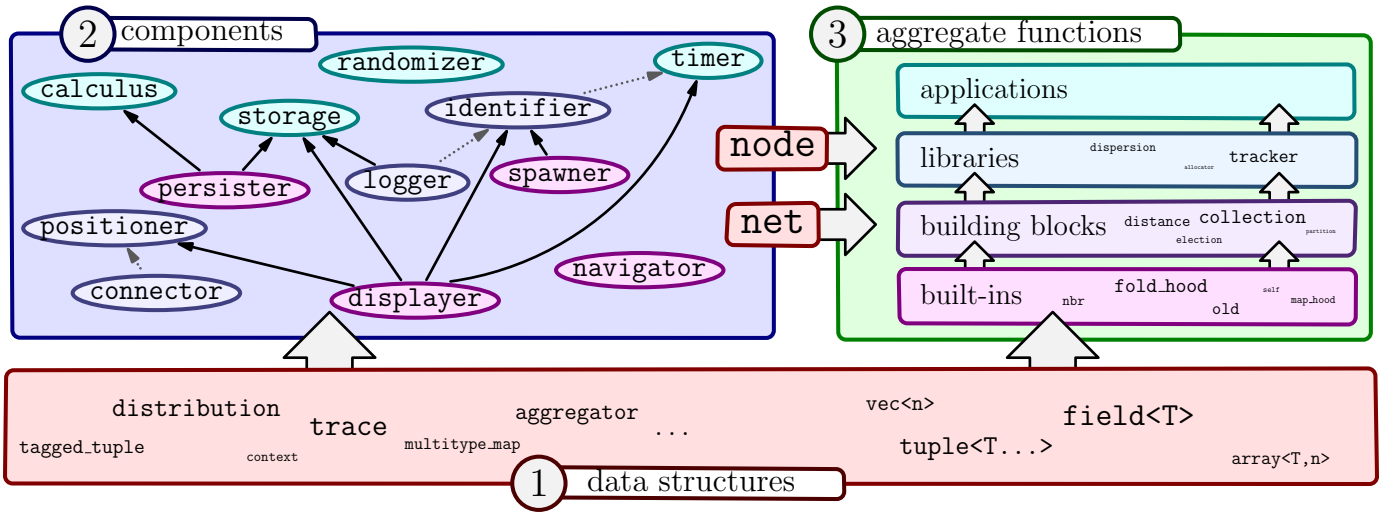

Fig. 1. Representation of the software architecture of FCPP as the combination of three main layers: *data structures* for both other layers, and *components* which provide node and network abstractions to *aggregate functions*. Components are categorized as general purpose (cyan), used across application domains; domain-specific (magenta), relevant only to some domains; and domain-dependent (blue) with variations for different domains. Dependencies between them can be either *hard* (solid), for which the pointed component is required as an ancestor of the other; or *soft* (dotted), for which the requirement only applies to some settings or domains.

TABLE 1
FCPP components.

Component	Provides
`calculus`	allows usage of aggregate programming constructs
`connector`	handles periodic broadcasts of messages
`displayer`	graphical user interface for the whole network
`identifier`	gives access to nodes through their unique identifiers
`logger`	periodically logs summary information
`persister`	stores internal status for recovering after reboot
`positioner`	handles movement and position sensing
`randomizer`	gives pseudo-random generation utilities
`navigator`	manages routing through a bitmap of obstacles
`spawner`	automatically creates nodes in the `identifier`
`storage`	attaches tagged data to nodes or to the net object
`timer`	accesses and regulates scheduling of rounds

aggregate function declaration

F ::= `FUN` t d(`ARGS`, t x$*$) {`CODE` i}
`FUN_EXPORT` d_`t` = `export_type`<$t*$>;

aggregate instructions

i ::= `return` e; | t x = e; i | `for`(`LOOP`(x, ℓ); e; ++x){i} i

aggregate expression

e ::= x | ℓ | t(e$*$) | ue | e o e | p(e$*$) | `node`.c(e$*$) | f(`CALL`, e$*$)
| [&](t x$*$)->t {i} | e ? e : e

type

t ::= T | bt | tt<$t*, \ell*$>

aggregate function

f ::= b | d

built-in aggregate functions

b ::= `old` | `nbr` | `oldnbr` | `spawn` | `self` | `mod_self`
| `map_hood` | `fold_hood` | `mux`

Fig. 2. Syntax of FCPP aggregate functions in *Extended Backus-Naur Form* (EBNF).

ancestor in the composition, as well as highlighting the components that are reused across domains and which instead have variations for different domains. We remark that a component can always be substituted with another, as long as they offer an analogous interface implementing a logically equivalent functionality. A high-level description of the available components is given in Table 1. For porting FCPP to work on real devices, variants of the connector, identifier and logger components had to be written. Those are sufficiently general to be reusable across different boards and operating systems: all the platform-dependent code has been gathered into a simple "networking driver" file.

The syntax of aggregate functions in FCPP is presented in Fig. 2. As FCPP is a C++ library providing an internal domain-specific language, *an aggregate function is a C++ function*, so every peculiar keyword is a C++ macro hiding implementation details, and all features of C++ are available, even though Fig. 2 is restricted to a small subset of the language for compactness of presentation. In the syntax we use the notation $*$ to indicate elements that may be repeated multiple times separated by commas (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 *aggregate instructions* i (within brackets and after keyword `CODE`), followed by the *export description*, listing the types used by the function in message-exchanging constructs.

