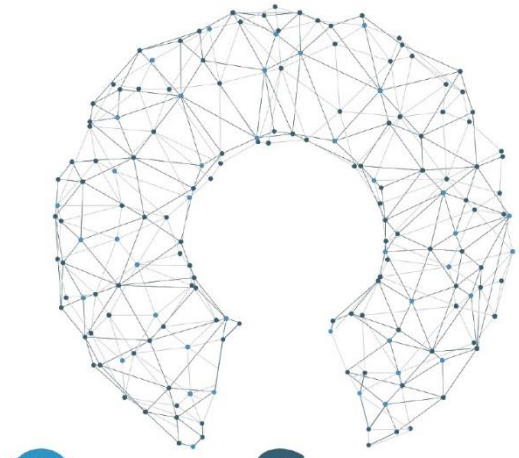


OpenSwarm

www.openswarm.eu

Call: HORIZON-CL4-2022-DATA-01

Type of action: RIA

Grant agreement: 101093046

Deliverable 1.2: Architecture, first version

Work Package 1: Requirements and System Architecture

Task Lead: INRIA

WP Lead: INRIA

This project has received funding from the European Union's Horizon Europe Framework Programme under Grant Agreement No. 101093046.

Funded by the European Union

Document information	
Author(s)	Thomas Watteyne
Reviewers	Micael Couceiro
Date	30-Sep-2023
Type	DEC
Dissemination level	PU

Document history			
Date	Version	Author(s)	Comments
30-Sep-2023	1	Thomas Watteyne	Initial version

Table of contents

Introduction

This deliverable contains the proposed initial architecture for the OpenSwarm project.

We define “architecture” as mainly three things.

First, the architecture provides a high-level description of the different blocks of the technical output of the project. That is, the scope of the architecture is general, and equally encompasses both hardware elements (e.g. the motors of the DotBot testbed platform) and software elements (e.g. the implementation of the EDHOC security block).

Second, the architecture describes how these blocks interact. That is, it describes what each block provides in terms of service to the other blocks, as well as what input it expects and what output it generates. We use the term Application Programming Interface (API), a term borrowed from software development, to refer to these inputs and output. This document hence serves as a guide for defining the API for the different blocks.

Third, this document serves as a guiding document throughout the project. It is the result of discussions between the different partners, who are each responsible for a number of blocks. This document helps the partners identify the scope of the elements they are responsible for, and how these interact with other elements.

Defining and agreeing upon a clear architecture before starting any implementation is essential for efficient collaboration. This proactive approach prevents discrepancies during the integration phase and facilitates a seamless development process throughout the project.

The contents of this document are expected to evolve throughout the project. Such changes are typical and anticipated in any development endeavour, as the scope of the elements and their interactactions will inevitably shift. Nevertheless, this document remains valuable, serving as a technical vision that directs various teams towards a unified goal.

This deliverable is organized as follows. We start by describing the **methodology** that empowered the consortium to create this document. We then detail the overall **goals** of the OpenSwarm implementation, as satisfying these goals is the primary purpose of the implementation, and, consequently, this architecture. We then provide a comprehensive **system overview**, shedding light on both the hardware and software elements at a high level. We then zoom into the **software components** of the system, elucidating their functions and associated APIs. Emphasis is placed on the recommended **data formats and protocols** for the development, with a bias towards standards-based and widely adopted open options. After establishing these foundational definitions, we describe the **development methodology** used throughout the project, placing a particular emphasis on ensuring modularity and seamless integration. We provide guidance on the **release** process, including version numbering guidelines. Finally, we specify **how the code is employed, pinpointing the interplay** of elements in each testbed and proof-of-concept deployment.

Methodology

The architecture outlined in this document is of paramount importance for the project's efficient execution. Its definition should stem from a rigorous methodology that ensures seamless coordination among all teams developing both software and hardware components. In the OpenSwarm project, consortium partners oversee various components.

The methodology for defining the architecture comprises four steps.

First, INRIA, acting as the lead architect, liaises with all partners to gain a holistic understanding of their technical contributions to the OpenSwarm implementation, as well as the extent of their anticipated developments. From that, INRIA proposes a strawman architecture, depicted through high-level sketches and bullet points. This initial proposal serves as a foundation for ensuing technical discussions.

Second, this strawman is shared with all partners, clearly identifying the ownership of each element. This allows partners to concentrate on refining their specific components, reworking the definitions and their respective APIs. The collective feedback and revisions from all partners result in a refined architecture.

Third, the consortium embarks on a consolidation process for the various APIs. This can only be initiated once the elements and their roles are clearly understood and agreed upon. As the API is the entry point into an element, its consolidation consists in locking down the interaction between an element and the ones around it.

Lastly, the fourth step involves the compilation of this document. This is done efficiently because the consortium uses SharePoint as its collaborative platform, allowing different partners to edit different portions of this document in parallel. It is worth noting that while the initial proposal suggested the use of Atlassian Confluence, the consortium converged on SharePoint, believing it to offer a more streamlined collaborative process. Although the tools differ, their application remains consistent: establishing a single source of truth for everything about the project. Apart from a slight variation in the document's template, the outcome of using SharePoint is identical to what would have been produced using Atlassian Confluence.

Implementation Goals

The requirements of the project are formally listed in deliverable D1.1 “Consolidated Requirement Database”. The purpose of this section is not to duplicate the information from D1.1, but rather indicate how these requirements influence the overall goals of the project, and how that influences the architecture.

We define the OpenSwarm implementation as the actions that translate the architecture (outlined in this document) into a working system. This working system involves both hardware and software components. Depending on the testbed, and subsequent experiments conducted on it, the Proof-of-Concept (PoC) or any other application of the implementation will integrate various hardware and software components of the architecture.

The architecture we have designed enables components to be used in, at least, three distinct setups:

1. **During verification on testbeds:** Different experiments are conducted on various testbeds. The appropriate set of components is chosen for each experiment, compiled into a set of computer software and robot firmware, and executed throughout a series of experiments.

2. **During validation in PoCs:** There are five PoCs the OpenSwarm project is building towards, spanning a wide variety of use cases. This demonstrates the genericity of the solution.

3. **By third parties:** Individuals or entities outside the consortium may use the components during and after the OpenSwarm project’s duration. The specific uses and users remain unpredictable, which further underscores the importance of modularity and user-friendliness in the design.

From an architecture point of view, one of the most fundamental goals is to accommodate all the above use cases, which leads to three main challenges:

1. **Component specificity:** Not every component is universally applied. Therefore, it is important to ensure a highly modular approach, given that different uses may demand varying combinations of hardware and software components.

2. **Varied contributions:** Different teams will be contributing with different types and styles of implementation. This variation encompasses trivial aspects, such as coding style, but also core aspects, such as whether the provided code by partners is in source form or as a precompiled binary. Consequently, a “black box” approach becomes imperative. While we define the API for a software component, once set, the implementing team should possess the autonomy to determine its internal operations.

3. **Intellectual property (IP) constraints:** As alluded to above and as a corollary to the previous point, in some rare exceptions, due to IP considerations, not all source code will be accessible to every partner. It is, therefore, important to implement mechanisms that enable source code and precompiled binaries to co-exist harmoniously in a running system.

System Overview

This section provides a general overview of the system and its architecture, with subsequent sections delving deeper into each element.

Figure 1 presents a visual representation of the system. While there are inherent specifics to testbeds, PoCs, and third-party users, Figure 1 has been crafted to encapsulate the essence of all these different setups.

The system features various devices interconnected through a wireless network. Each device encompasses hardware and associated code, interacting with different peripherals. Given the embedded nature of this code, we refer to it as “firmware”. The role of the firmware is to drive the different peripherals of the device. One main feature is the radio, as devices typically communicate wirelessly. Driving the radio includes the deployment of a robust protocol stack. This stack encompasses different protocols, such as CoAP, and different communication patterns, such as Coaty, all while managing security and lifecycle considerations.

One important aspect the OpenSwarm project addresses is device management, particularly from the perspective of an operator. This involves “Application Performance Management”, which allows to monitor the performance of the applications running on the device, as well as mechanisms that allow for the secure oversight of firmware. Naturally, the device runs an actual application within its application plane. One key focus of our project is collaborative Artificial Intelligence (AI), where multiple devices, each outfitted with inference capabilities, collaborate in measuring and detect specific phenomena. Furthermore, when these devices are deployed as a swarm, they have the aptitude to comprehend high-level operator instructions. This is enabled by the device hosting some sort of Virtual Machine (VM) capable of receiving commands from the operator and turning those into locally actionable commands.

The devices are interconnected using a wireless network. In the OpenSwarm project, we focus on the “Time Synchronized Channel Hopping” (TSCH), a class of wireless networks that is appropriate for applications that require a high level of determinism. The OpenSwarm

project further develops the novel networking concept of zero-wire networking, allowing for very low latency communication.

