POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in
Computer Science and Engineering

Exploiting high bandwidth, low latency wireless networks with Internet-connected drones

Supervisor:
Prof. Luca Mottola

Master Graduation Thesis by:
Pietro Avolio
Student ID n. 878640

Academic Year 2018-2019

To my Parents
who have never stopped believing in me
not even for a single moment

Acknowledgements

This thesis work has been an amazing journey inside the scientific research and I'm very grateful to everyone that supported me.
My first big thanks go to my advisor, Prof. Luca Mottola, for giving me the opportunity to work on this topic, and for challenging and inspiring me during the entire process.
Then, I want to thank my Parents and my brother Emanuele, because they always are the strongest pillar of my life.
And then Marta, because she and only she knows what bonds us in our heart of hearts.
All the NESLab guys, Andrea, Francesco B., Francesco C., and Fulvio, for all the interesting discussions, the tips, the jokes, and for being such a great support during these tough times.
Finally, all my lifetime friends for having played an important role in who I am today as a person: Alfredo, Alessandra, Cristina, Gianmarco, Lorenzo, Daniela, Ilaria, Francesco, Pierdomenico, Giuseppe, Giacomo, Valeria and Riccardo.

Abstract

High-performance networks are those local, metropolitan, and wide area networks that provide ultra-low latencies, bandwidths in the order of Gigabits per second, and a very high reliability. The access to this class of networks empowers applications enabling new forms of machine-to-machine and machine-to-human interactions. In the last years, they started to be available also outside research facilities, and the 5G cellular network is a clear example of this trend. 5G has an important additional feature: it is a *wireless* network. This characteristic allows also high-mobility robotic vehicles to access the improved network performance.
In this thesis work we investigate how Unmanned Aerial Vehicles (UAVs), a specific instance of high-mobility robotic vehicles, can benefit from networks with such improved performance. In particular, we first identify some issues and limitations of existing state-of-the-art solutions, then we identify and describe some new application scenarios in which UAVs can be involved. After this analysis, we come to the conclusion that it is not sufficient to connect UAVs to a better network to benefit from all the advantages. It is necessary a software platform to properly support this integration.
Hence, we propose a programming model and an architecture that specifically target the exploitation of high-performance networks. To evaluate our contribution, we implement a proof-of-concept software platform that embodies the requirements expressed by the programming model and by the architecture, and that is deployable into real drones. We named this software platform *NG IDrOS*.
Finally, we test the performance of this new software platform by measuring a series of different performance metrics in different scenarios. These metrics demonstrate that the overhead latency added by the software stack is lower than the latency introduced by the network transport, and that the performance of *NG IDrOS* is inline with the performance of similar software implementations.

Sommario

Le reti ad alte prestazioni sono quelle reti locali, metropolitane e geografiche che offrono bassissime latenze, larghezze di banda nell'ordine dei Gigabit al secondo e un'elevata affidabilità. L'accesso a questa classe di reti abilita le applicazioni a nuove forme d'interazione macchina-macchina e macchina-uomo. Negli ultimi anni, queste reti hanno iniziato a essere progressivamente disponibili anche fuori dai laboratori di ricerca, e la rete cellulare 5G è un chiaro esempio di questa tendenza. Il 5G possiede anche una importante caratteristica aggiuntiva: è una rete *wireless*. Questo consente anche ai veicoli robot a elevata mobilità di accedere alle migliorate prestazioni della rete.
In questo lavoro di tesi studiamo come gli Aeromobili a Pilotaggio Remoto (APR), uno specifico tipo di veicoli robot a elevata mobilità, possano beneficiare di queste migliorate performance di rete. In particolare, prima identifichiamo alcune limitazioni e problemi dell'attuale stato dell'arte, quindi identifichiamo e descriviamo una serie di nuovi scenari applicativi in cui gli APR possono essere coinvolti. Da questa analisi è evidente come non sia sufficiente connettere gli APR a una rete migliore per beneficiare di tutti i possibili vantaggi. È necessario che questa integrazione venga supportata da una piattaforma software specificatamente progettata tenendo al centro del processo le reti ad alte prestazioni.
Pertanto, proponiamo un modello di programmazione e un'architettura che mirano specificatamente all'utilizzo delle reti ad alte prestazioni. Per valutare il nostro contributo, implementiamo una piattaforma software che soddisfi tutti i requisiti del modello di programmazione, che rispecchi la struttura dell'architettura e che possa essere utilizzabile con droni veri e propri. Questa piattaforma prende il nome di *NG IDrOS*.
Infine, testiamo le performance di questa nuova piattaforma software misurando una serie di diversi indicatori di performance in scenari differenti. Questi indicatori dimostrano che la latenza introdotta dalla piattaforma software è minore di quella introdotta dal trasporto di rete, e che le performance di *NG IDrOS* sono in linea con quelle di implementazioni software simili.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The rapid advancements in network technologies led to high-performance networks to become more and more common outside experimental setups or application specific setups. *High-performance networks* are those local, metropolitan or wide area networks showing significant improvements in two major performance indicators: latency and bandwidth. When we talk about improved performance, we refer to a latency lower than 10 milliseconds and to a bandwidth in the order of Gigabits per second.
Applications can strongly benefit from these improved performance: high bandwidth allows a multitude of nodes to be connected at the same time and lets applications transfer large amount of data in a very short time; low latency allows distributed central loops in which clients can reach data providers and service providers in milliseconds. Whenever high bandwidth and low latency characteristics are present in the context of high availability, applications are further extended to mission-critical scenarios requiring ultra-reliable communication.

On these premises, new scenarios range from the context of massive internet of things such as sensor networks, to extreme real-time interactions such as tactile internet, and to lifeline operations such as disaster recovery and e-health services [1]. The adoption and spread of this kind of applications is expected to have high impact on both social and economic development of our society in the immediate future [2] and are thus getting increasing attention from both companies and governments.
Exploiting high-performance network characteristics without the constraints of a wired connection enables the final step to extend applications to the most interesting range of scenarios. Using wireless connectivity, devices can be deployed anywhere, they can operate without any constrictions and, most important, wireless connectivity fully allows client mobility.

UAVs — commonly known as *drones* — embrace the concept of client mobility at its finest. UAVs are getting increasing interest in both indus-

trial and consumer applications because of their ability to physically reach inaccessible or risky areas. Furthermore, they are incredibly versatile to be adapted to serve different scenarios because they can carry a specific payload for each specific mission.
Just like drones are the perfect example of high mobility clients, the 5G cellular network is a vivid example of wireless network connection providing the aforementioned high-performance characteristics. The new technologies designed to implement this new generation digital cellular network are expected to deliver data with less than a millisecond of delay and to provide peak download speeds of 20 Gigabits per second, as well as the capability to efficiently serve many more clients simultaneously [3] [4].

High-performance network connectivity can unlock an undiscovered potential of high-mobility clients, extending the class of applications they can be involved in, and enabling new machine to machine, machine to user, and machine to environment interactions.
A 5G connected drone could, for example, overcome the limitations coming from the constrained on-board computational power by leveraging cloud computing or edge computing capabilities. Similarly, leveraging the extreme low latency, an entire fleet of drones could in real-time exchange data and resources to collaborate in order to accomplish a given mission. The real use cases we are going to describe deal with search and rescue mission, surveillance, aerial mapping, video streaming, and pollution monitoring.

1.1 Contribution

The research work underlying this thesis started with the purpose of identifying the class of applications that would benefit from high-performance networks.
The goal was to identify the contexts in which high bandwidth, low latency and high reliability can significantly impact the applications to a point that these network characteristics become key enablers.
We studied the current state of the art of high-mobility robotic vehicles, such as UAVs. We focused on *autonomous* high-mobility robotic vehicles, i.e. those capable of accomplishing missions without or with limited human intervention. Those robots are able to run applications to govern their behaviour, movements, and interactions.

We identified the opportunity to investigate the exploitation of high-performance networks in two specific matters:

- To further overcome classical limitations of high mobility robotic vehicles, such as constrained computational power and constrained power source they typically carry on board.

- To extend the class of applications in which high mobility robotic vehicles can be involved in, by empowering them with new capabilities that rely on the improved network performance.

The way we are going to investigate these opportunities is not by just connecting high mobility robotic vehicles to a faster network and to verify if already existing solutions can perform better. Our approach consists in identifying, describing and then implementing and evaluating new solutions that are specifically tailored to leverage the network characteristics.
To analyse the outcomes of such integration between high mobility robotic vehicles and high performance wireless networks, in this thesis work we propose a programming model, an architecture and a proof-of-concept implementation, that we in detail describe in the following chapters.