Aggregate instructions always end with a *return* statement with the function result. Before it, there may be a number of local variable declarations (assigning the result of an expression e to a variable x of type t), and for loops, repeating an instruction i while increasing an integer index x until a condition e is met. *Aggregate expressions* can be either:

- a *variable* identifier x, or a C++ *literal value* ℓ (e.g. a string, integer, floating-point number);
- a *constructor call* t(e$*$) building an object of type t;

- the application of a *unary operator* ue (e.g. $-$, $\sim$, !, etc.), or of a *binary operator* $\mathrm{e\ o\ e}$ (e.g. $+$, $*$, etc.);
- a *pure function call* $\mathrm{p(e*)}$, where p is a C++ function not using FCPP and the node object;
- a *component function call* $\mathtt{node}.\mathrm{c(e*)}$, where c is a method provided by some component;
- an *aggregate function call* $\mathrm{f}(\mathtt{CALL}, \mathrm{e*})$, where f is either a defined aggregate function name d or an aggregate built-in function b (detailed below);
- a *conditional branching* expression $\mathrm{e_{guard}}$? $\mathrm{e_\top}$: $\mathrm{e_\bot}$, which evaluates and returns $\mathrm{e_\top}$ if $\mathrm{e_{guard}}$ is `true`, evaluating and returning $\mathrm{e_\bot}$ otherwise.

Several built-in aggregate functions are provided in FCPP. Among them, in this paper we will focus on the following:

- `nbr(CALL,i,f)`, that repeatedly updates a distributed value based on neighbours' values. It takes an initialisation value `i` of a type *t*, and a function `f` with a single argument of type `field<`*t*`>`. At every evaluation round, function `f` is applied to the collection of values that neighbours recently shared for this same `nbr` expression, using `i` at computation start when no values have been shared yet. The result of `f` is both shared back to neighbours for their future rounds, and returned by the `nbr` call.
- `max_hood(CALL,v,i)` given a neighbouring field `v` (e.g., the argument passed to `f` by an `nbr`) and a local value `i`, reduces the elements in the field to a single value by starting from `i` and repeatedly applying function `max` to each element of the collection and to the current partial result. Similarly, function `min_hood` applies function `min`; and function `list_hood` appends each element in the field to an initial collection. Note that these folding operators need a neighbouring value to operate with, which is usually obtained by sharing values with neighbours through `nbr`.

Among the macros used by FCPP, we remark that `CALL`, `CODE`, and `ARGS` ensure that the aggregate context (in the `node` object) is carried along during program execution, while updating a representation of the stack trace for internal *alignment*. Thanks to the alignment mechanism, the messages (implicitly) produced by a `nbr` construct are matched in future rounds (on the same or different devices) only to the *same* construct, i.e., an `nbr` called in the same stack trace and position in the syntax. This enables function composition, and recursion, without risk of interferences between messages originating in different parts of the program.

4 Integrating FCPP with Miosix

This section discusses the integration of FCPP with the Miosix operating system and WandStem WSN node. The integration and the experiments described in the next sections are all publicly available online.[3]

The Miosix operating system was chosen as it simplifies the porting of FCPP to embedded targets. This is because Miosix provides an execution environment with a high degree of standard compliance, especially for what concerns support for embedded applications written in C++, both in kernelspace and userspace. As previously discussed, the userspace subsystem in Miosix is optional, and for the FCPP integration it was decided to disable it, and run FCPP as a kernelspace application. The FCPP aggregate computing library has been designed to be easily ported to any execution environment providing a C++ compiler adhering to the C++14 standard, a requirement that Miosix meets. Miosix also natively provides access to the system time through `<chrono>`, so no modifications to FCPP were needed on the time *Application Programming Interface* (API) access.

Fig. 3. Packet format used to implement the FCPP communication model on WandStem nodes. The size in bytes of every packet field is reported, except for the FCPP message field that is variable length.

The hardware platform has been selected in order to provide a sufficient amount of RAM memory to meet the memory footprint of FCPP and Miosix combined, also leaving available room to allow developing applications. Additionally, among the platforms supported by Miosix, one was chosen providing a wireless transceiver for inter-node communication. Given the constraints above, the chosen platform is the WandStem WSN node [34], a wireless sensing board providing a 48MHz ARM Cortex-M3 CPU, 128KByte of static RAM memory, 1MByte of FLASH memory and an IEEE 802.15.4-compliant wireless transceiver.

The integration of FCPP and Miosix required to define a suitable wireless communication protocol for FCPP, as well as to design and implement two software modules: an interface between the FCPP message API and the wireless transceiver provided by Miosix, and a compressed logging module to simplify the collection of experiment results.

4.1 Wireless communication protocol

Instances of FCPP running on different nodes communicate through messages, an abstract data type encapsulating the shared state of AP programs. FCPP messages need thus to be mapped into wireless packets, by defining an appropriate communication protocol. In our integration we took advantage of the shared nature of the wireless medium natively providing the gossip-like communication model required by AP. Our communication protocol consists of a single packet type, shown in Figure 3. The packet is composed of a standard 802.15.4 [35] preamble, Start Frame Delimiter (SFD), length field (L) and Personal Area Network (PAN) header, followed by the node sender ID, a data block containing the encapsulated FCPP message, a send time delay field (D) and a Cyclic Redundancy Check (CRC) to detect corrupted packets. The PAN header in particular serves the purpose of identifying the network through an unique PAN ID, allowing for co-existence with other networks such as ZigBee, as well as to allow multiple independent FCPP networks to co-exist in the same area. While the PAN ID is a configurable parameter, the microcontroller of WandStem nodes provides a unique hardware ID that has been used as sender node ID, thereby requiring no per node configuration

3. https://github.com/fcpp/fcpp-miosix

and allowing the same firmware to be programmed on all the nodes in a network, simplifying deployment.

Packets do not have a destination field, as are instead implicitly addressed to all nodes in the same FCPP network within the radio range of the sender. Packets are transmitted using a Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA) scheme with exponential backoff to effectively share the wireless medium. As FCPP messages contain timestamps, coarse-grain MAC-level timestamping [36] is used to improve accuracy in case of packet collisions. This feature is implemented through the field D of the packet which is added by the MAC to encode the time delta between the upper layers expected send time, and the actual send time, making it possible to adjust timestamps accordingly upon message reception. Due to the point-to-multipoint communication model, the protocol does not include acknowledge packets, and relies instead on the capability of the AP communication model to tolerate packet losses.

As can be seen, the mapping of FCPP messages to IEEE 802.15.4 radio packets is simple and straightforward. We consider this a significant advantage of the AP model, proving it naturally lends itself to be applied to wireless networks.