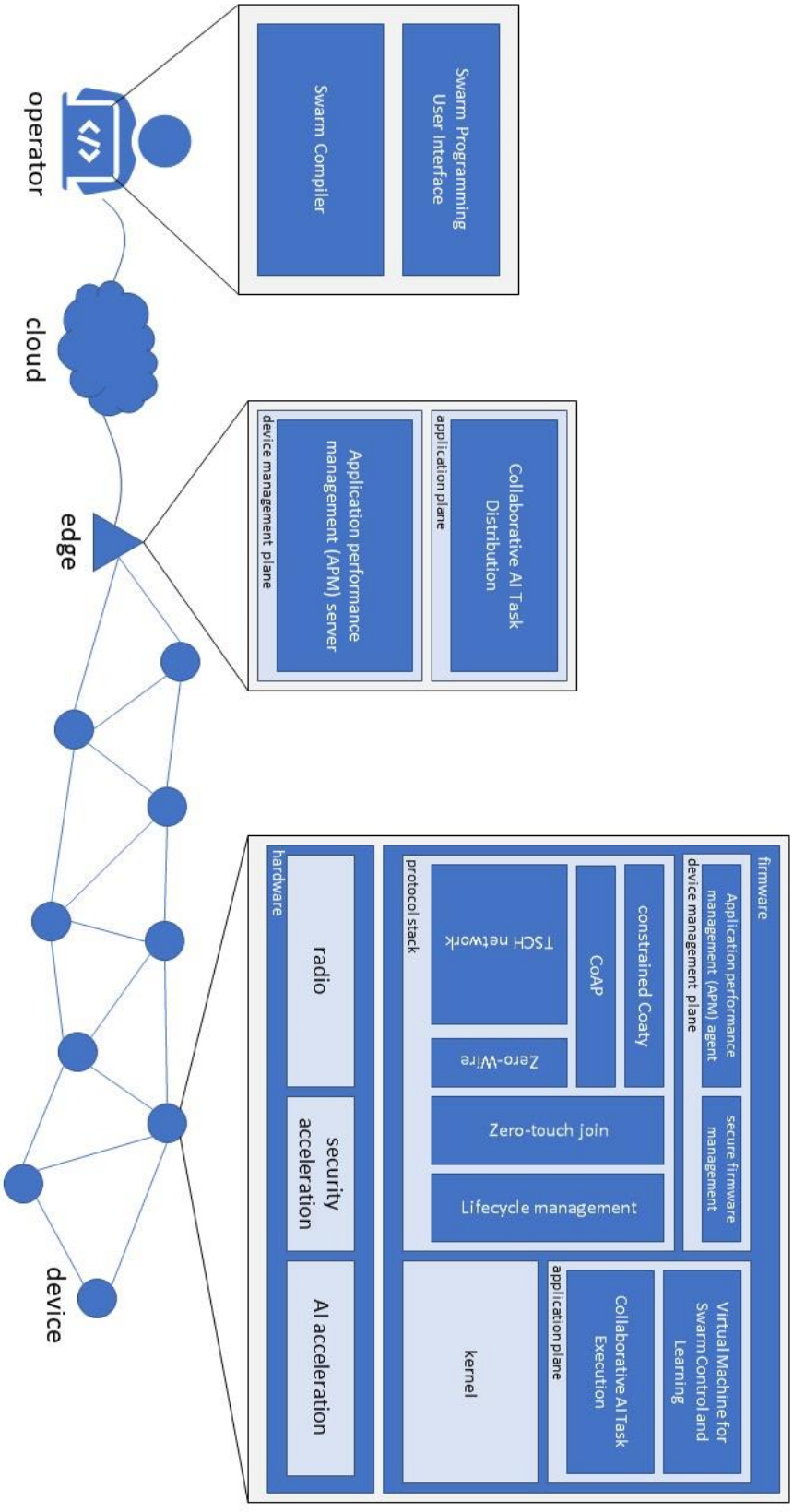


Figure 1. Overview of the architecure.

As depicted in Figure 1, the network links the devices to a designated edge device. The edge is deployed in the same area as the devices, and connects them to the wider Internet, acting as a gateway. Yet, unlike a typical gateway, the edge in OpenSwarm executes code that operates in tandem with the code on the devices. In the device management plane, the APM server collects Key Performance Indicators (KPIs) from the deployed devices, which are pre-processed by the APM agent on each device. Furthermore, on the application plane, the edge oversees a collaborative AI distribution task, orchestrating the actions of the collaboration AI task execution engine on every device.

The edge is connected to the cloud, which can be either the Internet, a Virtual Private Network (VPN) operating on the Internet, or a corporate network. The nature of the cloud does not have an impact on the behavior of the system. Its primary function is to enable remote operation, meaning that the operator does not need to be physically adjacent to the edge.

The operator is an individual responsible for managing network devices. This entails giving directives to the devices and, when needed, interacting with them. The operator interacts with a computer that runs software enabling them to operate the devices. The operator therefore interacts with a swarm programming User Interface (UI). Without loss of generality, this UI can be a web interface, i.e., a local website accessed via the operator's browser. This UI displays comprehensive information about the devices, allowing the operator to gauge their performance. The UI's main use is to facilitate device programming. One of the goals of the OpenSwarm project is to let operators express high-level instructions to the devices. Hence, "programming" does not imply writing code for individual devices, but rather expressing the desired collective behavior of the swarm. The exact articulation method will be shaped by the project's research and might involve graphical element selections on the screen, eliminating the need for operators to possess programming expertise. Naturally, there is a mismatch between the high-level nature of what the operator expresses (e.g., "collectively find an object"), and the low-level instructions a device understands (e.g., "change the speed of the left motor"). This translation is done, in part, by the swarm compiler, which turns the high-level commands into a representation interpreted by the VM on the device. The key to that compilation steps is that the output should be *both* **specific** enough that the VM knows what to do in a perfect unambiguous way, and **compact** enough that it can be transferred over the (constrained) wireless network for reasonable

fraction on the network’s capacity, with a latency acceptable to the operator, and requiring a reasonable amount of energy.

Figure 1 will be referenced in the next subsequent sections, which will detail each components’ characteristics.

Components

Organization of this Section

The OpenSwarm project develops many key innovations on the networking, AI and swarm programming scientific pillars. It is therefore natural for the project to put emphasis on the software components of its implementation. The purpose of this document is to provide a description of each of these components that provide the precision necessary to serve as a guide for the teams implementing these components.

An important technical challenge of the OpenSwarm project is ensuring the different components can be run together on one of the testbeds, one of the PoCs, or by a third party. This section serves as a blueprint, carving out the footprint of each component, and describing their interfaces.

This section is organized as follows. We group the components into three categories, which correspond to the different entities in Figure 1: Device Components, Edge Components and Operator Components. For each category, we go through the components one by one.

The description of each component has the same 3-section format. We start by describing what the component does. We try for each description to be as precise as possible, providing both general technical context so everyone can understand what that component is and why it is needed, as well as guiding assumptions about what form it can take. After this general description, we provide a high-level description of its API. In typical software parlance, the API refers to a list of functions one can call, and their exact signature. We deliberately chose *not* to define this at this point, as we believe it would unnecessarily constrain the implementation. Instead, we provide a functional and conceptual description of the APIs.

We use the terms Northbound API and Southbound API commonly used in organizations such as the Internet Engineering Task Force (IETF). These terms come from network stack design, where the Northbound protocol layer calls the layer under it.

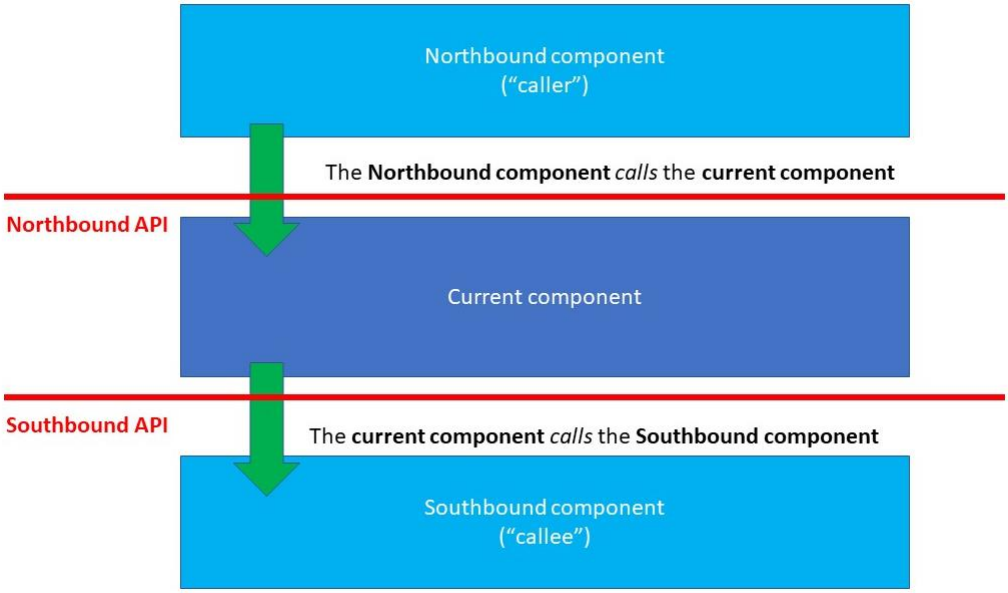


Figure 2. Defining Northbound API and Southbound API.

Figure 2 illustrates the Northbound API and Southbound API terminology. Consider a component (labeled "current component"), we position a component that calls it above it (North). The Northbound component calls functions of the current component's Northbound API. The current component's Northbound API hence consists of all the functions the current component offers, i.e., it gives access to all of the services it offers. Similarly, we position a component that the current component calls below it (South) and call it the Southbound component. The current component uses its Southbound API to exercise the services it needs from the Southbound component. The point is that the *Northbound API* lists the service the component *offers*, the *Southbound A*PI the services it *requires*.

Device Components

kernel

Description. We define kernel as the foundation of the Operating System (OS), and its most fundamental role: scheduling tasks. Here, we consider a Real Time Operating System (RTOS), where several independent tasks run concurrently and preempt one another. There are many RTOS' available, including Zephyr, FreeRTOS, uC/OS-II, uC/OS-III, RTX and RIOT. At the conceptual level, they are all equivalent: the stack pointer of each task is set to a different section in RAM, the kernel then monitors the tasks and context switches from one task to another taking into account whether as task is waiting for an event, also using task priorities. From the options above, uC/OS-II stands out because of its numerous

certifications. These indicate uC/OS-II can be used for critical applications. Zephyr is a more recent option which is receiving a lot of attention. Both options are open-source and free of charge.

In the OpenSwarm project, a lot of the code that is developed in the different software elements are “simply” functions that can be called. As a result, each component in Figure 1 should *not* be seen as an independent task. Based on the use case (testbed, PoC, third-party), it is at the development stage that elements are combined and that the code is organized in tasks. This is left to the implementation of these use cases.

uC/OS-II and Zephyr stand out. For new development, this document recommends for the consortium to pick one of those. That being said, it is unrealistic to ask a partner to reimplement any background implementation to port it to a new RTOS. This will be worked out during the implementation. We don’t see any particular risk in this exercise, as all kernels provide roughly the same basic service, and the implementation does not require anything beyond the basic functionality of switching tasks.

Northbound API. This component should provide at least the following services:

- The ability for a caller to create a task. That is, ask the kernel to set aside the resources necessary for running a task from an RTOS point of view. Typically, this includes specifying the stack size and the priority associated with that task.
- At some point in the lifetime of the device, the caller will want to start a task, i.e., have the kernel include it in its list of active tasks and switching in and out of it depending on resource availability and priorities. Likely, all tasks will be started when the device boots.
- While most kernels allow tasks to be stopped, we don’t foresee at this point that this will be necessary.
- While the tasks are running, they need to coordinate with one another. Typically, this is done through software constructions such as semaphores and mutexes, and/or message passing constructs such as mailboxes. The caller needs to be able to create these constructs.