The programming model is built upon two main pillars: the client mobility and the network interactions. In the first place, it has the purpose of providing a set of abstractions to both pilot the drone and to run navigation or other kind of applications on it. The application programmer should have the capabilities needed to interact with the motion-control stack and the onboard payloads such as sensors and actuators. In this way, the programmer is able to implement active sensing and active actuation techniques, that is to implement applications to make the robotic vehicle fully autonomous in accomplishing its tasks. Second, it provides abstractions for network interaction, enabling the application programmer to implement remote monitoring and remote control applications. The core contribution given by this programming model is to support new advanced capabilities that are in first place possible thanks to the improved network performance, and that are not present in state-of-the-art solutions.

The first capability is about sharing hardware resources among clients. Leveraging the ultra-low latency, programmers are able to build applications that access in real-time resources made available by other nodes in the network. The programming model specifically provides abstractions to support *Remote Sensors Sharing*, since sensors are core resources in robotic vehicles missions. The second capability is about providing abstractions to move the computation of application logic parts towards other hosts inside the network. Exploiting *Computation Offloading*, programmers are able to run complex applications on devices with limited resources by executing the most intensive parts on hosts that share their computation power and storage over the network. Thanks to the high bandwidth, large portions of source codes, the input parameters, and the return values can be moved among peers in a short time, regardless of their size. Ultra-low latency guarantees that invocations and results are exchanged in milliseconds. The last capability, named *Communication Bus*, leverages the reliability of the network to empower programmers to have communication mechanisms and

synchronization mechanisms at the core of their applications. The programming model supports abstractions to create, join, and exchange information over *communication channels*, which are created at runtime among peers. Also in this case, the ultra-low latency guarantees that messages are exchanged in real-time.

The proposed architecture has the purpose of showing how all the involved elements can be layered. Correctly layering the elements allows us to create a clear distinction of functional roles and to correctly define the requirements for each specific role. The architecture is built around the programming model and it is meant to be flexible enough to support future changes or extensions in the model itself. Being able to extend the architecture and the programming model is central in the context of high-mobility robotic vehicles, because it enables the integration towards new platform and systems.

To conclude, a proof-of-concept implementation of the aforementioned architecture and programming model is provided in order to be able to perform real-world use-cases evaluations and to test possible implementation solutions. The software platform we implemented specifically targets UAVs as instance of high mobility robotic vehicles, and the 5G cellular network as instance of high-performance networks. This choice is made without any lack of generality: everything is described applies to every kind of high mobility robot such as copters and rovers, as well as other high-performance network instances.

To evaluate the performance of this software platform, we designed and executed a series of experiments to calculate the overhead that the software stack adds over the network latency, as well as to evaluate how it performs compared to similar implementations. On average, the software stack introduces a latency that is lower than the one introduced by the network, and performs better than the comparison applications in most of the conditions: this demonstrates the suitability of the implementation choices we made.

1.2 Thesis structure

The rest of this thesis work is structured into five chapters.

Chapter 2 contains the detailed description of background concepts that are widely used in this document: UAVs, the 5G cellular network and the Mobile Edge Computing. We then perform the analysis of current state-of-the-art software platforms for drones in order to identify the gaps that limit these software solutions from fully exploiting high-performance networks.

Chapter 3 contains the description and analysis of some motivating scenarios showing how high-mobility robotic vehicles can benefit from improved network performances. Such benefits are able to solve part of the gaps identified during the state-of-the-art analysis, as well as to support totally new capabilities. From these scenarios, we derived a list of functional and non-functional requirements that drove the design of a programming model. Specifically, we describe three macro-functional areas that constitute the core contribution of the programming model: *Remote Sensors Sharing*, *Computation Offloading* and *Communication Bus*.

Chapter 4 presents the architecture of the software platform we designed to support the programming model, named *NG IDrOS*. Particular emphasis is given to the description of how functionality are layered and how components are decoupled, and how this design strategy led to extensibility and deployment flexibility.

Chapter 5 contains the description of the implementation we developed for NG IDrOS, how it embodies the programming model and the architecture that are outlined in previous chapters, and the choices we made during the process. The most relevant part of this implementation is the one dealing with the network. We decided to implement a network protocol based on *Calvin*, an environment for IoT applications that mixes the Actor model with the Flow Based programming model.

Chapter 6 presents the performance metrics, the experiments and the comparison we performed to evaluate the implementation of the software platform. In particular, we are going to show that the latency introduced by NG IDrOS is lower than the latency introduced by the network transport, and that its performance are in line with the performance of similar applications. This demonstrates the suitability of the implementation choices we made.

Finally, Chapter 7 summarizes the conclusion of this thesis work and proposes some possible future works to expand it.

Chapter 2

Background and State of the Art

2.1 Background

The purpose of this section is to provide to the reader some basic concepts about paradigms and technologies that will serve as background knowledge for future reasoning in this thesis work.
We will in detail describe UAVs as instance of high mobility robotic vehicles, and the 5G cellular network as instance of high-performance networks. Finally, we will outline the Mobile Edge Computing paradigm, which is tightly coupled with 5G.

2.1.1 Unmanned Aerial Vehicles

An *Unmanned Aerial Vehicle*, often shortened to *UAV* and commonly known as *drone*, is an aircraft without a human pilot on board. The UAV is usually a component of a more complex system including also a ground-based controller connected to the drone. The whole system, including UAV, ground-based controller and communication infrastructure is known as *unmanned aircraft systems (UAS)*. UAVs can fly autonomously or be remotely piloted.

UAVs developed mostly in military applications where missions are too dangerous for humans or even unfeasible using traditional crewed aircraft. Thanks to the rapid technological development of the XXI century, UAVs started to be more and more employed also in civil applications. A big input to civilian drones development came from the Do-It-Yourself community that started to grow in 2007 as a group of amateurs that began to assemble drones themselves [5].

Some current common civilian UAVs applications are [6]:

- Inspection and monitoring;
- Surveying and mapping;
- Aerial photography and imaging;
- Search and rescue operations.

Outside these common applications, several experimental projects are being carried on. For example, in 2016 Facebook tested solar-powered autonomous drones to act as relay stations to provide internet access to remote areas [7]. Similar efforts are being carried out by Loon LLC — an Alphabet Inc. subsidiary — to experiment high-altitude balloons placed in the stratosphere to create an aerial wireless network to provide internet access to rural areas [8].

UAVs are extremely versatile and can be exploited in a very wide range of scenarios because of their unconstrained, application-controlled mobility. Moreover, they are able to carry a specific payload for a specific application. They started to gain even more interest with the growth of the Internet of Things, to the point of letting researchers talk about the *Internet of Drones* [9]. The integration of UAVs inside the IoT ecosystem resulted to be particularly successful for two main reasons [10]:

- Drones benefited from many technical advancements developed for other IoT applications and devices. IoT empowered drones to evolve into *smart* drones, with the possibility of running applications to make them fully autonomous in fulfilling their tasks. The main result was to overcome the locality bonding which constrained drones to work in presence of a human operator.
- UAVs further extended IoT applications becoming IoT enablers on their own. Drones can be sensors or actuators at the same time, and their extreme mobility allows to carry these sensing and actuating actions everywhere, especially where the deployment of other IoT devices is impossible or inconvenient.

However, there are three factors that can affect their operability and should be carefully taken into account when designing drones applications:

- The constrained load capacity: drones are characterized by a maximum weight they can carry. This limit sums up the weight of the drone itself (including engines, batteries, etc) and the weight of the payloads. Increasing this limit requires, in most cases, to change the size or the type of drone itself, for example moving from a quadcopter to a hexacopter.

- The constrained computational power: often drones do not host enough on-board computational power to support real-time applications or computational intensive applications.
- The limited operational times: drones operability is strongly affected by power consumptions because this factor directly affects flight times. Large batteries grant extended flight times but, at the same time, they increase the overall weight, reducing the capacity for other payloads.

The extreme versatility we have described so far led to the development of a very wide range of different drones during the years. A 2011 classification, reported in 2.1, tries to characterize UAVs based on: operative range, flight altitude, flight duration and weight [11].

Such classification is the one usually taken into consideration by regulatory laws, distinguishing which drones can be piloted without any licence

Category	Operative Range [km]	Flight Altitude [m]	Flight Duration [h]	Weight [kg]
Tactical UAV				
Nano	< 1	100	< 1	$< 0{,}0250$
Micro	< 10	250	1	< 5
Mini	< 10	150 - 300	< 2	< 30
Close Range	10 - 30	3 000	2 - 4	150
Short Range	30 -70	3 000	3 - 6	200
Medium Range	70 - 200	5 000	6 - 10	1 250
Medium Range Endurance	> 500	8 000	10 - 18	1 250
Low altitude Deep Penetration	> 250	50 - 9 000	0,5 - 1	350
Low Altitudine Long Endurance	> 500	3 000	> 24	< 30
Medium Altitudine Long Endurance	> 500	14 000	24 - 48	1 500
Strategic UAV				
High Altitude Long Endurance	$> 2\,000$	20 000	24 - 48	12 000
Special purpose UAV				
Unnamed combat aerial vehicle	1 500	10 000	2	10 000

Table 2.1: Classification of UAVs based on operative range, flight altitude, flight duration and weight.

and which ones require a training course and a licence. Drones from *Nano*, *Micro* and *Mini* categories are the ones more involved in civilian applications, mainly for the affordable devices costs and operational costs. Also the limited flight altitude and the little weight allow to operate without strong regulatory constraints. Drones falling into these categories are the ones in scope for the purpose of this thesis work.