4.2 Experiment logging

Analysis of AP simulations and experiments is performed through logged data that each FCPP instance periodically prints. The logged data consists mainly in the local state of each node, and to best reconstruct the distributed state it is thus necessary to save the logs of each node. While for a simulation environment this is not an issue as all virtual nodes are simulated on a single desktop machine, actual experiments require a data collection solution. For our deployments, we relied on the availability of free RAM memory in the chosen hardware to implement delta-compressed in-memory storage of logged data, which allow the data of each node to be downloaded to a central computer when the nodes are collected after each experiment.

This module has been added to the standard FCPP codebase, and operates by allocating a suitable buffer (40KByte in our case) at the start of the experiment, used to store a compressed copy of all the printed messages. In our experiments the buffer was never filled, and actually never exceeded 1/4 of its capacity, so we were always able to fully reconstruct the entire network state.

4.3 Resource utilisation

The resources required by FCPP and Miosix are difficult to quantify, as the Miosix compilation process uses link-time garbage collection, that is during compilation a static call graph of all functions and class methods is computed and all unreachable code is removed from the build. The part of the operating system, C/C++ standard libraries and FCPP that become part of a firmware thus depend on what components are actually used by the application.

To estimate the required resources in a realistic setting we compiled a firmware including both the vulnerability detection and contact tracing applications (described in Section 6 and 7) together. Table 2 reports the measured results.

TABLE 2
Memory occupation breakdown of the FCPP/Miosix integration.

FLASH memory occupation	
Applications + FCPP	145029 Byte
Miosix (OS kernel)	38325 Byte
libgcc.a (Compiler intrinsics)	9268 Byte
libm.a (C standard library)	320 Byte
libc.a (C standard library)	66834 Byte
libstdc++.a (C++ standard library)	120770 Byte
Total	380564 Byte

RAM memory occupation	
.data (Global/static variables)	1360 Byte
.bss (Global/static variables)	3512 Byte
main stack	8952 Byte
heap	25336 Byte
logging buffer	40960 Byte
Total (no logging buffer)	39160 Byte
Total (with logging buffer)	80120 Byte

Microcontrollers are computer architectures that execute code directly from the on-board FLASH memory, so the code of the OS kernel, standard libraries, FCPP and application all contribute to the FLASH memory requirements. We used static analysis to compute a breakdown of the used memory, although being FCPP a header-only C++ library, the code size for the applications and FCPP could not be separated. From the table we can see that the applications and FCPP occupy the largest fraction of code size (38%), the second largest fraction is occupied by the C++ standard library (32%), while the Miosix kernel occupies 10%. Other standard libraries take up the remaining part of the firmware. Variables and data structures used by the application, libraries and kernel contribute instead to the RAM memory requirements. The RAM information in Table 2 are obtained through a mix of static analysis and profiling, as for example the heap allocations are only known at runtime and may differ from node to node. For profiling we used AP to measure the maximum occupied stack and heap among all nodes in the network. From the table we can see that most of the occupied RAM is dynamically allocated on the heap. The single largest data structure used is the buffer for the compressed logging, that is shown separately as it is only used to simplify the experiment data collection and would not be needed in an actual deployment. During the experiment also the maximum transmitted packet size has been profiled, measuring 87 Bytes of FCPP message content, 105 Bytes including all headers.

Summarising, the resource utilised by FCPP are well within the capabilities of modern microcontrollers. The presented applications, even compiled together occupy less than half of the WandStem available memory, leaving ample room for significantly more complex application development. Alternatively, a smaller microcontroller with about half the memory could be selected to save cost.

5 Core Network Services

A key advantage of AP is its ability to raise the abstraction level at which distributed embedded systems are programmed, by shifting the focus from individual nodes and

their interaction to the network level. As a result, the expressiveness of AP makes it possible to provide concise yet efficient solutions to common needs that arise in distributed systems programming. In this section, we will present some such examples that we used in our case studies, with a focus on core network services that are often needed as a basis to build complex distributed behaviours.

5.1 Network statistics collection

The first and simplest AP example we show is the `gossip_max` function, that propagates across the network the maximum between a local input data and the data "gossiped" by neighbours. It can be implemented in FCPP in the following way.

```
GEN(T) T gossip_max(ARGS, T value) { CODE
  return nbr(CALL, value, [&](field<T> x){
    return max(max_hood(CALL, x), value);
  });
}
```

This function is generic, with a type parameter `T` and a single argument `value` of that type. It also consists of a single `nbr`, which starts from a default corresponding to the local value, and computes a new gossip based from the gossips received from neighbours' in `x`. The block starting with `[&]` is the standard C++ syntax to introduce a lambda function. Such gossip is defined as the maximum between the current `value` and every gossip in `x`. The same principle can be applied to define the `gossip_min` function, with a trivially similar implementation.

These functions allow a network to converge to the maximum/minimum of a given quantity, and are particularly useful for the collection of network statistics. As an example, in our deployments of FCPP we took advantage of this feature to monitor the maximum dynamically allocated heap memory used among all nodes, the maximum occupied stack size as well as the maximum transmitted message size by simply adding the following lines to our programs:

```
node.storage(max_stack{}) = gossip_max(CALL, usedStack());
node.storage(max_heap{}) = gossip_max(CALL, usedHeap());
node.storage(max_msg{}) = gossip_max(CALL, node.msg_size());
```

where the functions `usedStack()` and `usedHeap()` query the Miosix OS for the currently occupied memory, while `node.msg_size()` is provided by FCPP.