Southbound API. The kernel has no real dependency on other software components. It uses the very basic resource of the microcontroller to be able to move the Program Counter

(PC) and Stack Pointer (SP), which all microcontrollers allow, typically by a small amount of assembly code.

TSCH Network

Description. Time Synchronized Channel Hopping is a low-power wireless networking technique in which the devices in the network tightly synchronize. A communication schedule is then introduced into the network that tells each device what to do in each of the timeslots: transmit, listen or sleep. OpenSwarm project partner ADI is world leader in TSCH products, with its SmartMesh commercial product. As part of the project, the ADI team is working on augmenting that implementation for the needs of the project. This is a fantastic opportunity as SmartMesh is a proven implementation. The protocol stack itself offers a very complete set of features (and corresponding APIs). Since the SmartMesh implementation is a commercial product, the ADI team has agreed to provide its variant for the project as a pre-compiled binary. This can be run on one of the cores of a dual-core micro-controller, which is ideal as it allows any code to run on the other core, while avoiding interference with the network stack, and while guaranteeing clear IP separation.

Northbound API. While the SmartMesh Northbound API is very complete, in the context of OpenSwarm, other components only exercise a small subset of that API. In essence, “all” other components will want to do is send a frame of data. Similarly, to be able to receive data, the Northbound component will need to arm the callback function that the TSCH network stack will be calling to notify the Northbound component it has received data.

Beyond this fundamental networking capability, we foresee a number of interactions with other components in the system:

- The APM agent may want to retrieve high-level KPIs from the TSCH network stack.
- The Zero-touch join component may want to inject the keying material it has negotiated into the TSCH network stack.
- The secure firmware management component may need to be able to retrieve the version of the protocol stack and coordinate its update.
- Similarly, the lifetime management component may want to retrieve information about the version of the TSCH network stack and its current state.

Southbound API. The TSCH network being wireless and synchronized, its implementation tightly integrates with the radio and time peripherals. That being said, ADI have already

started working on this integration, and we don't foresee any risk in this work not achieving the desired functionality or performance. Other than these interactions with hardware components, the TSCH network stack isn't calling other software components.

Zero-Wire

Description. Zero-Wire introduces the concept of *symbol synchronous networking*, a novel approach to building wireless mesh networking in which devices forward symbols (i.e., the smallest unit of data representation on the network) rather than packets in order to achieve "wire-like" performance. The symbol synchronous approach reduces multi-hop latency, which scales as a function of symbol detection time (µs) rather than packet reception time (ms). This novel approach to networking has two additional benefits. First, it eliminates the need for complex medium access control and routing algorithms. The current version of the protocol is implemented on an optical transceiver.

OpenSwarm partner KU Leuven introduced the concept of symbol synchronous networking and collaborates extensively with IMEC on developing the protocol in OpenSwarm. As part of the project, KU Leuven and IMEC are investigating approaches to port ZeroWire to Radio Frequency (RF) media using techniques such as: near-field mesh networking and redundant pulse position modulation.

ZeroWire is complementary to TSCH, On the one hand, TSCH offers high reliability and deterministic Quality of Service (QoS) for known traffic flows, at the expense of a high latency bound for unpredictable traffic. On the other hand, ZeroWire offers low latency networking for ad-hoc and mobile network scenarios, with no QoS support and limited consideration of reliability. For this reason, we envisage that TSCH will form the large-scale "backbone" network that coordinates the swarm, while ZeroWire will provide low-latency mobile networking for geographically co-located robots in collaborative scenarios.

Northbound API. The ZeroWire network will be delivered as a combined hardware/software package that integrates with the TSCH hardware and software stack described above. The ZeroWire networking API is very simple, of the form `send(address, data)` where the data payload is expected to be a small packet of no more than 8B and the 1B address may embody unicast, multicast or broadcast semantics. Similarly, a callback function of the form `receive(address, data)` must be registered by the Northbound component in order to receive messages from the network. As with the TSCH network,

ZeroWire nodes also expose support for the APM and lifetime management components. Zero-touch join and secure firmware management support are not expected to be provided within the scope of OpenSwarm as ZeroWire serves as a short-range peripheral network.

Southbound API. There is no Southbound software API as such in the ZeroWire network. Software interfaces directly with a custom-built network transceiver. However, the system will be designed in a modular manner, which enables the use of different transmitters in the future.

CoAP

Description. The Constrained Application Protocol (CoAP) can be seen as the application layer for constrained devices that is equivalent to the Hypertext Transfer Protocol (HTTP) for non-constrained Internet devices. It typically allows the devices and the operator's computer to exchange information in a structured manner, by providing a way to locate resources that are accessed (through its Uniform Resource Locator, URL), give some guidance about what action should be taken (getting resources, putting a new resource, etc.), and specifying the contents of that body. CoAP, like HTTP, is standardized at the IETF.

The OpenSwarm project does not plan on making contributions to the CoAP protocol, but rather use it as a convenient manner of enabling structure and standards-based discussion between entities. Specifically, CoAP can be used as the protocol between the program running on the operator's computer and the VM to distribute the bytecode generated by the swarm compiler. Similarly, CoAP can be used by the APM server running on the edge to gather KPIs from APM agent running on the devices.

CoAP has been developed for a number of years, and several open-source ready-to-use implementations exist, both for embedded devices and for the computer (including as extensions for modern browsers). On the embedded side, the TSCH stack terminates at the User Datagram Protocol (UDP) layer; which is ideal for CoAP as it "sits" on top of UDP.

Northbound API. Similar to the TSCH protocol stack, CoAP allows for the Northbound component to send and receive messages, by specifying a resource, method and body. Similarly, the Northbound component installs a callback function that the CoAP implementation calls upon receiving a message.

Southbound API. CoAP solely interacts with the TSCH Network stack, by sending and receiving UDP datagrams on port 5683.

Constrained Coaty

Description. Constrained Coaty is a data-centric middleware offering a unified API for swarm application developers to interact with other heterogeneous nodes in the system via defined communication patterns.

It realizes an application-centric overlay network between application components, based on exchangeable and multiple pub/sub technologies (currently based on MQTT). Additionally, it offers other features such as state sync, local routing, and persistency through this unified API, as well as a collection of metrics based on OpenTelemetry.

The current implementation successor of existing Coaty (https://coaty.io) can operate in a side-car pattern approach as a data-distribution agent or be integrated as a golang library.

Northbound API. Set of APIs for interacting with the overall distributed system from a swarm application perspective. These are gRPC if used in a sidecar pattern, or golang API if integrated as library.

Southbound API. Constraint Coaty interacts with the publish-subscribe messaging protocol MQTT or (optional) any IP based pub/sub protocol where we implement a binding.

Zero-touch join

Description. The zero-touch join component enables a device to join the network represented by a central authority, even in case when the device and the central authority do not share a common root of trust. The device, provisioned with a certificate at manufacturing time, performs a security handshake with the network's central authority. Through the assistance of a trusted third party that represents the manufacturer of the device, the device is authorized to accept the certificate presented by the network, and the network is authorized to accept the certificate presented by the device. From that moment on, the device is enrolled into the security domain of the network it attempts to join and can perform a lightweight mutual authenticated key exchange protocol, e.g. EDHOC. At the end of the handshake, the device and the central authority of the network establish a symmetric secret key, which is then used to provision the device with parameters necessary for the device to become a valid node in the network.

The standardization of the zero-touch join solution is ongoing and happens within the Internet Engineering Task Force (IETF), specifically the LAKE working group. The OpenSwarm project plans on contributing to the standardization process and implementing the solution as part of the developed software stack.

The zero-touch component depends on the implementation of the EDHOC protocol which is transported over CoAP. For efficiency reasons, the implementation typically uses a security (cryptographic) hardware accelerator. It provides a symmetric key used for network joining to the TSCH network stack.

Northbound API.

This component should provide the following services:

- A way to configure the initial keying material (e.g., certificates)
- A way for the Secure Firmware Management component to transport the attestation evidence during the zero-touch handshake.

Southbound API. The zero-touch join component relies on CoAP to send and receive the EDHOC protocol messages complemented with authorization information. Transport of EDHOC uses a standards-defined well-known resource identifier, method and payload. The component also relies on the security hardware acceleration component to accelerate various cryptographic operations needed to perform the security handshake. Once the zero-touch join procedure has completed, the derived keying material is configured into the SmartMesh IP stack.

In summary, this component uses the following services from other components:

- CoAP: sending and receiving of CoAP messages
- Security hardware accelerator: Acceleration of different cryptographic operations performed during the security handshake.
- SmartMesh IP: configuration of the key.

Lifecycle Management

Description. Firmware security vulnerabilities can be discovered at any point in time during the device lifecycle. Patching these vulnerabilities requires a secure firmware update mechanism. The Lifecycle Management component enables over-the-air firmware updates,

triggered from the Swarm Programmer User Interface. The Lifecycle Management component interacts with the SmartMesh IP stack to receive the packets containing the new firmware.

There are already standardized methods of performing a firmware upgrade. One example is the IETF SUIT mechanism which provides firmware integrity and optionally confidentiality. The OpenSwarm project does not plan on contributing to the standardization of the firmware upgrade protocol but rather to use the available mechanisms for a standards-compliant solution.

The Lifecycle Management component is in charge of receiving packets over the air and assembling them into a firmware image. The Secure Firmware Management component is in charge of verifying the firmware image integrity and booting using the newly available image.

Northbound API. This component interacts directly with the Swarm Programmer User Interface. It does not provide services to the other components.

Southbound API. This component relies on the Secure Firmware Management component to bootload the new firmware image. It also uses some base services from the microcontroller, e.g., writing to flash memory and performing a reset to trigger the new image bootloading by the Secure Firmware Management component.

In summary, this component uses the following services from the other components:

- Secure Firmware Management: this component needs to “commit” the new image in flash memory for Secure Firmware Management to bootload from it.
- TSCH network stack: this component receives packets over the network which contain fragments of the firmware image. Therefore, this component needs to define a callback function to receive packets from the SmartMesh IP stack.
- This component also uses some base services from the microcontroller, e.g., writing to flash memory and performing a reset to trigger the bootloading of the new image by the Secure Firmware Management component.

Application Performance Monitoring/ Management (APM) Agent

Description. The task of the APM agent is two-fold, depending on the definition of APM. APM may stand for Application Performance Monitoring, as well as for Application

Performance Management. The APM agent builds the monitoring and management framework in combination with the APM server. In this section we focus on the APM agent, which is part of the device firmware (FW).