A more recent classification, realized in 2017, is instead focused on the capabilities the UAV owns rather than on the wide disparity of size and capacity [12]. This classification, reported in Table 2.2, is better at highlighting *what the UAV can do* and is thus convenient to understand when a specific UAV is suitable for a specific application.
Categories we are mainly interested in for the purpose of this thesis work are *Navigation*, *Sensors* and *Data*. The capability to pilot the drone using an onboard navigation system, in particular, is a basic assumption for the class of applications we are going to describe.

Category	Capability Description
Launch and recovery	The UAV owns or uses specific capabilities to insert itself into the airspace ("launch" phase) and/or capabilities to be retrieved by an operator ("recovery" phase)
Navigation	The UAV is able to receive real-time information from either an operator or from an onboard navigation system
Sensors	The UAV hosts technologies that detect and measures various physical properties
Data	The information collected by the UAV must either be stored locally or transmitted to a remote location
Stealth	The UAV is able to mask its presence in the airspace, challenging both radars and visual detection

Table 2.2: Classification of UAVs based on capabilities.

2.1.2 5G cellular network

The 5G cellular network, often just shortened to *5G*, is the 5th generation cellular network technology described by the International Telecommunication Union (*ITU*) in the IMT-2020 Standard.

Table 2.3 summarizes the minimum requirements for IMT-2020 5G candidate radio access technologies, according to [13].

Capability	Description	5G requirement
Downlink peak data rate	Minimum data rate technology must support	20 Gbit/s
Uplink peak data rate	Minimum data rate technology must support	10 Gbit/s
User experienced downlink data rate	Data rate in dense urban test environment 95% of time	100 Mbit/s
User experienced uplink data rate	Data rate in dense urban test environment 95% of time	50 Mbit/s
Latency	Radio network contribution to packet travel time	1-4 ms
Mobility	Maximum speed for handoff and QoS requirements	500km/h
Connection density	Total number of devices per unit area	$10^6/km^2$
Area traffic capacity	Total traffic across coverage area	10 Mbps/m^2

Table 2.3: 5G cellular minimum network requirements from IMT-2020 standard

5G is not only an incremental advance over the 4G technology but, exactly like the previous generations did, it is a major paradigm shift that includes very high carrier frequencies with massive bandwidths, extreme base station and device densities, and unprecedented numbers of antennas, as it is reported in [14]. However, unlike the previous four generations, it will also be highly integrative: tying any new 5G air interface and spectrum together with LTE and WiFi to provide universal high-rate coverage and a seamless user experience [14].

The high global interest in 5G research is fuelled by two orthogonal advancements: in one hand the technical aspects such as the expected end-user high bandwidths, low latency and the ability to support a very high number of devices. On the other hand, the functional features that 5G is going to

support, such as fixed-mobile convergence, device to device communication, and the compliance with the Open Access network architecture [15].

5G is expected to have a rapid spread in the following years, being all the major telecommunication carriers committed on a fast large-scale deployment. It is the perfect technological candidate for bringing connectivity to high-mobility robots like UAVs, given the widely extended coverage, the lightweight modems and the easy integrability with existing connectivity stacks currently supporting 3G/LTE mobile networks.

These are all reasons that make the 5G cellular network the perfect instance of high-performance network for all the examples and use cases later described in this thesis.

2.1.3 Multi-access Edge Computing

Multi-access edge computing, formerly mobile edge computing and often shortened to *MEC*, is a technology developed with the aim of reducing mobile networks load by locating storage and computing resources closer to clients, avoiding data to be uploaded and downloaded to and from devices and data centers connected to the internet [16]. This need is directly connected to the exponential growth of mobile traffic, mainly addressable to the changed habits of internet users, together with the spreading of the Internet of Things and machine-to-machine connections.
MEC is based on virtualization technologies and infrastructures to enable the deployment of applications directly at the edge of mobile networks, that are macro base stations eNodeB of the LTE cellular network o Radio Network Controller (RNC) of the 3G cellular network.

Moving applications closer to clients not only reduces the network load but also allows new forms of interactions. It enables users to benefit from the proximity of data sources, which leads to ultra-low latency and improved bandwidth utilization. To provide an example, measures read from a sensor will not need to hop across multiple datacenters spread in several countries, but it will be directly available to the application.

Besides the technical aspects, MEC addresses two distinct market needs at the same time:

1. The need of network operators to react to the growth of mobile traffic in order to maintain quality of service, but especially to keep generating revenues and to reduce costs.

2. The need of enterprises to engage customers in more efficient, secure, and low latency connections leveraging new forms of interaction.

MEC takes the form of an IT service environment that network operators are expected in the future to open to certified third parties in order to deploy personalized applications and address enterprise requests [17].
Applications will be able to leverage a wide set of APIs to access real-time information about the radio network status in order to further improve and personalize the user experience. MEC is recognized by the 5G Infrastructure Public Private Partnership, (*5G-PPP*), as one of the key emerging technologies for 5G networks, together with Network Function Virtualization and Software-Defined Network [18].

Figure 2.1, taken from [19], depicts the MEC framework and shows the involved high level functions.
The network layer represents the connectivity to local area networks, cellular networks, and external networks such as the Internet.
The host layer contains the Mobile Edge (shortened to ME) host entity and the related management entity. The ME host is further split into three different entities: ME applications, ME Platform and the virtualization infrastructure. The ME platform provides a collection of baseline functionality that are required to run applications and it enables them to discover, advertise, offer and consume services. Together with the virtualization infrastructure, the ME platform provides computing, storage and network resources to ME applications. These latter run as virtual machines on top of virtualization infrastructure, and they interact with the ME platform via the APIs

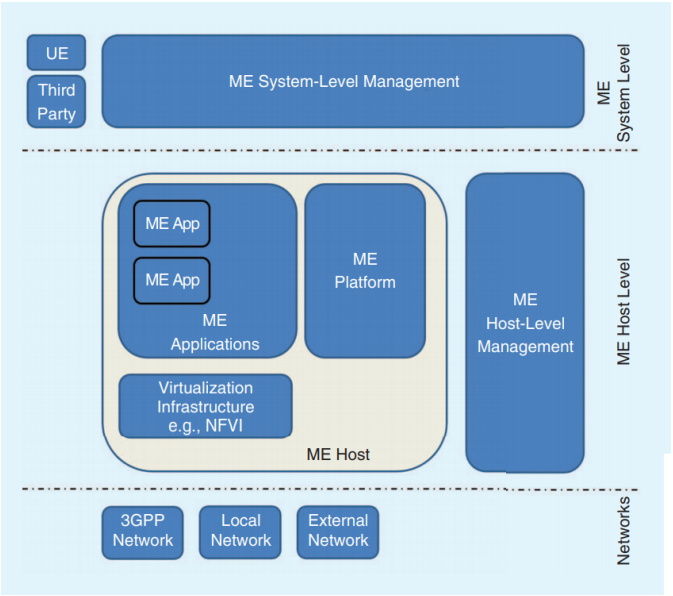


Figure 2.1: The MEC framework.

exposed by the platform itself. Applications can also offer services back to the platform that can further provide those services available to other applications.
The system layer contains the system-level management entity, which has the global visibility over the whole ME system.

Mobile Edge Computing is a technology that specifically targets mobile networks, but the concept of *edge computing* in the purpose of moving storage and computing capabilities closer to clients, in order to achieve the benefits of reduced latency and increased bandwidth and locality, is a concept that can be exploited outside the context of mobile networks. In this thesis work, different use cases will involve Mobile Edge Computing together with 5G, but everything that is going to be described in this document can be applied to any other high performance network and edge computing solution.

2.2 State of the art

2.2.1 Evolution of UAVs solutions

The research stream of software solutions for civilian UAVs progressively addressed problems of increasing complexity.
The first issue that was addressed was the autonomy of drones. At the very beginning, drones were piloted by a human being by means of a radio controller and the limited aid of basic flight controllers. When flight controllers became more sophisticated, being capable of autonomously takeoff, reach a series of locations (called *waypoints*), keep the altitude and land, radio controllers were replaced by ground stations. Ground stations are able to plan the mission and monitor the execution, with human intervention relegated to the handling of critical situations and failures only. Using ground controllers, it is now possible to execute more complex applications and also to build interactions with other systems. UAVs capabilities are incredibly extended but autonomy is not yet totally achieved: a connection with the ground controller is at any time necessary since no application logic is hosted directly on the drone, that is completely slave of the flight commands coming from the ground controller. Drones operative range is limited to the connectivity range of their ground controller. But the *range* is not the only constraint: it is not sufficient that a connection exists, but it also needs to be reliable and fast enough during the entire mission. An application coordinating multiple drones, at this point in the state of the art, needs to synchronize the relative ground stations, or to empower the ground station to control multiple UAVs simultaneously. Applications are strongly affected by these limitations: a data gathering mission, for example, was rarely able to transmit sensors readings in real time, while it was most likely to store them to be later dumped and analysed.