5.2 Leader election and distance from leader

Leader election is a common problem for the coordination of distributed systems. It may be tempting to use a `gossip_max` call to select a leader, but that would not adjust in case the network changes and the former leader is lost. Optimally self-adjusting solutions to the leader election problem have been proposed which do not rely on a-priori network information [37], also available in the FCPP coordination library as `wave_election`. For this scenario, however, we assumed knowledge of a reasonably accurate upper bound to the network diameter. This allowed us to use the simpler FCPP library function `diameter_election_distance` function, which also computes the distance (in terms of network hops) between each node and the leader, implemented with code equivalent to the following:

```
using tuple_t = tuple<device_t,hops_t>;
FUN tuple_t diameter_election_distance(ARGS, hops_t diameter) {
  CODE
  tuple_t loc(node.uid, 0);
  return nbr(CALL, loc, [&](field<tuple_t> x){
    x = mux(get<1>(x) < diameter, x, loc);
    tuple_t best = min_hood(CALL, x, loc);
    if (best != loc) get<1>(best)++;
    return best;
  });
}
```

In this function, first a local leader `loc` is defined, as the current device with distance zero. Then a leader estimate is evolved every round with `nbr`, by starting from the local leader, and then selecting the best plausible leader offer from neighbours. First, through multiplexer operator `mux` (a strict version of an if, which also applies to field values pointwise), only offers from neighbours that are below the network diameter in distance (and hence plausible) are selected, using `loc` in place of invalid offers. This ensures that in case a leader is disconnected, in a time proportional to `diameter` it will be discarded from the network allowing election of new valid leaders. Then, the best leader offer still standing in `x` is selected, by lexicographic minimisation (on node ID first, then distance). Finally, the distance estimate is increased by one if the leader is not the node itself. In this code `get<1>` is the standard C++ syntax to access the second element of a tuple. This function was used in our application through this simple line of code where `tie` is the standard C++ syntax to unpack a tuple to individual variables:

```
tie(node.storage(min_uid{}), node.storage(hop_dist{})) =
    diameter_election_distance(CALL, diameter);
```

5.3 Neighbour list and network topology discovery

Wirelessly connected distributed systems are often deployed without a statically planned network topology, due to the uncertainty caused by obstacles in the environment blocking or attenuating the radio signal as well as due to the presence of mobile nodes. For this reason, computing at runtime the network topology, as well as keeping it updated when the environment changes, is a common requirement.

While in all examples so far the distributed values used always were of scalar type, FCPP also supports array or list types, making it possible to gather neighbour lists for logging or possibly disseminating them across the network. We logged such information by adding to the node storage a value `nbr_list` declared as `std::vector<device_t>`. The list of neighbours can obtained through the simple code:

```
node.storage(nbr_list{}).clear();
list_hood(CALL,node.storage(nbr_list{}),node.nbr_uid(),nothing);
```

Which at every iteration clears the previous neighbour list and re-computes it starting from `nothing` and adding all of the neighbours' identifiers listed in `node.nbr_uid()`, thanks to the basic library function `list_hood`.

5.4 Coarse-grain clock synchronisation

Clock synchronisation is another core network service that is essential for the coordination of distributed systems. While

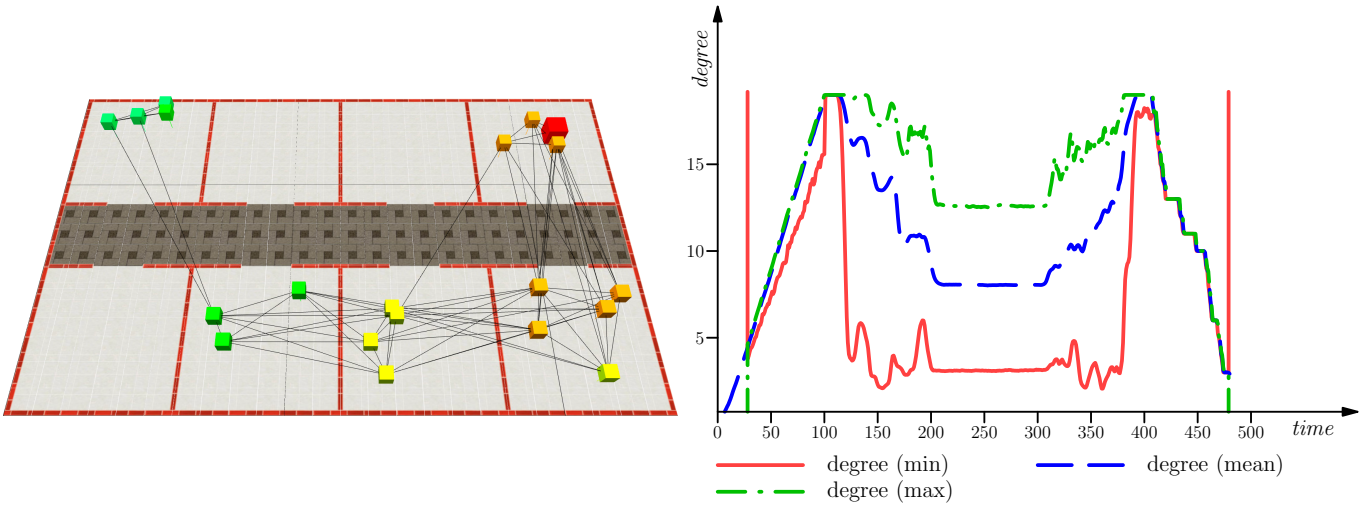

Fig. 4. Screenshot of the simulation (left) and evolution of the minimum, average, and maximum degree in the network as simulated time progresses (right). In the screenshot, nodes are coloured according to their hop-distance from the leader (the larger red cube).

achieving high resolution clock synchronisation is outside the scope of this paper, we show here a simple coarse-grain clock synchronisation implementation that focuses on demonstrating the expressiveness of AP, yet is already sufficiently precise to be used in our experiments to solve the problem of synchronising the collected experiment traces among nodes that were turned on at different points in time.