When we talk about the Monitoring aspect, the task of the APM agent is to gather information about the "health condition" of the entire device and to send it to the APM server. This information is usually represented in terms of logs and traces, but we follow a different approach based on the collection of metrics. Since we deal with ultra-low power embedded devices, the overhead in terms of latency, power consumption and storage requirements caused by APM must be as small as possible. Thus, we decided to collect metrics, because they are in general much smaller in terms of payload size compared to long logging strings. Metric collection happens continuously and on-demand. Depending on the frequency, at which values tend to change. Metrics such as CPU Usage, which alter very fast, are recommended to be sampled on-demand, whereas metrics that keep track of certain counter values, which rarely change, can be updated during runtime. However, we will not “stream” these metrics to the APM server, i.e., send them out in real-time. Metrics are usually packed into health reports, heartbeats or snapshots. All these terms are interchangeable and describe the same methodology of initializing a periodic counter, which triggers the sending of such a health report to the APM server. The “packing” of the metrics into health reports is also a central task of the APM agent. Data Encryption, Compression, Aggregation and Serialization are major research areas linked to the APM agent.

When regarding the APM agent under the management aspect, its task is to react on certain management orders. These orders can either be issued by the APM server, i.e., from outside, or can be generated on-board in the APM agent itself, i.e., from inside. Orders from outside may be for instance a reaction triggered by a decision tree. This tree has proposed a certain order based on anomalies detected at the APM server, which have just become visible by the analysis of the metrics through some machine learning algorithm. Orders from inside basically work with the same mechanism and may be caused by an on-board threat detection algorithm, which has spotted, e.g., unusual behavior over a certain peripheral interface.

Northbound API.

On the device, there are **no other blocks** calling the APM agent.

On the edge, there is the **APM server** which interacts with the APM agent.

Southbound API.

At the Southbound API side, the APM agent is much more complex, since it basically needs to have access and control to all other FW blocks.

Looking at the Southbound API from a monitoring perspective again, the approach offered by the company “Memfault” sounds promising. Memfault offers a software development kit (SDK), which can be integrated out of the box into several popular RTOS implementations, such as Zephyr. The SDK offers a set of functions, which can be called to update the metrics in the internal storage of the APM agent. This would serve for the use case of continuously updating the metric values. For the other use case, where we want to update metrics on-demand, the other FW blocks would need to provide functions to sample the metric values. These functions are ultimately called by the FW agent at the end of a health report interval, just before packetizing and sending. To demonstrate the possibility to use Memfault in such constrained environments, we have already set up a proof of concept (PoC), in which Memfault was included into a Zephyr application running on an nRF chip. However, we have not used the networking functionality of the nRF, but relied on the SmartMesh IP solution. The resulting hardware setup can either be realized with two boards (nRF + SmartMesh IP mote) connected via jumper wires or with the AIOT board designed by INRIA. Communication between the application chip and the networking chip is possible over UART based on functions of the SmartMesh IP C-Library. The library needs to provide functions to gather metrics from the networking chip, i.e., for on-demand collection and Memfault provides functions for continuous collection of metrics inside the main Zephyr application running on the application chip. The PoC was successful and thus built the basis for future design decisions concerning the APM agent’s Southbound-API.

Lastly, as also in the description of this section, we also want to address the management point of view for the APM agent. Depending on the possible outcomes of the decision trees, i.e., the orders which can be executed by the APM agent, certain functions need to be defined in the Northbound APIs of the other FW blocks. Here is a concrete example: On the APM server, we detect critical anomalies in the Radio Duty Cycle (RDC) a certain device, we send an order to adjust the radio usage to save power immediately to not run out of battery power. Therefore, the APM agent on the device must forward the order to the TSCH

module, for instance. To execute this command, the TSCH module needs to offer a corresponding function in its Northbound API.

Secure Firmware Management

Description. The Secure Firmware Management component assures the integrity of the firmware running on a device. The component generates evidence about the state of the firmware and sends it over the network to an entity that verifies it. The evidence is typically a hash over the contents of the memory. Only once this attestation evidence is verified by the remote entity, the device is admitted to join the network. There are two steps in this procedure: *1)* generation of the evidence; *2)* sending of the evidence over the network to the remote verifier through a secure protocol. The generation of the evidence is done locally at a device: a piece of code that resides in write-protected non-volatile memory performs a measurement over the firmware layer that might change during the lifetime of the device. This evidence is then used in the cryptographic protocol executed between the device and the remote verifier and transported over CoAP and EDHOC.

The work on a lightweight attestation protocol suitable for constrained devices and networks is ongoing in the Internet Engineering Task Force (IETF). The OpenSwarm project contributes to this standardization process in the LAKE working group. We will reuse the security handshake performed during the zero-touch join to piggyback the attestation evidence.

This component is also in charge of the traditional bootloading of the microcontroller. It also verifies if a new firmware image is present on the device, verifies its integrity and source authentication and if valid replaces the current image with the new one.

Northbound API. This component exposes the following services to the other components:

- A way for the Lifecycle Management component to update the firmware running on a device

Southbound API. This component uses the following services from the other OpenSwarm components:

- Zero-touch join component: sending and receiving of messages during the zero-touch join security handshake

- Security hardware accelerator: Acceleration of different cryptographic operations during the generation of the evidence (e.g., a hash function)
- Base components of the microcontroller: This component relies on the presence of some functionalities of the microcontroller, like memory protection unit, privileged/unprivileged execution, memory lock functionality.

Virtual Machine for Swarm Control and Learning

Description. The Virtual Machine interprets and executes the synthesized supervisors (i.e., the device's control logic based on its high-level capabilities and specifications) exported by the Swarm Compiler. It is a program that runs within each device (e.g., robot) to trigger actions according to the specifications defined by the user via the Swarm Programming.

The Virtual Machine integrates with the device controller. For each event defined in the supervisor, a corresponding callback function must be implemented by the designer for the specific hardware platform. The callback functions are used to detect *uncontrollable* events (i.e., external inputs) as well as to trigger *controllable* events. The latter could be used to perform an action on the local device (i.e., robot action), coordinate with another device, or even a group of devices, for example, by making use of the collaborative AI scheduler.

As the supervisors (i.e., the controllers produced by the Swarm Compiler) are being executed in the Virtual Machine, there will likely arise situations where a device could choose from multiple actions, none of which violates any specification at the present or future times. The Virtual Machine will feature on-device learning algorithms to optimize the selection of controllable events (e.g., robot actions) according to objectives that are hardcoded or provided via the user interface (e.g., selecting priorities, choosing learning parameters).

Northbound API. Once all supervisors (i.e., control logic) for a device have been compiled (Swarm Compiler), it must be possible to upload them onto the device, and subsequently trigger execution. The Virtual Machine shall have functions to enable (i) upload of a file containing a description of the supervisors, (ii) start or resume execution, (iii) halt execution, (iv) to stop execution (i.e., resetting the control logic's state). These functions could be for example be called by the Swarm Programming User Interface.

The Virtual Machine shall have a function to upload the learning parameters. This function could for example be called by the Swarm Programming User Interface.

The Virtual Machine offers functions for registration callback functions related to uncontrollable and controllable events.

The Virtual Machine would normally be used to realize a robot's behavior at the highest level. Alternatively, it could be called by a higher instance (e.g., a robot main script which initiates several parallel processes such as for localization).

Southbound API. The Virtual Machine becomes aware where an uncontrollable event occurs in the current timestep (e.g., a sensor reading exceeds some threshold value; a robot received a message from a neighboring robot), and if so, triggers one of the robot's admissible controllable actions. This is realized using callback functions.

Collaborative AI Task Execution

Description. The collaborative AI task execution component is responsible for executing AI workloads. These workloads generally consist of *(1)* either using an existing trained model to infer an output prediction based on the current state of the environment, or *(2)* updating the parameters of an existing model based on novel (labeled) observations. The collaborative AI task distribution component is responsible for scheduling the different tasks.

Northbound API. Different AI tasks are scheduled by the collaborative AI task distribution block (on the edge). This component should also be able to access the current state of the different AI tasks.

A task execution component will also receive information from other task execution nodes e.g., *(1)* observations passed along as feature vectors instead of raw observations (to save bandwidth) and *(2)* model gradients to learn from other nodes without the need to send raw sensor observations.

Southbound API. Individual AI tasks will also communicate their results to other nodes. They will be able to communicate their processed observations of the environment, and model gradients in order for other nodes to update their models.

On the nodes the task execution modules will access the AI acceleration module to perform calculations.

Edge Components

Collaborative AI Task Distribution

Description. The Collaborative AI Task Distribution component is responsible for scheduling of tasks (i.e., which device will execute which subset of tasks), and communicating those scheduling decisions to the relevant devices via the wireless network. Considering the distributed applications, the scheduling section can also be integrated into the device enabling it to decide which task to choose. The scheduler considers a variety of constraints as well as the state of individual devices and the whole system. This includes device-related parameters (e.g., location, current/predicted available energy, capabilities), and environment-related parameters (e.g., geography, environmental energy availability, recharging possibilities).

Overall system objectives and related task parameters (e.g., location, required capabilities, deadline) are generally provided by an operator. The task distribution component subsequently splits high level objectives into atomic subtasks that can be assigned to individual devices for execution. This latter part is managed by the Collaborative AI Task Execution component running on the devices themselves. To enable device- and environment-aware task scheduling and distribution, the component needs to continuously monitor the device and environment state. As changes occur or new tasks arrive, the schedule can be updated to account for such changes.

Northbound API. This component exposes an interface to the operator that allows them to configure high-level system-wide objectives and tasks. This includes different types of activities such as for example *(i)* exploring and mapping an area, *(ii)* searching for specific objects/resources, *(iii)* moving object from one location to another. The interface allows semantically defining such tasks and setting deadlines.

Southbound API. This component directly interacts with the devices over the wireless network. It communicates with the Collaborative AI Task Execution component to assign specific (AI-based or other computational) tasks to devices (e.g., map a specific sub-region of the operational area, move an object, search an area for some resource/object). Moreover, it obtains information about the state of devices and their environment via the Application Performance Monitoring server. This allows the scheduling algorithm to take into account real-time state information about the whole system.