In this strive for autonomy, the next step was to move the application logic directly on the drone. This was possible also thanks to the miniaturization and cost reduction of computers, that could be carried onboard and directly connected to the flight controller, together with sensors and actuators. Applications can now be deployed directly into the drone, expanding the operative range to potentially unlimited. At this point, however, new constraints arise. First point: the computational power of computers that can be carried on board is limited, not comparable to the one provided by ground controllers, and the power they absorb affects the already little flight times. Plus, a connection with the drone is still required for all those missions that expect to monitor results in real time, or the ones that need some human intervention at some point. Still, synchronization and information exchange with other UAVs during the flight remains a problem.

To address these new constraints and needs, drones were connected to the internet. The spread and the significant performance improvements of wireless networks, and especially of cellular networks, made remote UAVs monitoring, control and cooperation possible. The operative range is no more constrained and synchronization inside fleets is now much simpler. This integration between internet and drones resulted to be particularly successful because drones joined the fast growing IoT ecosystem, and started to benefit from the researches and the improvements reached in other related fields.
The integration between UAVs and internet was so successful and the number of internet connected drones in the near future is expected to be so high that led researches to think about the *Internet of Drones*: "a layered network control architecture designed mainly for coordinating the access of unmanned aerial vehicles to controlled airspace, and providing navigation services between locations" [20].

The purpose of this section is to describe the most relevant solutions developed during the years, either as part of research projects or as part of commercial products. For each system we are going to highlight which topics they directly address among the ones described in the evolution process.

2.2.2 Robot Operating System

Robot Operating System, often shortened as ROS, is a suite of software frameworks and libraries for robot software development. ROS was born in 2007 at Stanford University [21] with the purpose of making a baseline system that would provide a starting place for others in academia to build upon. Despite his name, ROS is not an operating system in the traditional sense of process management and scheduling. It provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level

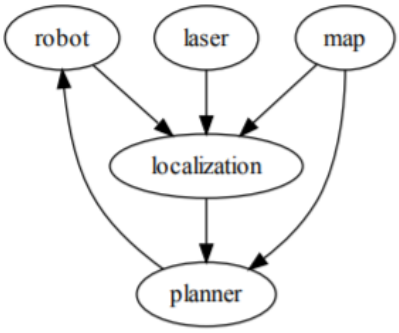


Figure 2.2: Example of a basic ROS proces, taken from [22].

device control and message-passing between processes.

The core of the ROS architecture is centered on processes: they are represented as graphs where computational centers are the nodes, and edges are called topics and represent information sharing routes among the nodes. Through topics, the nodes can share information about sensor data, control, state, planning, actuator, and application specific messages [22].
Around the core system, several packages of tools have been built to increase the capabilities of systems by simplifying and providing solutions to a number of common robotics development. Packages including common and widely adopted algorithms and tools are shipped included with the ROS distribution, while many other are developed by individuals, addressing very specific problems, are distributed through code sharing. This huge modularity, strongly designed upon open-source models, contributed to build a huge ROS ecosystem and to spread the diffusion of ROS in very heterogeneous application scenarios.

The scope of ROS is very wide: given its very low level nature, it can be employed as "robot middleware" upon which it is possible to build potentially any kind of application. In March 2011, a package for MAVLink compatibility was released in the ROS ecosystem, porting ROS on UAVs [23]. The benefit coming from the adoption of ROS in research works is that it allows focusing on a specific topic while exploiting his extraordinary wide set of already implemented tools. For example, researchers worked on drones interoperability through web services by means of REST interfaces: their choice of using ROS as underlying layer allowed their software solution to be insantly extended to the already available capabilities [24] in the ROS ecosystem. Similarly, the possibility of reusing components already implemented and tested by a very large community was crucial for researchers

working on the design and test of missions for a network of autonomous underwater vehicles [25].

2.2.3 IDrOS

IDrOS is a software platform developed by Daniel Cantoni as part of his master thesis "*System Support for Internet-connected Drones*" and research work at NESLab laboratory of Politecnico di Milano [26].
The main purposes declared in his work are:

- To fully integrate UAVs into the IoT context, enabling machine-to-machine interaction in order to eliminate the need of a human based control.
- To overcome the limitations of the classical *waypoints based* navigation in order to make UAVs capable of autonomous navigation.

From these purposes, three functional requirements are derived as driver for the entire research work:

1. Provide internet interfaces to control UAVs. Those interfaces should support different internet protocols in order to allow the integration of drones in a variety of applications.
2. Support an application model based on *Active Sensing* techniques in order to allow autonomous UAV navigation exploiting data read from sensors. This application model should abstract all the implementation details related to drone piloting.
3. To be compatible with different sensor models by enabling developers to write custom sensor drivers to be installed into the system.

As depicted in Figure 2.3, IDrOS architecture is split into three layers: *hardware abstraction*, *application logic* and *internet interface*.
Hardware abstraction This layer hides the implementation details related to the communication with the drone flight controller and with the sensors installed onboard. It provides high level interfaces to be exploited by upper layers.
The *sensor* component provides capabilities to:

- Provide a standardized access point to sensors.
- Discover installed sensors at runtime.
- Retrieve values read from the sensors, hiding the implementations details of each specific sensor.

The *drone* component provides capabilities to:

- Pilot the drone, hiding the implementation details relative to the specific communication protocol.
- Access to telemetry data coming from the flight controller.

Application logic This layer provides the functionality needed to run applications. It is further split into four components:

- *Modules Manager* This component exposes functionality to let the user upload, list, delete applications. Applications can reference to either *sampling applications* or *data analysis applications*, actually the two types of application supported by the implementation.
- *Mission Manager* This component contains the biggest part of the application logic needed to handle the flight, launch and control applications and acquire data from sensors.
- *Sensors Manager* This component offers an even higher level of abstraction to interact with the sensors installed on board, actually wrapping the *sensor component* from the *hardware abstraction* layer.
- *Fail-Safe Manager* This component holds all the application logic in charge of handling the drone when some error occurs at any level in the software or the drone flight is compromised by physical events. The main goal of this component is to offer the functionality to safely land the drone.

Internet Interface This layer holds all the capabilities needed by the user or by other machines to communicate with the UAV and the IDrOS instance. It supports several bindings to different internet protocols and it is structured to easily support the addition of new ones.
The binding provided in the current IDrOS implementation are CoAP, Constrained Application Protocol, and MQTT, Message Queue Telemetry Transport.

IDrOS supports two deployment modes:

- Directly into the drone by means of a companion computer connected to the flight controller. This is the deployment mode that best fits with the purposes of the research work because the UAV is completely autonomous. Limitations to this approach come from the available computational power on the drone, that should be sufficient to run iDrOS and given tasks.

- Into the ground-based controller. In this deployment mode, the entire mission is managed from the ground and flight parameters only are sent to the drone. This overcomes limitations coming from computational capacity, but it is required that a fast and reliable communication channel (e.g. radio) is available. Drone operability is limited to the coverage of such communication channel (e.g. for radio it is limited to few kilometres).