Our simplified clock synchronisation maintains a network-wide global clock computed through the FCPP library function `shared_clock`, implemented with code equivalent to the following:

```
FUN times_t shared_clock(ARGS) { CODE
  return nbr(CALL, times_t{0}, [&](field<times_t> x){
    return max_hood(CALL, node.previous_time() == TIME_MIN
             ? node.current_time() : x + node.nbr_lag());
  });
}
```

The function consist of a single `nbr` operator, which starts from a default clock of zero, and computes a new clock from the received neighbours' clocks in `x`. Each time, the shared clock is defined as the maximum of the sum of the clocks in `x` with the time elapsed since the last message from the corresponding devices in `node.nbr_lag()`, except for the first round (when the previous time is `TIME_MIN`) for which the shared clock is set to coincide with the local clock. Application code can then add a global clock to the node storage through the following line.

```
node.storage(global_clock{}) = shared_clock(CALL);
```

5.5 Simulation

Among the described network services, we present here results for the leader election, distance from leader as well as neighbour discovery in a mobile node scenario tested both in simulation and in a real world deployment.

The simulation is organised as follows. We considered an indoor environment consisting of eight $6 \times 6m$ rooms at the two sides of a long 24×3 corridor, as depicted in Figure 4 (left). We deployed 20 devices, which are turned on one at a time in the simulated interval of time from $0s$ to $100s$, all located in a same room. From time $100s$ to $200s$, the devices move to different rooms (each at different times), staying there still until starting to gather back to a single room, in the interval from time $300s$ to $400s$. Finally, the devices turn off one at a time between times $400s$ and $500s$. We performed 1000 runs with different random seeds, averaging the results across them. We assumed that devices had a 0% probability of connection from a distance of 12 meters, a 50% probability of connection from a distance of 8.4 meters, and 100% probability of connection from zero distance, interpolating between these numbers through a smooth step-like function derived experimentally. We scheduled rounds periodically every second, collecting logs accordingly, and allowing messages to persist for 5 seconds across multiple rounds before being dropped, in order to stabilise the neighbour list.

The left side of Figure 4 shows the state of the network leader election and distance from the leader for one such simulation after the nodes were deployed in the rooms. The network was in this case fully connected, and all nodes agreed on the same leader, being the node with the lowest network ID, and the distance computation was also correct.

For what concerns the neighbour discovery, Figure 4 (right) shows instead the evolution of the minimum, average, and maximum degree in the network as nodes are first progressively turned on, then deployed in different rooms, collected and progressively turned off. The figure has been computed by averaging the results of all 1000 simulations.

For what concerns clock synchronisation, the global clock was simulated, but it was always fundamentally exact as expected, since the possible sources of error for it were not modelled.

5.6 Deployment

The same FCPP code that was run in the simulation was also compiled together with Miosix and deployed on eight WandStem nodes. The experiment set-up was similar to the simulations, where the nodes were first turned on one by one over the course of $155s$, then deployed in different rooms from time $210s$ to $420s$. After the deployment, nodes

were left in a fixed position until time $1020s$, then were brought back in the same room, an operation that lasted until time $1180s$. Finally, the nodes were turned off one by one from time $1200s$ to $1740s$. The simulation and deployment setups were designed to be qualitatively similar, aiming to prove that the gathered results show similar patterns. We did not aim at a quantitative comparison, as it would be impossible to perform due to both the unavoidable uncertainties in real-world experiments (interference with existing WiFi networks, etc.), and the lack of radio modelling in AP simulators (interferences, walls, etc.).

The left part of Figure 5 shows the network topology obtained through the neighbour discovery information during the time frame from $420s$ to $1020s$. Links that were present more than 50% of the time during the observed timeframe are shown with a solid line, while links that were present less than 50% of the time are shown with a dashed line. During the time when the nodes were in a fixed position all the link reliabilities, except for the one between nodes 0 and 6, were either above 95% or less than 10%. This is to be expected in a static environment where links with good signal reception only experience occasional packet losses due to collisions, resulting in high link reliabilities, while links where packets are received at the limit of the receiver sensitivity result in low link reliabilities. The link between nodes 0 and 6 is the only link exhibiting an asymmetric reliability, where node 6 received packets from node 0 with 97% reliability, while node 0 received packets from node 6 with only 55% reliability. Such occurrences are possible in CSMA/CA networks, and can be attributed to a hidden terminal situation, where packets from node 6 could have been stomped by other nodes such as 1, 5 and 7, not in the radio range of 6, and thus unaware of its transmissions. Node 2 remained instead disconnected from the rest of the network 99.3% of the time and only occasionally received packets form node 7.

The right part of Figure 5 shows instead the average and maximum degree of the network during the entire experiment, including the gradual powering on of the nodes in a single room, their deployment and subsequent recollection. The pattern matches the one observed in the simulations.

The left part of Figure 5 also shows the leader election and hop distance computation during the time frame from $420s$ to $1020s$. Nodes 0, 1, 3, 4, 7 consistently selected node 0 as their leader, in accordance with having a 100% link reliability towards 0. Node 6, having a 97% link reliability towards 0 sporadically disconnected and elected itself as leader, and node 2, being isolated most of the time select itself as leader except for the brief connection periods where it correctly considered 0 the leader. For what concerns the hop distance computations, nodes 0,1,5,7 stably reported the correct hop distance. Nodes 3 and 4 due to their weak links transiently changed their hop distance from 2 to 1, while nodes 6 and 2, when isolated from the rest of the network reduced their hop number to 0 as they became leaders of the isolated part of the network. All in all, the neighbour discovery, leader election and hop distance computation operated as expected also in a real world deployment exhibiting non-trivial connectivity patterns such as asymmetric link reliabilities.

For what concerns the global clock, the experiment did not log the clock synchronisation error and the current clock resolution is only $7.8125ms$ which is insufficient to appreciate clock skew over the course of the experiment, but the global clock nonetheless was used as time reference for aligning the logs for producing Figure 5 (left), showing it achieved the goal of compensating the time offset of nodes turned on at different points in time. Further studies aiming to achieve high-resolution clock synchronisation in an AP framework will be presented in future works.