Application Performance Monitoring/ Management (APM) Server

Description. The APM server is installed on the network edge and communicates with the APM agents on the devices in the network via the network manager, also called border router (BR) in 6TiSCH networks. As for the APM agent, we will look at the APM server also from two perspectives: a first one, where the M in APM stands for monitoring, and a second one, in which the M stands for management.

The monitoring aspect at the APM server mainly comprises the tasks of data storing, visualization and analysis. When the metrics from the APM agent of the different devices reach the network edge, they need to be collected and stored in a time-series database. After that they can be visualized to either have a “live picture” (depending on the heartbeat interval length) of the health condition of the system or to plot the metrics on top of a timeline. For these purposes, there are open-source tools, like the TIG-Stack (Telegraf, InfluxDB, Grafana).

The management aspect of the APM server is more complex, since monitoring “just” made the health reports visible to the user. However, the user must find bottlenecks and detect critical trends in the network by himself. The management duty of the APM server is now to analyze the data and to design orders, which are sent to the motes. A practical example for a management task of the APM server is the detection of a critical scenario and the formulation of a corresponding order. Assume that a machine learning algorithm evaluates the health report data of the swarm and detects changing bandwidth requirements at certain motes. The management task of the APM server could then be the computation and roll-out of a new scheduling function, which cares about the allocation of cells in a TSCH network.

Northbound API.

There are no other blocks calling the APM server. However, it needs to provide a GUI for displaying the health report data and for the possibility to issue commands via the management plane directly.

Southbound API.

At the southbound API, the APM server obviously needs to interact with the APM agents in the motes’ FW over the network. Thus, the APM server and the APM agents will need to

agree on a common data format for the health reports and management commands, exchanged between each other.

Operator Components

Swarm Programming User Interface

Description. The Swarm Programming User Interface (UI) allows a user to design the behaviors of individual robots that make up the swarm based on the supervisory control theory framework and link these behaviors via events that get shared across device boundaries. The UI can also be used to facilitate deployment. In particular, the user can request for the controllers obtained by the Swarm Compiler to be uploaded onto the devices, and for swarm to start/resume, halt, and stop/reset execution.

The user defines the high-level robot *capabilities* and *specifications* through a set of automata-like state machines called *generators*, which represents what a robot can or should do. Alternative representations such as Petri nets could be considered too.

The generators are then passed to the Swarm Compiler which combines them by a process called *synchronization* to produce a device's controller called *supervisor*, which only restricts a device's controllable events (e.g., robot actions) if this is necessary to prevent a violation of the specifications at present or future times. The Swarm Compiler converts it to a compact data structure that can be used by the Virtual Machine.

Capabilities may include the device's ability to detect that an external event has occurred (e.g., a sensor reading has exceeded a certain threshold value; it received a message that it needs to handle) or actions that it can perform (e.g., actuate its motors to move in a certain way; broadcast a one-time message to neighboring robots).

Specifications use the events defined by the capabilities to restrict the actions to be triggered only when they need to.

Northbound API. The interface is directly used by the swarm designer; hence it is not called by other components.

Southbound API. This calls the Swarm Compiler to convert the high-level robot capabilities and specifications into a compact data structure and export it as a single file.

It calls the Virtual Machine for relevant devices to uploading the control file as well as to upload or update learning parameters.

It calls the Virtual Machine to start/resume, halt, and stop/reset execution.

Swarm Compiler

Description. The Swarm Compiler is a library used by the Swarm Programming User Interface to convert the high-level robot behaviors and specifications into a compact structure (e.g., JSON representation) that can be executed by the Virtual Machine.

It outputs a single file containing the data structure that represents the high-level robot behaviors and specifications.

Northbound API. When instructed by the Swarm Programming User Interface, it converts the high-level robot capabilities and specifications into a compact structure and exports a single file that can be used by the Virtual Machine.

Southbound API. The compiler does not call other components.

Data Format & Protocols

Goal

Through this document, we want to write down general guidance to the teams working on the OpenSwarm implementation. It would be counterproductive to *mandate* certain data formats and protocols, which would necessarily result in a large amount of re-implementation for code that already exists. Instead, our goal is to suggest certain choices for the team that are starting an implementation.

Data Format

We call data format the representation chosen for information itself. That is, it is the representation of the application-level payload of a protocol such as CoAP or HTTP. It is also the representation of data stored on a computer.

Choosing a data format is essential for an efficient integration of the software components. For example, it allows the team writing a parser for log file to seamlessly analyze the files generated by code written by a different team. Similarly, teams working on embedded code and computer code that exchange data can very easily work together.

We are looking for the following characteristics of a data format. First, we want it to be well established, with utilities for parsing and generating these formats that are ubiquitous in different editors and languages. This is typically favored by the fact that a format is standardized, by a well-established and active standardization body. Second, we are looking for a format which is ideally human readable, i.e., consisting of printable characters so a user can simply open a file with a generic text editor and understand what it contains and visually inspect the contents. While this may seem less important for a system that is running with no human intervention, it’s an important feature during development and whenever a human inspects data or augments the system with new features. Finally, while readability is a good feature, its counterpart is that it is not particularly compact, which makes it less efficient for transferring on a constrained network.

Given all of these requirements, JavaScript Object Notation (JSON) appears like the ideal candidate. It is standardized by the IETF with a very large adoption, including in web

development. For example, it is very common for the application-level payload of protocols such as HTTP to use JSON encoding (through its “content-type” field). Because of it being well adopted, virtually all popular editors and languages support it, making JSON-formatted data very easy to manipulate for a developer. JSON is a text-based representation, which doesn’t make it particularly compact. That being said, Concise Binary Object Representation (CBOR) is a companion binary representation to JSON: any JSON string can be translated to CBOR and vice-versa. For cases where conciseness is important, for example when a device sends information to the edge, that can be done using CBOR. One that information reaches the non-constrained world (for example when received at the edge), that device can transpose the data from its concise binary CBOR representation to the more convenient JSON representation.

JSON can be used throughout the OpenSwarm project. One example is the API of the swarm UI. It the swarm UI is built around web technology, it exposes a number of functions that can be called, as HTTP resources. The browser of the operator, after having loaded the dynamic webpage of the UI, interacts with the server by sending HTTP requests. The body of those requests and of the responses from the server can be represented using JSON. Similarly, the swarm UI may offer an option of the operator to store the behavior they select for the devices to carry out. This can be a file that the operator can then store for future use, send by e-mail to a different operator, etc. The format of that file can be JSON. As third and final example, the APM server collects KPIs from the APM agents. The agents can use the compact CBOR representation of the data when sending to the APM server, which can generate log files containing the KPIs in a JSON format so a swarm administrator can easily parse them.

Protocol

A protocol is the convention to entities use when communicating with one another. The protocol typically defines the format of the messages being exchanged, over what other protocol they run, and how the exchanged can be secured.

Of course, this section is not discussing the specific protocol used between devices and between a device and the edge, as that is the mandate of the work carried out to define and implement the TSCH network stack. Rather, this section focuses on the high-level protocol between entities on the regular network, in particular between the edge and the operator.

Similar to the data format, we are looking for a well-established protocol which is ideally standardized by a well-established organization. One additional crucial requirement comes into play: we want the operator to be able to interact with the edge without requiring any complex setup. We can imagine the operator as a “control tower” interacting with multiple edges and swarms at the same time. In that case, the operator will be remote to at least some of the deployments. If we were to use HTTP as a protocol, the installation would be complex. One option would be to mandate that the edge has a public Internet Protocol (IP) address. Another option would be for all edge devices and the operator’s computer to join the same Virtual Private Network (VPN). While all these options are possible, they place a major burden on the installation, which is further complexified by the fact that many constrained networks use IPv6 (typically through an adaptation technique called "IPv6 over Low-Power Wireless Personal Area Networks”, 6LoWPAN). Having native IPv6 support often means opening tunnels.

An alternative is MQ Telemetry Transport (MQTT). The MQTT architecture is organized around an MQTT server: all elements wishing to communicate connect to that server, using very well-established tools. Virtually all high-level languages support MQTT. Once connected, entities use a publish/subscribe pattern to communicate. Publishing means sending data to the server and specifying its topic (a string). Subscribing consists in asking the server to deliver all messages published with that particular topic. Wildcards are available, which allows efficient subscription to several topics at once. The payload of the messages can be formatted using JSON. The only complexity with MQTT is that one needs to run an MQTT server. INRIA, the project coordinator, already run several MQTT servers. Another alternative is to use a public and free server, such as the one run by HiveMQ. While such servers do not allow for the connection to be encrypted, one simple solution is to encrypt the data itself.

Development Methodology

Goals of the Methodology

This section describes the methodology used for any OpenSwarm software/firmware development. A good development methodology satisfied the following requirements. First, source code must reside on a clearly identify server accessible from anywhere on the Internet. Second, development must be done using efficient source control: one must be able to identify what line of code was modified by whom, and when. In case of a mistake, it should be possible to roll back: revert some change made by mistake. The corollary to this is that full history is maintained. During development, it is essential to have a system by which people can create tickets to indicate new features to implement and to report bugs that need fixing. Such a ticketing system must be closely linked to the versioning system: it should be possible to know what ticket was being contributed with which changes to the code. It should be possible for a developer to seamlessly create a “private copy” of the code, which they then modify, and contribute to the main code when done. Of course, developers should be able write test code and have those tests run on the source code automatically, ideally each time the code is changed. This ensures non-regression: when some code is added in one part of software, it doesn’t break some other part. Finally, there must be a convention of how to declare the code as working, by creating a release of the code that is clearly identified by a version number.

In parallel to these functional requirements, the OpenSwarm consortium does not want to dedicate resources to maintaining some complex infrastructure. Rather, we want to adopt very well-established tools and techniques.

Languages

When possible, we recommend C as the language for developments on an embedded platform. We recommend Python for any development on a regular computer, including on web servers.

Writing code, tracking tickets, testing

Again, our goal is not to break well-established routines by senior development teams. Rather, for teams starting their development journey, or established teams contributing new code to the OpenSwarm project, we strongly recommend the methodology described below.

The methodology takes full advantage of the GitHub ecosystem. We have created a GitHub team for the project at https://github.com/openswarm-eu. All developers create an account on GitHub, the administrators of that project then add those developers to the project. We use Git, an established versioning control system. This consists of a server (run by GitHub), to which Git clients connect. Developers hence install a Git client on their computer to retrieve the code ("pull" in Git parlance), modify the code ("commit") and contribute those changes back onto the server ("push").