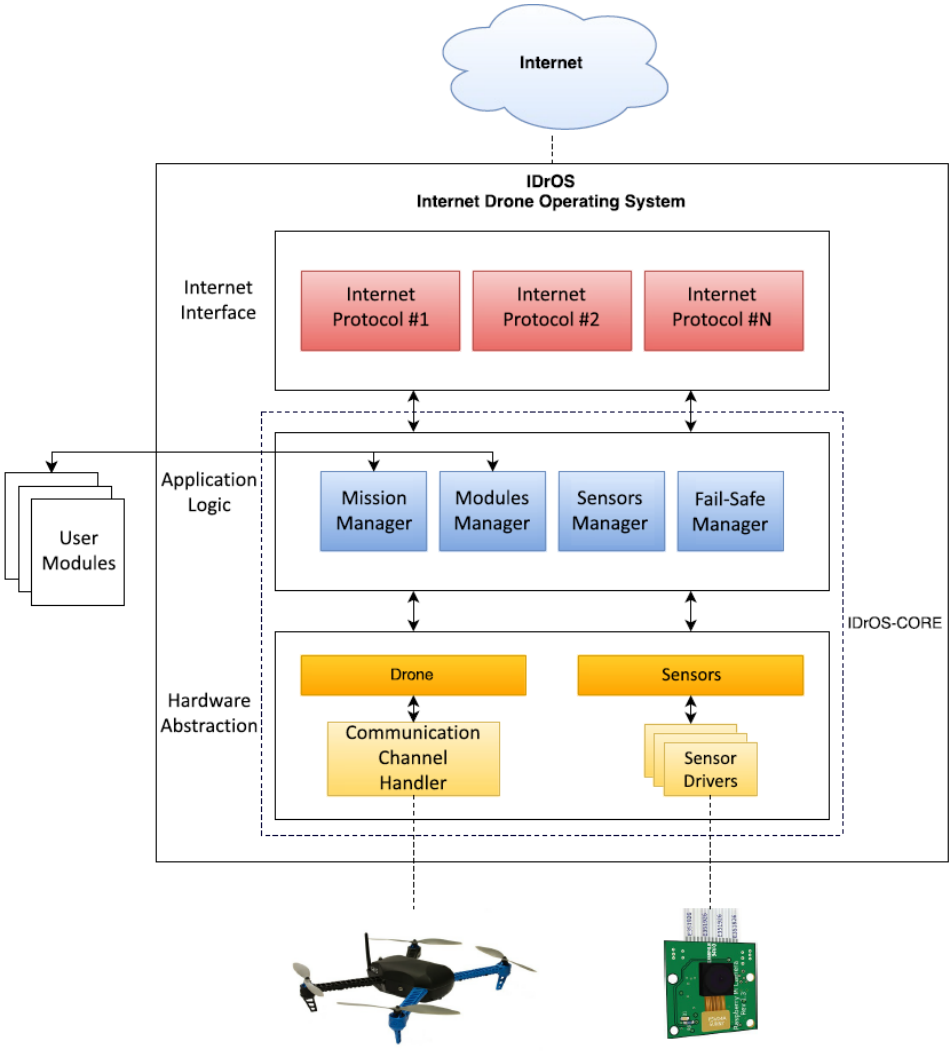


Figure 2.3: IDrOS architecture.

2.2.4 DroneKit

DroneKit is an open source SDK for Python and Android released by 3D Robotics, Inc in 2015 [27].
The main issue addressed by DroneKit is to abstract several low level aspects of MAVLink, which is one of the most used protocols to communicate with small UAVs.
DroneKit provides high level interfaces for connecting, monitoring and controlling a vehicle [28]. Using such interfaces, researchers and developers can easily implement the parts of their applications dealing with path planning, autonomous flight and telemetry monitoring. The reduced effort to be spent in controlling the drone allows developers to focus more on the core part of their applications.

DroneKit became popular especially as part of the drones Do-It-Yourself community. By abstracting low level concepts of MAVLink, DroneKit makes very easy to pilot the drone without advanced aerodynamics knowledge or flight control knowledge. Basic commands such as *takeoff*, *land* and *flight-to* are sufficient to handle the movement of the drone, without the need of understanding how DroneKit translates these instructions to the flight controller, or how the flight controller translates these inputs to the rotors.

DroneKit is distributed in the form of two distinct SDKs providing the same functionalities in two different deployment setups:

- *Python SDK* This SDK is designed to deploy applications directly on the drone, by means of a companion computer directly connected to the flight controller. This deployment setting is designed to make the drone fully autonomous since no connection is needed with ground controllers.

- *Android SDK* This SDK is designed to develop mobile applications to control the drone. Only flight information and commands are sent to the flight controller by exploiting remote MAVLink interface capability. This deployment setting constraints the operative range of the drone since a low latency, reliable connection should be in place between the mobile device and the drone.

2.2.5 Flytbase

FlytBase is a commercial platform provided by FlytBase, Inc that includes hardware and software solutions to allow easy deployment of intelligent drones, connected to cloud-based business applications.
FlytBase allows developers to build new applications using drones control APIs and cloud APIs.

FlytBase provides also a set of already implemented applications and the possibility to get consulting during the development of new ones.
Typical FlytBase applications are:

- Warehouse Management;
- Security & Surveillance;
- Emergency Response;
- Delivery;
- Fleet Survey and Mapping.

FlytBase is made up of two main architectural components:

- *FlytOS* It is an operating system built on ROS (*Robot Operating System*) and Linux to be installed directly into a companion computer connected to the flight controller. It allows to pilot the UAV supporting different flight stacks, to manage a wide range of payloads and onboard sensors. It also offers standard OS capabilities such as logging, updates management and user authentication. FlytOS is designed to be natively integrated with FlytBase cloud capabilities.
- *FlytCloud* It provides APIs for real-time access and control to drone navigation, telemetry and payload. FlytCloud offers Artificial Intelligence capabilities, such as real-time object classification, object counting and patterns change detection, as well as data analytics capabilities, such as OrthoMosaic mapping, 3D reconstruction, Normalized Difference Vegetation Index (*NDVI*)

The main customer target of FlytBase is constituted by companies; for this reason they provide ad-hoc integration with several enterprise solutions such as Microsoft Dynamics 365, SAP, and Salesforce.

2.2.6 Gap in the State of the Art

As it is extensively described in Section 1.1, the purpose of this thesis is to investigate the integration of UAVs and high-performance networks in two separate directions: to overcome historical UAVs limitations and to further extend their capabilities.
Following these two directions, we can state that iDrOS and DroneKit mainly focus on the first one, that Flytbase focuses on the second one, and that ROS is so general purpose that it addresses both directions.

Both DroneKit and IDrOS main purpose is to make drones autonomous by enabling application programmers to write and deploy applications onto

them. Applications can leverage interfaces to pilot the drone and also to manage the payloads. Despite DroneKit, IDrOS provides specific capabilities to manage sensors installed on board with the purpose of enabling application developers to implement active sensing techniques. They both offer functionality to remotely monitor the drone, by acquiring the telemetry and other parameters. DroneKit and IDrOS also enable machine-to-machine interactions by creating remote interfaces that can be leveraged to control and monitor. However, none of them offers any capability to support coordination or inter-drone communication: any effort in this sense is entirely left to the application developer.
DroneBase adds to the aforementioned capabilities all the benefits coming from cloud connectivity, introducing new concepts such as scalability and elasticity. DroneBase is built upon ROS, but it is a commercial closed-source solution, and it mainly targets companies. It can be looked at as a reference to investigate how UAVs applications are employed in business, but it can not be taken into consideration for research purposes.
Talking about ROS, its scope is very general since it targets the entire spectrum of robots. It could virtually support each kind of UAVs application. Using ROS, it is very easy to reuse components written for other research areas, but as a drawback this extreme generality requires a lot of effort to develop specific and scope optimized solutions.

None of the analysed solutions was built keeping high-performance networks at the core of the design process. This means that even enabling them to such an improved network connectivity will not fully materialize the potential benefits. All of them allow building custom applications but the programming models and the architectures they are built upon are not adequate to fully enable new application scenarios that involve ultra-low latency, high bandwidth and extreme reliability.

In addiction to this, the adoption rate of the state-of-the-art systems we have described so far is very low. Even if advanced solutions are available, the vast majority of real applications do not leverage them. Many real-world scenarios still rely or require a human pilot on site. Even *beyond line of sight* piloting is still very far from being adopted. We can conclude stating that there is a lot of work to be done in order to fully discover and exploit the potentiality of drones.

Chapter 3

Exploiting High-Performance Networks

3.1 Motivating Scenarios

The purpose of this section is to describe some real world use cases showing how the integration between high-mobility robots and high-performance networks leads to new application scenarios. In order to describe real use cases, we are going to use UAVs as instances of high-mobility robots, and 5G as instance of high-performance networks. This specification, however, does not affect the generality of the discussion: everything that is described in the following sections can be abstracted and related to any type of robot and network.
From these use cases, we are going to derive the capabilities and the functionality that a software platform must provide for this integration to be successful. These capabilities serve as input for the formal definition of the programming model, provided in Section 3.2.

3.1.1 Sensors Sharing

In this use case, two drones are involved in a search and rescue mission over a mountain area recently involved in a snow avalanche. Several people may be involved in the disaster and the purpose of the UAVs crew is to find trapped skiers in order to report their presence to the rescue teams. One of the two drones is equipped with a radio receiver to scan for signals coming from the avalanche beacons. Avalanche beacons are safety equipment for skiers that actively transmit a radio signal at a specific frequency. The second drone is equipped with an infrared camera in order to identify heat sources that may come from trapped skiers. The two drones are equipped with two different companion computers running two distinct applications that control their flight path. The two drones are connected via 5G network,

and the applications are able to synchronize and to mix data coming from both the radio receiver and the infrared camera with the purpose of finding the best path. First, the two UAVs follow the increasing gradient of the radio signal, then the area is scanned with the infrared camera.

The software platform supporting such mission must allow the UAVs to *share their sensors*, that is to allow a specific client to read the data coming from a sensor physically connected to a different host. Exploiting the low latency of high-performance networks, data can be exchanged in near real time and clients experience no difference in reading from a local sensor or from a remote sensor. Moreover, thanks to the high bandwidth, a huge number of sensors and clients can push and pull data simultaneously without affecting the quality of the service. The size of each reading can be bigger and still transferred in a short time, effectively supporting a higher resolution for the measurements.