6 Case Study: Vulnerability Detection

As a first case study, we consider a simple scenario of network vulnerability detection, using archetypal routines common in aggregate programs. Assume that we want to keep track of whether our network contains *weak points*, which are devices that are only connected to a single other device. Such situations may be of concern, as a weak point can easily be disconnected from the rest of the network, permanently or temporarily, due to communication malfunctions. This routine can be programmed in FCPP through this very simple function.

```
FUN void vulnerability_detection(ARGS, int diameter) { CODE
    tie(node.storage(min_uid{}), node.storage(hop_dist{}))
      = diameter_election_distance(CALL, diameter);
    bool collect_weak = sp_collection(CALL,
      node.storage(hop_dist{}), count_hood(CALL) <= 2,
      false, [&](bool x, bool y) {
        return x or y;
      });
    node.storage(some_weak{})
      = broadcast(CALL, node.storage(hop_dist{}), collect_weak);
}
```

This function takes a single parameter, which is an upper bound to the diameter of the network. In the first instruction, a *leader election* algorithm is called, using this estimate to select the node with the minimum identifier (UID) and the hop-distance of the current device from it. In the second instruction, a *collection* algorithm is called, which aggregates towards the leader the Booleans representing whether the current node is a weak point (if the `count_hood` count of neighbours is at most two, including the current device), using the `or` operation with null element `false`. Finally, in the third instruction a *broadcast* algorithm is called, propagating the collection result to the whole network.

6.1 Simulation

We evaluated this case study in the same scenario considered in Section 5.5. Figure 6 (left) shows the results, averaged across the whole 1000 runs. As the network turned out to be sufficiently dense, weak points were produced only rarely (red line), except for the initial turn-on phase (before $100s$) where weak points were more likely. After that, peaks of about 1% frequency occurred during the dispersal phase, and smaller peaks occurred during the gathering phase (due to the nodes yet-to-be gathered possibly becoming weak points of the network). In both cases, the whole network was able to track the presence of such weak points (blue line), with a small delay: the wider amplitude of the blue line represents the fact that many devices agree in knowing that some device is a weak point.

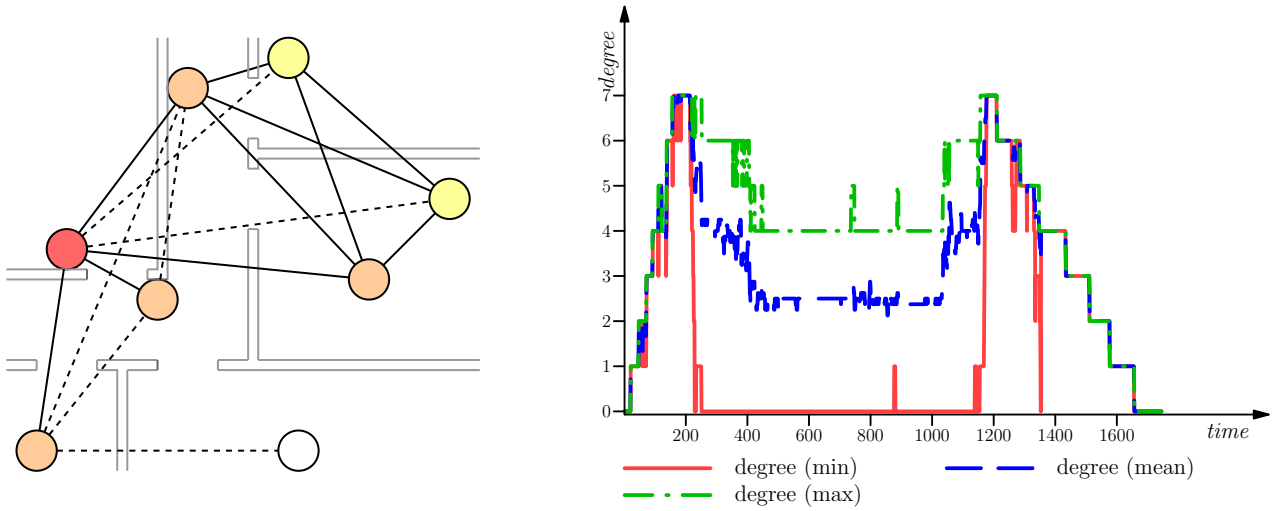

Fig. 5. Deployment plan (left) and average degree over time during the experiment (right). The network connectivity graph was computed using AP.

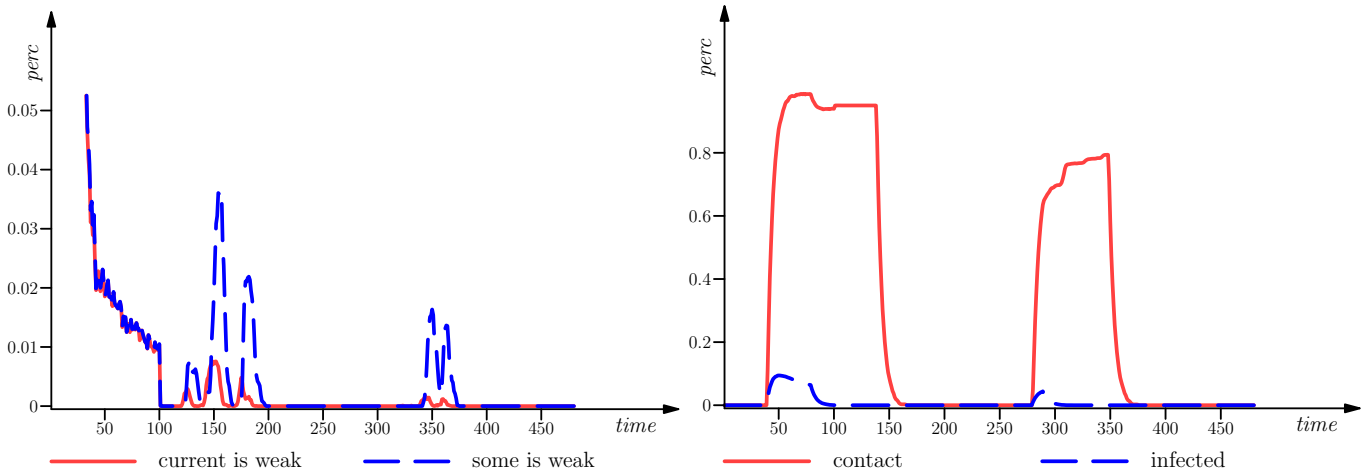

Fig. 6. Results of the simulated case studies, vulnerability detection (left) and contact tracing (right)

6.2 Deployment

The same FCPP program was also compiled for Miosix and deployed on the eight WandStem nodes. The deployment pattern matches the one described in Section 5.6. Figure 7 (left) shows the results. The network initially correctly reported the presence of nodes with less than two links before time $45s$, corresponding to when the third node was turned on. After a transient time to propagate the information, it stably reported that no weak points were present as long as the nodes were in the same room. After the nodes were transferred in different rooms, due to the poor connectivity of nodes 2 and 7, nodes agreed most of the time that the network contained weak points. When the nodes were recollected in a single room, the lack of weak points was correctly detected. Finally, when turning off nodes, when fewer than 3 nodes remained turned on, the weak point was detected again.