On GitHub, we create as many repositories as appropriate, possibly one per software block, as well as one for each testbed and PoC. A repository is an independent unit of code. We favor having multiple smaller repositories, each carrying out a well-defined task, rather than one very large repository containing many different independent pieces of code.

We use GitHub issues as the ticketing system. There is one ticketing system attached to each repository. This provides a granularity that makes perfect sense. We recommend making several small issues rather than a small number of large ones. As an order of magnitude, it shouldn't take more than 2 days to fix one ticket.

We use the branching model integrated by default in GitHub. We call "main" the branch which contains the latest code. When addressing an issue, a developer creates an issue branch within GitHub, then works on that, and creates a pull request. Ideally, this pull request is reviewed by a third person, before being approved and merged.

For testing, we recommend the use of pytest, a lightweight test framework. Developers write tests, grouped in as many files as appropriate. When invoked, pytest locates those tests and executed them one by one. This can be done by the developer directly on their computer.

For continuous testing, we use GitHub Actions. This is a newer service in the GitHub ecosystem of tools which consists of farms of servers running tests automatically. Each time a developer pushes new code, some available server downloads the code, runs the tests,

and reports the output. If the test fails, this shows up on GitHub, and can be used to avoid merging new code into the main branch. Note that a GitHub Actions runner (a server that runs the tests) can easily be run in the partner's premises. This can for example be done when tests involve specific hardware, which can be plugged into this custom runner.

Releases

While the action of releasing code is a rather trivial task in a software development exercise, it is very important, from an organizational point of view, to clearly define the rules for releasing. This is why we dedicate a section for it in this architecture document.

Version Numbers

Each release comes with a version number. In the OpenSwarm project, we use a tuple with 4 elements: major version, minor version, patch version and build. When represented as text, we use a dotted notation, e.g., 1.3.2.4. When represented in binary format, we use a 4-byte array, which would be [0x01, 0x03, 0x02, 0x04]. Using a 4-byte array means in can be represented in a single word of a 32-bit architecture, which is convenient.

By convention:

- we bump the major number when there is an incompatibility between versions. That is, version 1.4.00 and 2.0.0.0 are incompatible.
- We bump the minor version when a new feature is added. That i.e., versions 1.2.3.4 and 1.3.3.4 are compatible, but the latter has more features than the first.
- We bump the patch version when we create a new release that fixes an old one. That is, version 1.2.3.5 has the same features as version 1.2.3.4, but fixes one of more bugs.
- We reset the minor and patch numbers when the higher-order number bumps. That is, when a new feature is added, we go from 1.2.3.x to 1.3.0.x, *not* to 1.3.3.x.
- The build version is an exception. It can be used to refer to a specific build of a build system. That is, some builders assign a unique build number to each build, for traceability. The development team can choose to use that build number as part of the version number. For example, this allows a developer to know that version 1.2.3.45 was released by build number 45. This means the build number doesn’t necessarily have to reset when the patch number of changes. Taking the example above, the version can go from 1.2.3.45 to 1.3.0.46. Note that, given that the build number is encoded in only a single byte, its maximum value is 255. Since builder create a build for every code change, clearly there will be many more builds than

255. We leave it up to the development teams to choose what strategy to adopt, for example to use the least significant byte of the build number of the builder.

Compatibility matrix and compound releases

Using the OpenSwarm code on a testbed or a PoC means combining a selection of OpenSwarm components (see the section below). Each of these components has a release number. It is therefore important to agree on how to do a compound release. We recommend giving a compound release a version number as well, using the exact same rules as above. Of course, a compound release can only consist of a collection of actual releases from each of the components. It is important to clearly note the exact versions of the releases of each of the components in that compound release. The exact mechanism of how this is done is left to the implementor. One option is to put these version numbers in the release notes of a release, on GitHub (see below). Another option is to use a tool such as West to pull specific versions of different repositories.

Release Process

We use the release process of GitHub. When the code is ready to be released, the developers make sure the latest code is merged into the main branch, and that all tests pass on that branch. The lead developer then creates a new release through the GitHub web interface and chooses version number REL-A.B.C.D, when A, B, C and D refer to the major, minor, patch and build versions, respectively. Once created, the lead developer adds a description in the markdown description associated with the release. For any release that is built (e.g., firmware that is compiled into a binary, a Python package that is released using a wheel), that artifact is attached to the release, directly on GitHub. Right after the release, we bump the version number, ready for the next release.

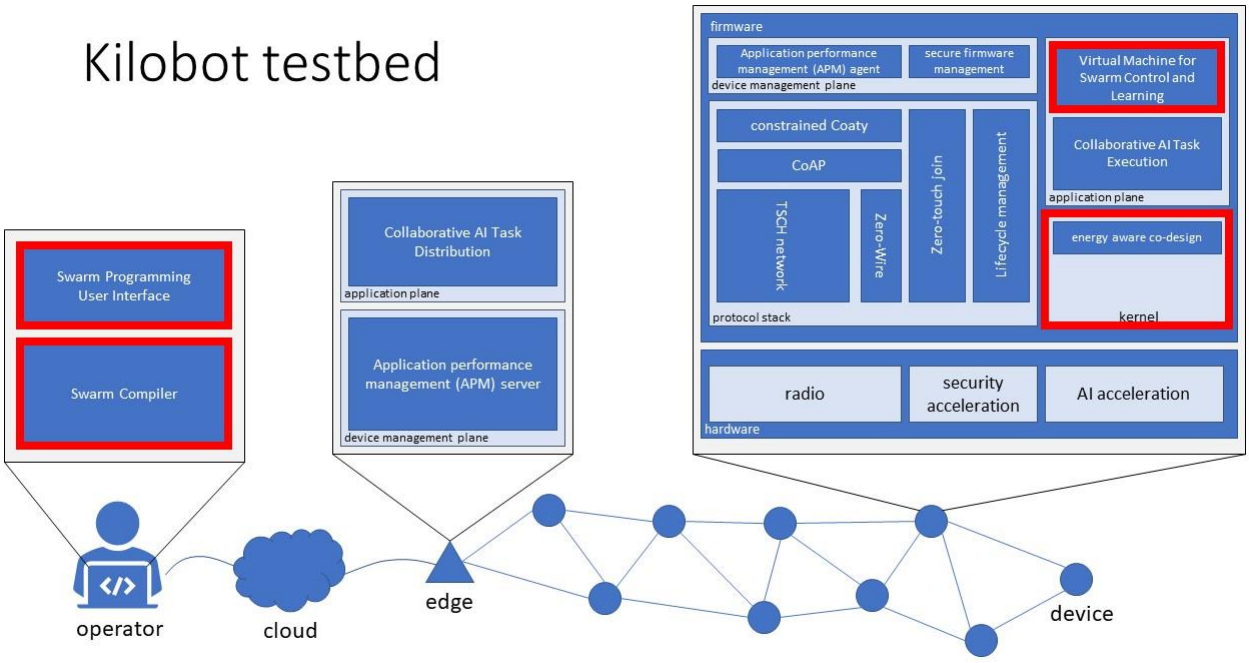
The process above assumes a manual release process. In a pure Continuous Integration/Continuous Delivery (CI/CD) pipeline, all these steps are done automatically when all tests pass. This is of course the best setup, and development teams are encouraged to set up such a CI/CD pipeline. Regardless of the manual/automated nature of the release process, the versioning described above stays valid.

Using the Code

The OpenSwarm implementation is a very complete collection of software components. The implementation is used in two testbeds, and in five PoCs within the project, and by third parties during and after the project. Of course, using all software components in each of the cases doesn't make sense, and different combinations are needed. This section details the combinations for each of the testbeds and PoCs.

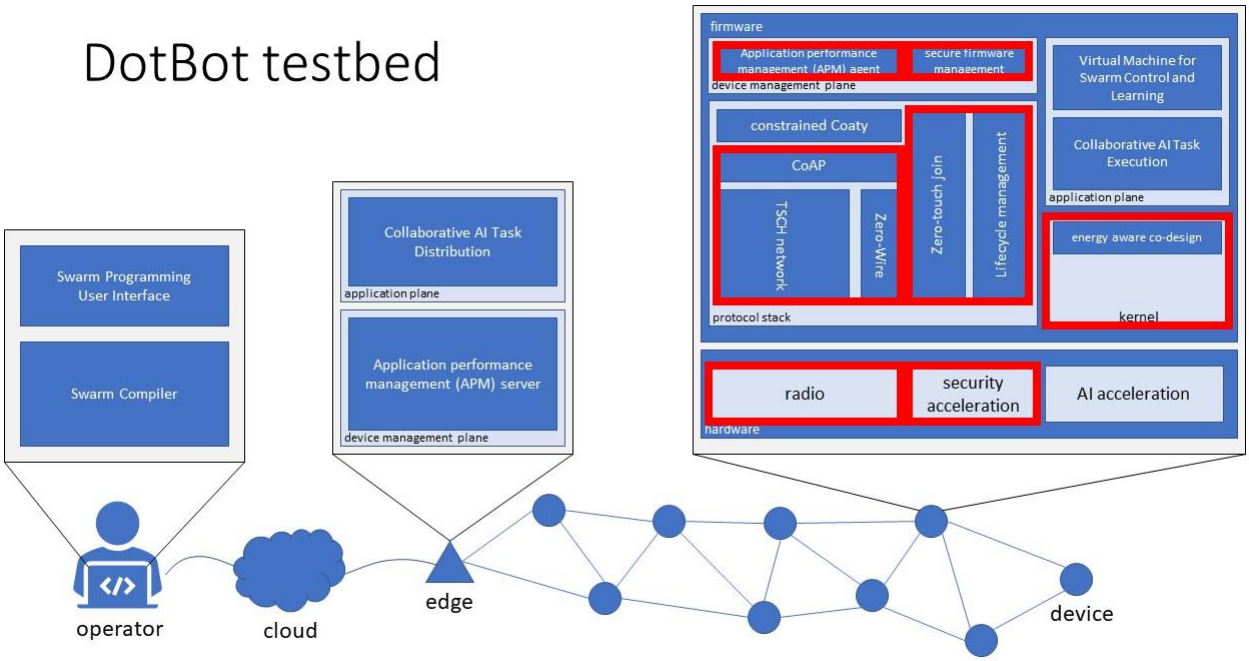
Using in a testbed

This section shows what software components are used in each of the testbeds.