The possibility to share sensors between clients is an enabling feature for:

- Creating a specialized crew of drones in which each member can contribute to the mission with specific hardware equipment.
- Optimizing the overall power consumption by not duplicating the hardware.
- Reducing the costs, since specific hardware and software components can be deployed on a single client only and shared among the others. This reduces purchase, installation, configuration and maintenance costs.

3.1.2 Best Sensor Selection

In this scenario, a crew of UAVs is used to supervise a large distribution warehouse. Each drone patrols a specific path, and they take turns in order to overcome battery recharging times. When a threat is identified additional drones are sent to the place in order to monitor a wider area. The warehouse is composed by an internal, covered part, and an outside part. The same drone may patrol both the inside and the outside of the warehouse. Each UAV is equipped with a standard GPS sensor, but since the precision in the inside part of the warehouse is not high enough to support the patrolling mission, they are able to dynamically switch from the GPS to an indoor positioning system that is installed for the purpose. As soon as the precision of the GPS returns above the minimum threshold, the navigation system can switch back to the GPS sensor.

The purpose of this use case is to show how a software platform supporting

this scenario must provide functionality to switch the provider of a specific measurement at runtime, based on user-defined metrics and logics. In the use case provided, the switch from one positioning system to the other is triggered by the accuracy value of the GPS sensor.
The software platform must allow an application developer to implement custom logic to trigger the switch among quantity providers: an example may be to prefer a remote sensor and to accept a lower accuracy when the battery is low, and to prefer a local sensor and a better accuracy when the power level is high. Providing high level APIs to application developers to build that kind of custom logics allows to adapt this capability to any specific mission scenario.
The examples provided so far describe a switch between a local sensor (i.e. a sensor physically connected to the drone) and a remote sensor (i.e. a sensor exploited remotely), but the same mechanism may be implemented for local to local sensors switch, or for remote to remote sensors switch.

3.1.3 General Purpose Sensing

A drone is used to produce aerial shooting of a football match. The images taken with the onboard camera offer to the viewers a unique point of view. However, it is really difficult for a human pilot to keep the camera steadily aligned with the position of the ball and to follow the rhythm of the match. The application logic that pilots the drone and controls the camera is able to remotely access the data coming from the tracking cameras already present in the stadium. Knowing in real time the position of the ball and its estimated trajectory allows to correctly position both the camera and the drone itself.

The purpose of this use case is to show that the remotely exploited sensors are not necessarily part of the mission as it happens in the first use case. The purpose is to extend the range of this capability by allowing clients to read data from sensors that are deployed for a different specific mission, but that are at the same time available to support different applications. This allows sensors to become *multipurpose*, providing the ability to serve applications different from the ones for which they were originally intended for. In the use case just described, the drone is leveraging some sensors that it knows for sure are present (the ball tracking cameras in the stadium). The UAV knows exactly how the sensors work and how to interact with them. This scenario can be further extended by enabling clients to discover sensors at runtime, as it happens in the use case below.

A wide countryside area is facing air pollution issues due to malicious and illegal spill of toxic waste. People living in the area started to install sensor stations near their houses, similar to home weather stations. These sensor

stations are able to detect the pollution agents in the air and to publicly expose the data they are collecting. UAVs flying over the area can remotely access these data and use them to search for the illegal spills, by following the increasing gradient of pullution agents concentration.

The difference with respect to the first use case is that now the number, the type and the presence of the sensors is not a priori known by the application. A software platform supporting this kind of applications must enable runtime discovery of sensors and a specific mechanism to understand the characteristics of unknown sensors, such as the unit of measurement and the accuracy they provide.
In such context, sensors may appear and disappear at any time, and most important, they are in relation with the position of the client. The discovery mechanism must not apply to the entire population of the sensors, but it must be limited to a specific area. In this sense, the software platform must be robust with respect to the disappearance of a sensor, because it goes away or because the client moves too far from it. On the other side, new sensor may appear at any time.

3.1.4 Computer Vision for Assisted Navigation

A UAV is used to perform last mile delivery in a large city. While flying among buildings, the GPS signal is not always present or accurate enough to guarantee the precision needed for the autonomous flight. When the GPS signal is not sufficient, the application can switch to a computer-vision based navigation system. This navigation system is highly computational intensive and its execution latency limits the max speed that the drone can reach and the overall mission execution time. The application is able to offload the computation of such navigation system to an external host that is more suitable for computation, in order to reduce the latency. Whenever the network conditions degrade and it becomes convenient to run the navigation system onboard, the application is able to inload it any time. As soon as the GPS signal is accurate enough, the application switches back to this preferred navigation system.

This use case requires a software platform able to offload parts of the application logic to a remote host and to inload it again. It must support a peer-to-peer code exchange and a remote invocation mechanism. The straightforward purpose of such capabilities is to improve performance by moving complex computation to more suitable hosts. Other scenarios may involve the exploitation of remote storage capabilities, the execution of critical parts of the application in a secured hosts, or the possibility to reduce battery consumption by reducing the CPU load. For this kind of capabilities, all the improved characteristics of the high-performance networks are key

enabling elements: the low latency guarantees that the invocation requests and results are quickly transferred among the clients; the high bandwidth allows offloading large code parts or to perform a remote invocation with a huge number of input parameters; the high reliability guarantees that no invocation requests and no execution results are lost during the transfer from one client to the other.

3.1.5 Peer to Peer Synchronization

An open land space is used for takeoff and land operations of a huge traffic of drones. No central authority exists to control the traffic of the area. A distributed synchronization mechanism exists among the drones in order to avoid collisions. When a drone wants to takeoff or it is approaching the space to land, it computes a desired trajectory and publishes it over a communication bus. The trajectory is transferred to all the other drones in the area in order to allow possible colliding drones to change their plan. When two or more drones identify a possible collision, they open a dedicated channel over the communication bus and synchronize to change the trajectories. This collision avoidance mechanism does not keep a central state, but the knowledge needed to avoid crashes is kept in each client in the area.

This use case requires a software platform that is able to support the communication of clients based on a communication bus, where messages are sent over specific channels. Clients are able to create, join and leave channels at runtime. For this scenario, not only the ultra-low latency of high-performance networks is relevant to ensure rapid messages exchange, but also the high reliability ensures in-order delivery and that no messages get lost. This last point enables this mechanism to be leveraged in critical parts of the missions.

3.2 Programming Model

In this section it is described the programming model we designed to specifically target the integration with high-performance networks. As it is mentioned in Section 1.1, the purpose of this integration is to overcome classical limitations of UAVs and to expand the class of applications they can be involved in.

From the use cases outlined in Section 3.1 we derived three macro-functional areas that drove the design of the programming model: *Remote Sensors* addresses the functionality described in Section 3.1.1, Section 3.1.2 and in Section 3.1.3; *Computation Offloading* addresses the functionality described in Section 3.1.4; *Communication Bus* addresses the functionality described in Section 3.1.5.

Before outlining the programming model in detail, it is now useful to clearly state which are the classical limitations that we want to overcome as well as which are the new kinds of applications towards which we want to extend the usage of drones to.
As it is described in Section 2.2, there is a trade-off between the degree of autonomy that a UAV can achieve in executing its mission and the complexity of the mission itself. A higher degree of autonomy requires that the drone is not dependent on any ground station and that the applications are directly loaded onto it. Being applications directly run onboard, the complexity that can be managed is strictly dependent on the computing platform that is installed on the drone. Typically, the computing platforms available for civilian drones offer low power consumption processors with small computational power. A similar argument can also be extended to storage capabilities, which are strictly related to hardware performance. The programming model we developed proposes abstractions to overcome these limitations by leveraging an approach inspired from the cloud computing and the mobile code paradigm, that allows to offload and inload application parts at runtime towards suitable hosts.

From the analysis of the state-of-the-art, it is also clear that current solutions do not provide specific capabilities to support drones coordination and cooperation. Leveraging the high reliability and the high performance of the networks we are focusing on, coordination and cooperation can be fully exploited as parts of the mission, creating new operative scenarios and extending the application area of UAVs. The programming model that we propose wants to address this gap by providing capabilities that allow drones to share physical resources such as sensors and actuators. In addition to physical resources sharing, the programming model also focuses on providing abstractions to reliable and real-time information exchange, with the purpose of providing specific capabilities for drones coordination.

The last gap we targeted to overcome is to immerse drones and drones applications into the real world. This means to extend the logical boundaries of the mission they are running by making them able to exploit resources that were deployed for general purposes and that exist before and after the duration of the drone mission. The way this gap is addressed is by providing capabilities to register, discover and query such shared resources at runtime.

In order to develop a programming model that can be translated into a fully functional software platform, the last section of this chapter is focused on the flight control, the mission management and the remote drone interface. These topics do not represent the core contribution of this thesis work, however they contribute to the full picture.