7 Case Study: Contact Tracing

As a final case study, we considered a prototype contact tracing application for the control of disease spreading. We let every node keep a list of its most recent *contacts*, as pairs of UIDs and timestamps: as this list may grow quite large, and reveal sensitive information, every device kept its own list locally without sharing with others. In order to file contact warnings, a list of the most recent reported *infections* was also kept, also as pairs of UIDs and timestamps. As this second list was potentially smaller, devices shared knowledge about infected people, each of them comparing it against its personal list of contacts. In case of a match between a contact and a reported infection, the contact warning was triggered. We set up a fixed window of $60s$ before discarding both contacts and reported infections.

7.1 Simulation

We evaluated this case study in the same scenario considered in Section 5.5, assuming that a single node becomes infected at time $40s$ and then again at time $280s$. Figure 6 (left) shows the results, averaged across the whole 1000 runs. In the first infection, the network is confined to a single room, thus being strongly connected: as expected, every device follows filing a contact warning for the following $60s$. In the second infection, the network is already dispersed across the whole floor, and that translates into a smaller peak

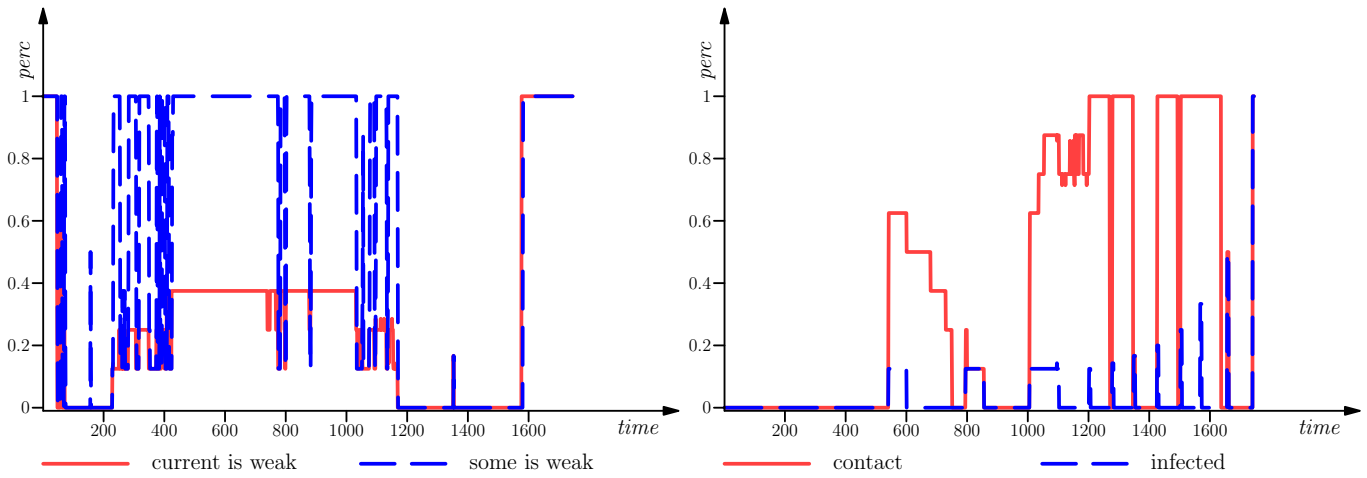

Fig. 7. Results of the case studies during the experiment, vulnerability detection (left) and contact tracing (right)

as some devices are no longer connected to the reported infection.

7.2 Deployment

Porting of the contact tracing application to Miosix required to map the infection event to a short press of the only user button of the WandStem board (a long press of the same button was instead used to turn off the node), and the infection notification was displayed by turning on a red LED. The so modified FCPP program was compiled for Miosix and deployed on the eight WandStem nodes, following the same pattern described in Section 5.6. Figure 7 (right) shows the results.

Node 1 signalled an infection from time $540s$ to $600s$ and correspondingly, nodes 0, 3, 4, 5 reported an infection too due to their direct contact, resulting in 62.5% nodes being infected. Later, node 6 signalled an infection from time $795s$ to $855s$. In this case, only node 7 reported an infection due to direct contact. Node 1 then signalled an infection again from time $1005s$ to $1100s$, but in this case around time $1020s$ nodes began to be recollected in the same room. As a result, while the percentage of infected devices started at the same value as the previous time node 1 signalled an infection, it quickly rose to 87.5%. The 100% positivity rate starting from time $1200s$ is due to nodes being turned off. As the only button in WandStem nodes is used both for turning off nodes and signalling an infection, the long press required to turn off the node resulted in nodes transmitting an infected packet just before turning off, and being all in the same room resulted in a 100% positivity rate almost all the time in the last part of the experiment.

8 Conclusions

To the best of our knowledge, this paper is the first attempt to link the aggregated computing paradigm (FCPP) with the most important requirements of distributed systems, by considering not only a validation by simulation but also targeting real world microcontroller-based sensor nodes.

The investigation here presented focused on demonstrating that the approach based on aggregated computing is feasible and provides important features of distributed systems with limited programming effort and almost no knowledge of the hardware/software specific characteristics of the wireless node, with the expected portability benefits.