The figure above details the software components when running the OpenSwarm implementation on the Kilobot testbed. The Kilobot is a small robotics platform, which moves across a tabletop by switching on vibrating motors. Communication is done by blinking an LED on the sender, and detecting the light with a light sensor on the receiver. The testbed is composed of up to 1,000 Kilobots.

The tests of the OpenSwarm implementation focus specifically on swarm programmability and power consumption. These components are hence run on the Kilobots and the operator.

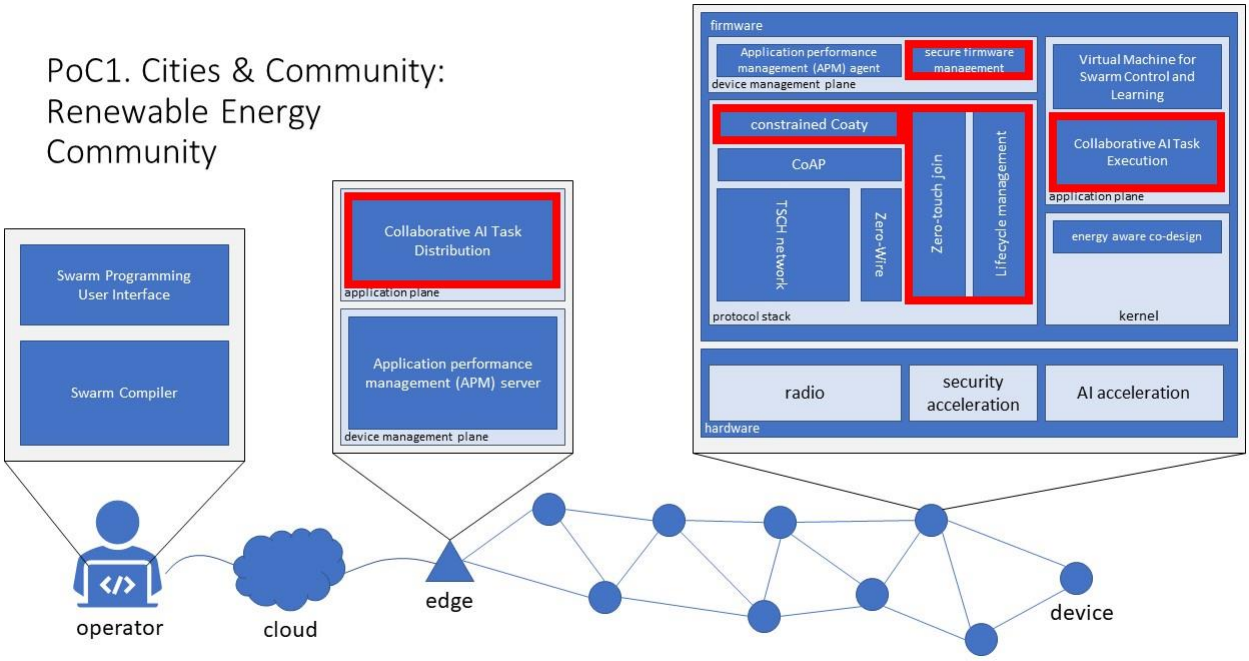


The figure above details the software components when running the OpenSwarm implementation on the DotBot testbed. The DotBot is a 10cm x 10cm robotic car, which can move across a building floor at walking speed. Communication is done wirelessly. The DotBot is equipped with a state of the art low-power microcontroller and radio. The testbed consists of up to 1,000 DotBots.

Tests on the DotBot platform focus on the networking stack (including security), its management and low-power.

Using in a PoC

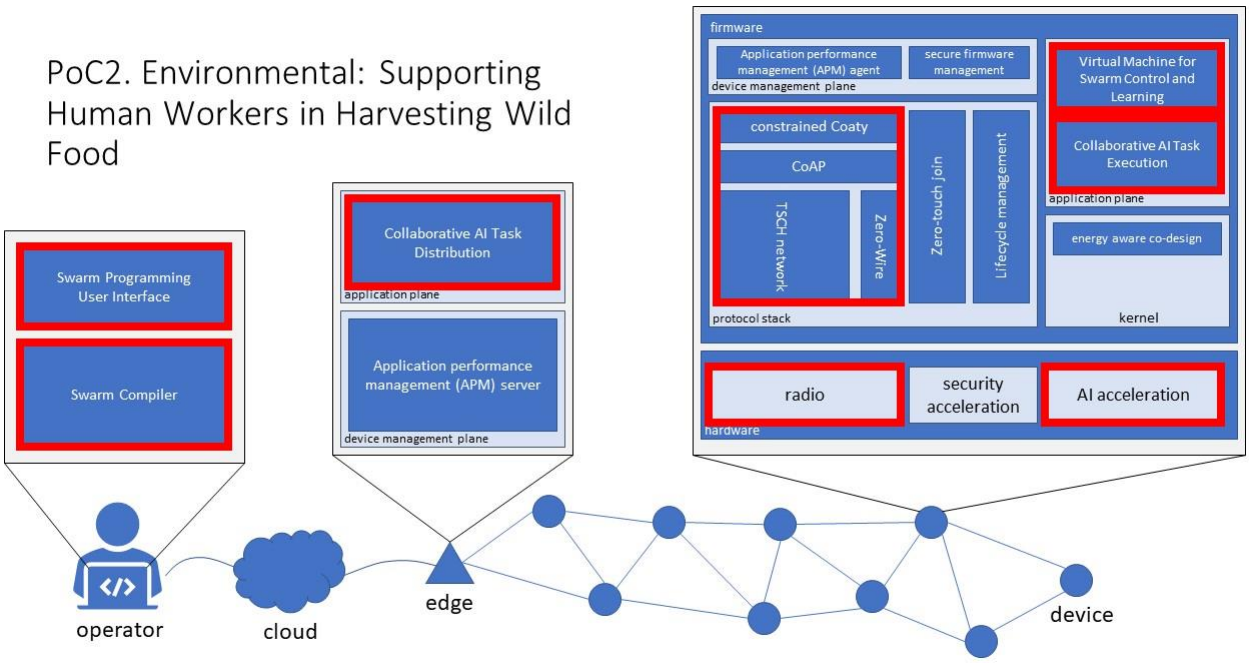
Similar to the testbeds, each PoC is different. The figures below detail the software components run on each of the PoCs.



The figure above details the software components when running the OpenSwarm implementation on PoC1 “Cities & Community: Renewable Energy Community”. Through this PoC, we enable the implementation of renewable energy communities (RECs) to achieve climate neutrality and decarbonize the energy system. OpenSwarm develops a system of smart sensors (the devices) that orchestrate and interact in a collaborative manner with the customer gateway (the edge) using distributed swarm intelligence and low-latency communication. The devices monitor raw data (e.g., instantaneous energy consumption), forward that to the edge device which generates a summary of the data (e.g. using a Exponential Moving Average) to the OpenSwarm cloud. The cloud, which runs remotely, dynamically orchestrates the local electricity demand (the consumption of the homes) with the supply (the production of their solar panels). This use case verifies the OpenSwarm network's ability to operate renewable energy communities with a hierarchical network organization, as well as to analyze the current consumption pattern using distributed AI.

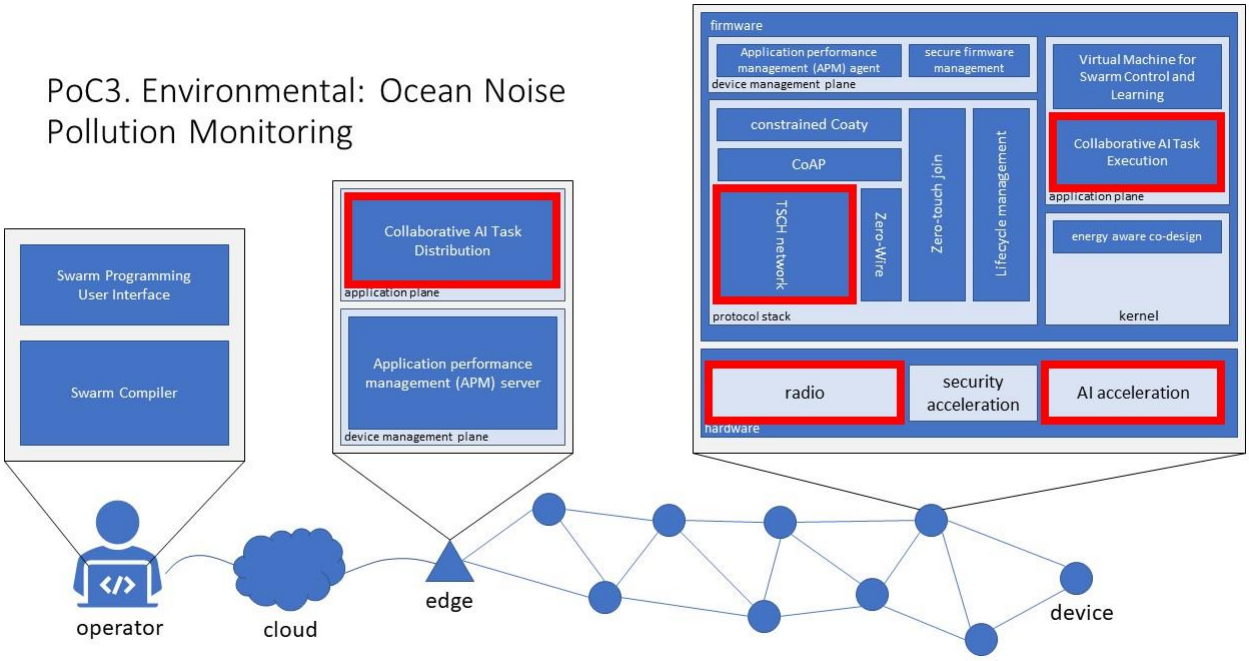
The use case enables the flexible and adaptable interaction of energy consumers and producers, offering business opportunities for new and existing players in the energy domain. It potentially allows for lower energy prices for the end-user, as electricity can be shared between neighbors directly and independently in an energy island. The employment

of renewables and its orchestration in the grid leads to a higher share of energy consumption with zero carbon footprint.