3.2.1 Sensors

This part of the programming model is focused on enabling UAVs to interact and manage sensors.
The first aspect we addressed in designing this programming model is the problem of sensors description and observations description. This is crucial to allow clients to interact with sensors that are discovered at runtime. Then we focused on the way clients should interact with sensors in order to read their description and to request for observations. The last aspect we addressed is the one directly connected with the exploitation of high-performance networks: sensors sharing among clients.

The following requirements are the ones to be satisfied by the programming model:

- *FSR1* It must be possible to describe a sensor. Clients must be given the capability to interact with unknown sensors by gleaning their characteristics from their descriptors.
- *FSR2* Clients must be able to interact with different types of physically connected sensors.
- *FSR3* Clients must be able to expose some physically-connected sensors to be used by other clients.
- *FSR4* Clients must be able to query the list of sensors exposed by other clients.
- *FSR5* After their discovery, clients must be able to use sensors shared by other clients.
- *FSR6* Clients must be able to interact with different types of remotely connected sensors.

Several approaches have been proposed to describe sensors, and the vast majority of them relies on ontologies [29], as direct derivation of knowledge representation theory. This approach is capable of effectively describing a vast diversity of sensors and their observations but it results to be too formal for an operative usage. While ontologies are perfect metadata for large datasets to be stored and shared, they do not well adapt to the needs of real-time discovery, query and usage. A perfect descriptor in this application scenario must be able to represent both static properties — i.e. those that do not change over time, such as the identifier of the sensor — and dynamic properties — i.e. those that may change over time, such as the accuracy of the sensor.

The approach we propose for sensors description in our programming model

is based on a series of categorizations. We identify as *push* sensors those sensors that autonomously produce an observation at a given time rate; we identify as *pull* sensors the ones for which each observation is the result of an explicit request to the sensor.
The description of both push and pull sensors is expressed by means of *properties*. They can be categorized as: *static properties* that are the ones not changing over time, and *dynamic properties* that are the ones whose value may vary during the lifetime of the sensor.
The last categorization to be made is between *mandatory* properties and *optional* properties. Mandatory properties allow us to define a set of minimal information that should be provided to describe a sensor, while optional properties provide enough flexibility to capture differences among sensors and also to implement application-specific use cases.

The following list contains the mandatory properties we identified and their description:

- *Identifier* This property is used to identify the sensor.
- *Type* This property can hold "push" or "pull" in order to identify the type of sensor.
- *Interval* For push sensors, this property represents the frequency at which new observations are produced, for pull sensors it represents the time interval between the observation request and its effective availability.
- *Quantity* This property indicates the quantity measured by the sensor.
- *Quantity Unit of Measure* This property indicates the unity of measure of the quantity measured by the sensor.
- *Accuracy* This property represents the accuracy of the sensor at the time of the request. It can be either static or dynamic.
- *Accuracy Unit of Measure* This property indicates the unity of measure of the accuracy.
- *Accuracy Type* can hold "static" or "dynamic" to indicate whether the accuracy of the sensor is the same over time or it may vary.

This way of describing a sensor satisfies our first requirement FSR1.

To address requirement FSR2, sensors must be accessed through a common interface in order to hide every implementation detail related to hardware or software characteristics of the sensor itself. Two are the methods exposed by this interface:

- *read* For push sensors, it retrieves and returns the last observation produced by the sensor, without requesting for a new one; for pull sensors, it requests a new observation and returns it as soon as it is available.

- *describe* Returns the set of properties defined as descriptors for the sensor.

Using this interface, clients are able to use sensors without effectively being aware of their characteristics: they are just providers of given quantities, accessed through the *read* method. Also the "push" or "pull" nature of the sensor is hidden behind this method. All the other information about the sensor can be gleaned by looking at its descriptor retrieved through the *describe* method.
This interface is implemented by the *sensor driver*, which is the piece of software intermediating the interaction with the sensor. The sensor driver is installed into the client to which the sensor is physically connected. We call the client into whom the sensor driver is installed the *host* for that specific sensor.

To address requirements FSR3, FSR4 and FSR5, clients are able to expose some of their physically-connected sensors to make them available for other clients. When a sensor is remotely exposed, it continues to be exploitable by the client to which it is connected. At the same time, other clients are able to discover it and access its observations.
Clients are able to query the set of all the remotely exposed sensors. Queries are based on one or more properties from the sensor descriptor. Using the sensor identifier and the identifier of its host, any client is able to remotely use that sensor.
We identify as *local sensors* the ones that are physically connected to a client, and we identify as *remote sensor* the ones that a client accesses remotely and that are phisically connected to a different host. Figura 3.1 reports an example of this taxonomy.

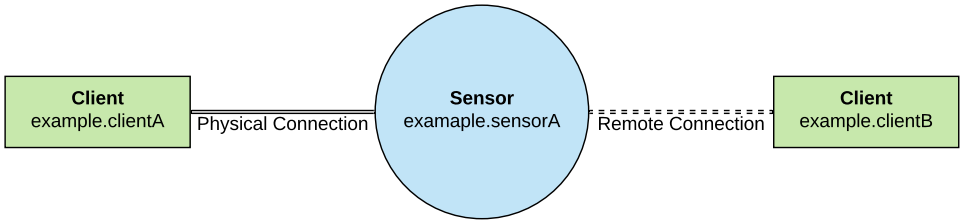


Figure 3.1: Taxonomy of sensors sharing. Sensor *example.sensorA* is physically connected to client *example.clientA*, which is the sensor host for this sensor. Client *example.clientB* remotely exploits *examples.sensorA*, which is a remote sensor for this client.

To address requirement FSR6, clients must interact to remote sensors using a common interface that hides all the details of the remote connection. This interface exposes the exact same methods exposed by the sensor driver:

- *read* For remote push sensors, it retrieves and returns the last observation provided by the sensor; for remote pull sensors, it requests a new observation and returns it as soon it is available.
- *describe* Returns the set of properties defined as descriptors for the remote sensor.

This interface is implemented by a *Remote Sensor driver*. It works as proxy for all the remote sensors and implements all the information exchanges needed between the client and the host of the remote sensor. The methods exposed by this proxy are identical to the ones exposed by the local sensor driver. In this way, there is no difference from an application logic point of view in accessing a remote or a local sensor through their driver.

It is clear that in the setup described so far it is possible that several sensors are able to measure the same quantity for a client, being them physically connected to it or remotely accessed. To fully enable the *best sensor selection scenario* as it is described in the use case in Section 3.1.2, we decided to further abstract the interaction with the sensors by means of a component called *Sensors Manager*. The purpose of this software component is to implement a comparison between the sensors providing the same quantity.

When a measurement is requested to the Sensors Manager, three possible behaviours are possible:

- *All* In this working mode, only the measurement from the best sensor is returned as result of the request.
- *Conservative* In this working mode, the best provider is not computed for each request, but it remains cached until some specific event is triggered.
- *Exclusive* In this working mode, the application developer explicitly specifies the provider of a quantity.

The first working mode ensures that the best provider is selected every time a measurement is requested, but it is the one requiring more efforts since all the providers must be queried every time. This working mode best fits in contexts where the accuracy of providers is expected to change very frequently. The second working mode best fits situations where the accuracy of sensors is expected to remain quite constant over time, since the provider of a quantity is computed only after specific events. In this case, only one sensor is queried when a request is issued, but it may happen

that a suboptimal measurement is returned until the best provider is not computed again.
The working mode can be set per quantity, meaning that different quantities can be retrieved using different working modes, and it must be possible to change it at runtime.

3.2.2 Computation Offloading

This part of the programming model is focused on enabling clients to offload parts of their application logic to be executed on a remote host. Starting from the scenario described in Section 3.1.4 we derived the following functional requirements:

- *FCOR1* Clients must be able to offload one or more functions.
- *FCOR2* Clients must be able to select the host towards which the functions are offloaded.
- *FCOR3* Clients must be able to inload functions that were previously offloaded.

The approach we propose in our programming model starts from the concept of *mobile code* which is largely adopted in distributed environments [30]. The underlying concept is the capability to move the code to be executed among hosts, together with its execution stack. However, while the full code mobility paradigm allows a code to be exchanged among peers multiple times, in the programming model we propose the code exchange is supported only on a peer-to-peer basis. This means that when a function is offloaded towards a remote host, it can be inloaded again only from that specific host.
We can distinguish two specific roles in the process:

- *Offloader* Is the client moving out the code.
- *Remote Host* Is the client receiving the code, and executing it upon request.

This restriction to the full code mobility paradigm is due to the high complexity needed to support the mobility among multiple peers, which is not relevant for the requirements we want to satisfy. The main purpose of computation offloading in the context we are analysing is to move the execution towards hosts where more resources are available, and such hosts are likely to be stable and do not change frequently over time.