Moreover, the presented design flow can be adopted also in systems with a fine granularity of the processing platforms, down to the level of microcontrollers.

The validation has been carried out by simulation, as usual, to assess the scalability; but also using as test vehicles real hardware nodes mounting an open-source operating system for embedded applications (Miosix), which also allows the replication of our experiments by the research community. Note that the validation on real devices allowed us to check the performance of the proposed solution under non-ideal radio links, including for instance packet losses, while also assessing the feasibility in terms of usage of scarce resources like the memory and the computing power of an average microcontroller.

The real-world assessment has been carried out in a domestic environment using eight nodes, and the scalability of the approach has been demonstrated via simulation. Future work may scale up the size of the physical validation, using more wireless nodes, and may include high-resolution clock synchronization features and more advanced communication protocols to further improve the network reliability and capabilities.

Acknowledgments

The authors would like to thank Ferruccio Damiani, Gianluca Torta and Enrico Bini for their valuable suggestions and fruitful discussions on the application of aggregate programming to embedded systems.

References

[1] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the internet of things," *IEEE Computer*, vol. 48, no. 9, pp. 22–30, 2015.

[2] G. Audrito, M. Viroli, F. Damiani, D. Pianini, and J. Beal, "A higher-order calculus of computational fields," *ACM ACM Transactions on Computational Logic*, vol. 20, no. 1, pp. 5:1–5:55, 2019.

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

[4] 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)*, ser. Annals of Computer Science and Information Systems, vol. 8. IEEE, 2016, pp. 1495–1504.

[5] D. Pianini, S. Montagna, and M. Viroli, "Chemical-oriented simulation of computational systems with ALCHEMIST," *Journal of Simulation*, vol. 7, no. 3, pp. 202–215, 2013.

[6] 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*. IGI Global, 2013, pp. 436–501.

[7] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: The TOTA approach," *ACM Transactions on Software Engineering Methodologies*, vol. 18, no. 4, pp. 15:1–15:56, 2009.

[8] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] A. Kondacs, "Biologically-inspired self-assembly of two-dimensional shapes using global-to-local compilation," in *18th International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, 2003, pp. 633–638.

[10] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco, "Mobile data collection in sensor networks: The TinyLime," *Pervasive and Mobile Computing*, vol. 1, no. 4, pp. 446–469, 2005.

[11] C. Lasser, J. Massar, J. Miney, and L. Dayton, *Starlisp Reference Manual*. Thinking Machines Corporation, 1988.

[12] 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*, vol. 28, no. 2, pp. 16:1–16:28, 2018.

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

[14] J. A. G. Reyes, R. S. Robles, B. E. S. Recéndez, and J. G. A. Olague, "Implementation of a timestamping service for sunspot sensors," *Procedia Technology*, vol. 7, pp. 4–10, 2013, 3rd Iberoamerican Conference on Electronics Engineering and Computer Science, CIIECC 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212017313000029

[15] "FreeRTOS project page." [Online]. Available: https://www.freertos.org

[16] "ChibiOS project page." [Online]. Available: https://www.chibios.org

[17] "VxWorks project page." [Online]. Available: https://www.windriver.com/products/vxworks

[18] "QNX project page." [Online]. Available: https://blackberry.qnx.com

[19] "RTAI project page." [Online]. Available: https://www.rtai.org

[20] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," vol. 35, no. 11, p. 93–104, nov 2000. [Online]. Available: https://doi.org/10.1145/356989.356998

[21] 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*, 2004, pp. 455–462.

[22] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Riot: An open source operating system for low-end embedded devices in the iot," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[23] "Zephyr project page." [Online]. Available: https://www.zephyrproject.org

[24] "Miosix project page." [Online]. Available: https://www.miosix.org

[25] Martina Maggio, Federico Terraneo, Alberto Leva, "Task scheduling: a control-theoretical viewpoint for a general and flexible solution," *ACM Transactions on Embedded Computing Systems (TECS)*, 2014.

[26] Federico Terraneo, Fabiano Riccardi, Alberto Leva, "Jitter-compensated VHT and its application to WSN clock synchronization," in *IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[27] Federico Terraneo, Alberto Leva, Silvano Seva, Martina Maggio, Alessandro Vittorio Papadopoulos, "Reverse Flooding: exploiting radio interference for efficient propagation delay compensation in WSN clock synchronization," in *IEEE Real-Time Systems Symposium (RTSS)*, 2015.

[28] Federico Terraneo, Paolo Polidori, Alberto Leva, William Fornaciari, "TDMH-MAC: Real-Time and Multi-hop in the Same Wireless MAC," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018.

[29] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[30] G. Audrito, F. Damiani, and G. Torta, "Bringing aggregate programming towards the cloud," in *ISoLA*, ser. Lecture Notes in Computer Science. Springer, 2022, to appear.

[31] M. D. McIlroy, J. Buxton, P. Naur, and B. Randell, "Mass-produced software components," in *1st international conference on software engineering*, 1968, pp. 88–98.

[32] H. I. Cannon, "Flavors: A non-hierarchical approach to object-oriented programming," Artificial Intelligence Laboratory, MIT, USA, Tech. Rep., 1979.

[33] G. Bracha and W. R. Cook, "Mixin-based inheritance," in *International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) / European Conference on Object-Oriented Programming (ECOOP)*. ACM, 1990, pp. 303–311.

[34] F. Terraneo, A. Leva, and W. Fornaciari, "Demo: A High-Performance, Energy-Efficient Node for a Wide Range of WSN Applications," in *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN '16, 2016, p. 241–242.

[35] "IEEE 802.15.4-2020 Standard for Low-Rate Wireless Networks." [Online]. Available: https://standards.ieee.org/ieee/802.15.4/7029/

[36] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi, "The flooding time synchronization protocol," in *Conference On Embedded Networked Sensor Systems*, 2004.

[37] Y. Mo, G. Audrito, S. Dasgupta, and J. Beal, "Near-optimal knowledge-free resilient leader election," *Automatica*, 2022, to appear.