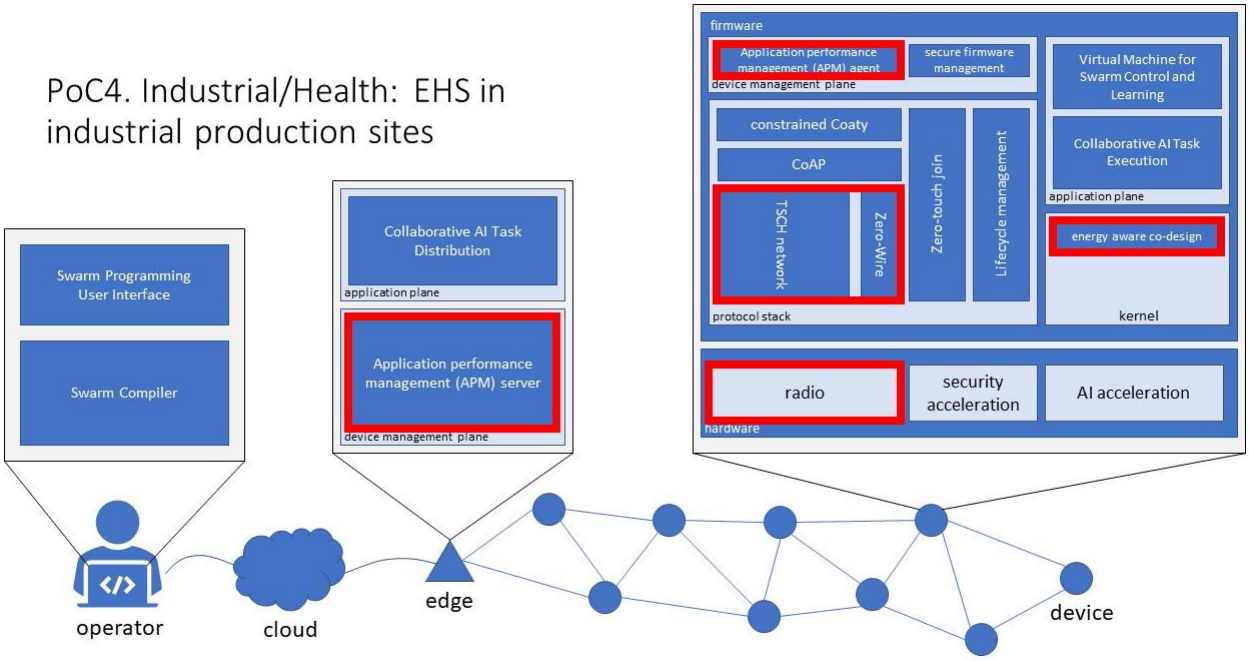
The figure above details the software components when running the OpenSwarm implementation on PoC2 “Environmental: Supporting Human Workers in Harvesting Wild Food”. Through this PoC, we use drones to map where there are a lot of wild berries or mushrooms, allowing pickers to pick fruit faster. Each drone (the device) is equipped with a camera, a wireless radio, and enough processing power to run motion control and image recognition routines on-board. Each drone runs the OpenSwarm networking technology, allowing it to communication with a leader drone (the edge). Each drone takes pictures of the trees it flies over and identifies where fruits are. It does so for example by using a model previously trained on a dataset of pictures. The drone forwards that information to the edge, that serves as the expedition leader and coordinates with the cloud. The cloud builds a map of the locations of the wild berries or mushrooms, and navigates the drones.

Swarm intelligence uses forest-related data for better monitoring and estimating the fruits' life, location and yield. Swarm intelligence coordinates the work between farmers and robots, thereby making the harvesting work more efficient (faster picking).



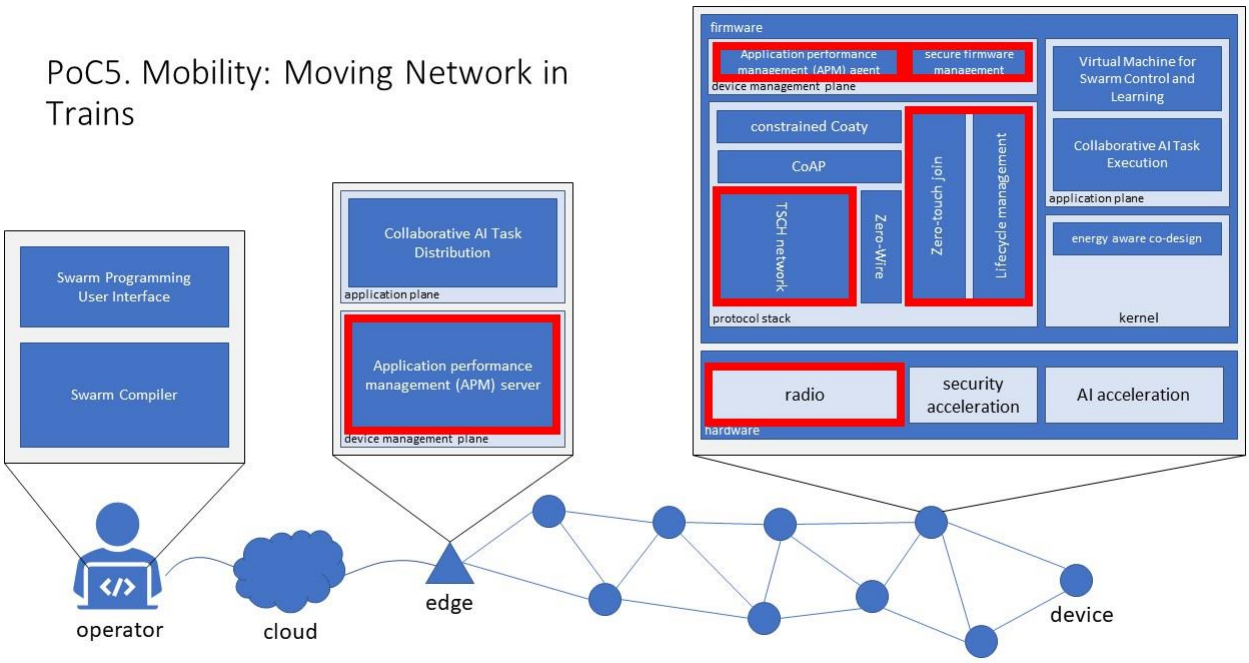
The figure above details the software components when running the OpenSwarm implementation on PoC3 “Environmental: Ocean Noise Pollution Monitoring”. Through this PoC, we develop an automated system that monitors and counts boats in a protected marine area (PMA), and tracks their speed to help manage underwater noise pollution. The proposed system uses smart buoys (the devices) equipped with hydrophones. The buoys can communication with one another using OpenSwarm’s mesh networking technology. One of the buoys is the gateway (the edge), and is connected to the OpenSwarm cloud. The buoys listen and detect boats from the sounds recorded by their hydrophones. They send the timestamped sound signature to the edge, which compares them, computes the location and speed of the boats, before forwarding that information to the OpenSwarm cloud running on a server on the Internet. The project demonstrates swarm communication, constrained AI framework and energy-aware swarm operation. It takes advantage of the OpenSwarm swarm compiler to allow a user to efficiently program and control the behavior of the devices.

The AI model can assess the presence and well-being of wildlife and thus contribute to managing the traffic so that damage to nature is minimized. This system can be applied to all 3,150 European PMAs and to approximately 5,000 marinas in Europe.



The figure above details the software components when running the OpenSwarm implementation on PoC4 “Industrial/Health: EHS in industrial production sites”. This use case focuses on the importance of Environment, Health, and Safety (EHS) measures in industrial sites, which can be potentially dangerous as workers and moving machines are close to one another. We equip workers with smart OpenSwarm wearables, and attach smart tags to static and mobile machines/robots in the factory (the devices). These devices form a highly reliable low-power wireless network around a gateway device (the edge). The devices monitor the distance between one another and send those measurements to the edge, which summarizes the information (e.g. removing duplicate information) and forwards that to the OpenSwarm cloud. This allows the cloud service to closely monitor the distance between workers and machines/robots, and it ready to stop robots to avoid injuries.

Environment, Health, and Safety measures are legally required in Europe. Their continuous monitoring is expected to result in a significant reduction in accidents (and associated costs) and improved well-being for workers, as well as reduce the duration machines are not operating.



The figure above details the software components when running the OpenSwarm implementation on PoC5 “Mobility: Moving Network in Trains”.

The shift of passengers and goods from air/road to rail significantly reduces CO2 emissions. Existing sensor systems for monitoring train components typically need to be installed at manufacturing and often don’t allow for train cars to be rearranged between trains. OpenSwarm creates a reconfigurable network of wireless sensor nodes (devices). We retrofit them on an Intercity Express passenger train. They monitor the state of key elements of the train, for example the vibration of moving elements. They communicate with a gateway on the train (edge). The devices monitor the raw data, the edge summarizes the data and sends that to the OpenSwarm cloud, which determines whether any element on the train needs maintenance. This use case also validates the “zero-touch” security solution that allows the sensors attached to train cars to reorganize into different trains in a secure manner.

A high percentage of train connections do not reach their destination and a significant delay is caused by disruptions and the availability of train cars and traction units. This results in extra costs, and in a decrease in customer satisfaction. OpenSwarm's AI-augmented approach is expected to bring game-changing accuracy, making it strategic for any train manufacturer and operator. This solution can be retrofitted to existing trains, a much

cheaper and flexible solution than having to integrate them at manufacturing. The shift of goods from road trucks to train, and passengers from planes to trains reduces carbon footprint.

Summary

The purpose of this document is to define the architecture of the OpenSwarm implementation. It is meant to be used as a reference and be consulted throughout the project. It gives a big-picture view of the software components of the project before delving in great details into each of the components. For each, this document describes what the component does, as well as its conceptual Northbound and Southbound APIs. That is, what the component offers to and except from other components, respectively. This document further details the development methodology, including, in great detail, the tools used to ensure efficient collaboration on the different teams, including source control, ticketing, tests and releases. Finally, this document lists the software components on each of the testbeds, and in each PoC.

Glossary

6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks
AI	Artificial Intelligence
API	Application Programming Interface
APM	Application Performance Management
CBOR	Concise Binary Object Representation
CI/CD	Continuous Integration/Continuous Delivery
CoAP	Constrained Application Protocol
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MQTT	MQ Telemetry Transport
OS	Operating System
PC	Program Counter
PoC	Proof of Concept
RAM	Random Access Memory
RDC	Radio Duty Cycle
RTOS	Real-Time Operating System
SP	Stack Pointer
TSCH	Time Synchronized Channel Hopping
UDP	User Datagram Protocol
UI	User Interface

URL	Uniform Resource Locator
VM	Virtual Machine
VPN	Virtual Private Network