Offloading and inloading operations are handled by a dedicated component. This component is in charge of handling the entire lifecycle of an offloaded function, from the moment the offloading is invoked to the moment that

the function is inloaded again. When the offloading request is issued, the function source code is sent to target host. Every invocation the to function is intercepted and issued to the target host, together with the possible input parameters. Once the execution on the target host is completed, possible return values are sent back to the invoking host. Possible errors and exceptions must be propagated back too. This effectively mixes the code mobility model with the Remote Method Invocation model, and it is an additional reason to the restriction of the full code mobility paradigm, since complex mechanisms must be put in place to identify where to invoke a function once it is has been moved among multiple peers.

In order to support a wider range of application scenarios, an additional requirement must be satisfied:

- *FCOR4* Both stateful and stateless function must be supported.

The execution of stateless functions depends only on possible input parameters and nothing else. Stateful functions, instead, require keeping track of additional information that forms the state of the function. This information must be transferred from the local host to the target host when the function is offloaded, and transferred back from the target host to the local host when the function is inloaded again.
In the programming model we propose, the variables forming the internal state of a stateful function must be explicitly marked in order to identify them as part of the set of information that must be transported back and forth.

In a context of extreme client mobility, like the ones in which UAVs usually operate, losing connection to target host means also to lose the internal status of offloaded stateful functions. To overcome possible issues coming from this possibility, two offloading modes of stateful functions are supported in the programming model:

- *State Saver* In this offloading mode, every time the function is invoked on the target host, the internal state of the function is sent back to the invoking host, together with possible return values or errors.

- *Transfer Once* In this offloading mode, the state is transferred back from the target host to the invoking host only when the function is inloaded back.

The first working mode is designed to ensure the maximum operability in poor network conditions scenarios, or when hosts are most likely to appear and disappear. As a drawback, it requires more effort related to the overhead of transferring the state back after each invocation. The second working mode is more conservative and requires less effort, but it is more suitable

to failures due to interrupted connection between the invoking host and the target host.
In the programming model we propose, no explicit management of failures is in place, except for a basic timeout mechanism, since the intrinsic reliability of the high-performance networks is sufficient to limit the occurrences and the complexity of failures.

3.2.3 Communication Bus

This part of the programming model is focused on enabling clients to exchange information and messages. Exploiting the improved characteristics of high-performance networks, applications can leverage information exchange between clients also for critical parts without worrying about reliability of the exchange mechanism.
We identified a mono-directional multi-recipients information exchange mechanism based on the publish-subscriber paradigm which can be formalized in the following functional requirements:

- *CBFR1* Clients must be able to instantiate communication channels.
- *CBFR2* Clients must be able to join communication channels instantiated by other clients.
- *CBFR3* Clients must be able to leave a communication channel they previously joined.
- *CBFR4* Clients must be able to send messages in the channel.
- *CBFR5* Each client in the channel must receive messages sent by other clients in the channel (except for the sender itself).
- *CBFR6* No messages are dropped by the communication channel.
- *CBFR7* Messages are received by each client in the exact order they are sent.

To the core requirements listed above, we decided to add an additional optional one:

- *OPTCBFR1* Clients must be able to limit the access to the channels they create.

The approach we decided to formalise for our programming model is similar to what is described by the MQTT protocol [31], which is widely used for sensors communications and mobile applications. All these requirements are satisfied by the functionality provided by a dedicated component. This component implements a specific interface providing the following methods:

- *create* To instantiate a new communication channel.
- *join* To join an existing communication channel.
- *send* To send new messages over the channel.
- *getMembers* To retrieve the list of clients in the channel.
- *incomingMessageCallback* Invoked when a new message is sent to the clients.

To address requirement *OPTCBFR1* two additional methods must be provided:

- *create_whitelist* To instantiate a new communication channel together with the list of nodes that can join it.
- *create_blacklist* To instantiate a new communication channel together with the list of nodes that can not join it.

In order to address requirements *CBFR6* and *CBFR7* we entirely rely on the characteristics of the high-performance network to which clients are connected to. No specific retransmission controls or error correction mechanisms are in place. The purpose of this choice is to keep the end-to-end latency of message exchange at the minimum by not introducing the overhead needed by such mechanisms.

3.2.4 Flight Control and Mission Management

This part of the programming model is focused on providing access to the flight controller, on offering capabilities for mission management and on exposing an interface that is remotely accessible to manage several aspects of the drone. What is described here is mostly revisited starting from the programming model proposed by Daniel Cantoni in his master thesis [26]. These topics are not part of the core contribution of this thesis work. The flight controller, for instance, is specifically needed for UAVs and does not fully apply to other high-mobility robots, which need a similar but different component to move on the ground or in the water. For this reason, this part of the programming model needs to be revised when targetting different robots.

The access to the flight controller must be intermediated by a high-level interface in order to hide specific details related to the type of drone (e.g. multi-rotor, fixed wing, single rotor, etc) but also specific details related to the model of the flight controller installed onboard.
This interface must provide methods to both send commands to the flight controller and to retrieve the status of the drone from it.

The methods to control the drone include:

- The command to arm and disarm the drone;
- The command to take off and reach a given altitude from the ground;
- The command to land;
- The command to reach a specific position specified by its latitude, longitude and altitude;
- The command to switch among the flight modes supported by the flight controller.

The methods to retrieve the status of the drone from the flight controller are:

- Retrieve the drone speed in x, y and z axis;
- Retrieve the drone ground speed and air speed;
- Retrieve the orientation of the drone;
- Retrieve the status of the GPS signal;
- Retrieve the current flight mode;
- Retrieve the current location expressed as latitude, longitude and altitude.

All these flight control commands can be programmed, together with sensors interactions, communication bus capabilities and computation offloading, to create a *mission*.
The *mission management* part of the programming model is focused on providing the functionality needed to:

- Store missions into the drone;
- List all the missions stored into the drone;
- Run a specific mission;
- Interrupt the execution of the mission.

Specifically to the last point, we categorize between *cyclic* missions, that are the ones that run in loop until they are explicitly stopped, and *acyclic* missions, that are the ones with a predefined end. Only one mission at a time can run on the UAV.

The last part of the programming model to be described is related to the functionality needed to remotely access the drone via internet. Remotely accessing the drone is needed to both control it and monitor it. This also allows the drone to be integrated inside the IoT ecosystem, enabling machine-to-machine interactions that would not be possible otherwise.
Through this interface, it must be possible to:

- Install, uninstall and list sensor drivers;
- Upload, delete, list missions;
- Start and stop a mission.

As per the original IDrOS programming model, several protocols must be available to access such interface. This topic is further detailed from an architectural point of view in Section 4.2.1.

Chapter 4

New Generation IDrOS: Architecture

4.1 Overview

In this chapter it is described the architecture of *New Generation IDrOS* (shortened to NG IDrOS), that is the software platform we have designed to support the programming model that is described in Section 3.2. This software platform is an evolution of the one originally developed by Daniel Cantoni in his master thesis [26], that is presented in Figure 2.3 and described in Section 2.2.3. The software platform that he described, designed and implemented is mainly focused on integrating drones into the IoT ecosystem and in supporting active sensing techniques. From the original IDrOS, we inherit and enhance the parts addressing the internet interface, the flight control and the mission management. The new parts we designed address the functionality directly related to the exploitation of high-performance networks.

We decided to split NG IDrOS into two distinct software components:

- *NG IDrOS Drone* This the part of the platform is directly deployed onto the drone. It works as interface towards the hardware and offers capabilities to run user-written missions.

- *NG IDrOS Central* This part of the platform works as central node between all clients for coordination and synchronization purposes. It can either be deployed on a drone, on a dedicated host or into the cloud.

We will describe the reason behind this split and both the two parts in the following sections. The purpose of designing this architecture is to provide a bridge between the programming model that is described in Section 3.2 and the implementation that is described in Section 5.

As it is shown in Figure 4.1, the architecture of NG IDrOS Drone is split into three layers: *Remote Control Layer*, *Application Logic Layer* and *Connection Layer*. Similarly, the architecture of NG IDrOS Central is split into two layers: *Application Logic Layer* and *Connection Layer*. The architecture of NG IDrOS Central is represented in Figure 4.3.

Layering the architecture allows to separate the components from a functional point of view. Layers and components are integrated with each other using decoupling interfaces. Layering and decoupling allow changing and extending some parts of the software platform without affecting all the others, making it very simple to extend the software platform with new functionality, and building integrations with new systems.

The last section of this chapter is dedicated to describe some deployment settings and some interactions between NG IDrOS Drone and NG IDrOS Central.

4.2 NG IDrOS Drone

This part of the software platform is the one directly deployed onto the drone and it offers capabilities to control the hardware, to exploit high-performance network capabilities, to manage user modules, and to run missions. It also provides a remote interface, accessible through the internet, to monitor and control the drone. This remote interface is exploitable to integrate the plat-

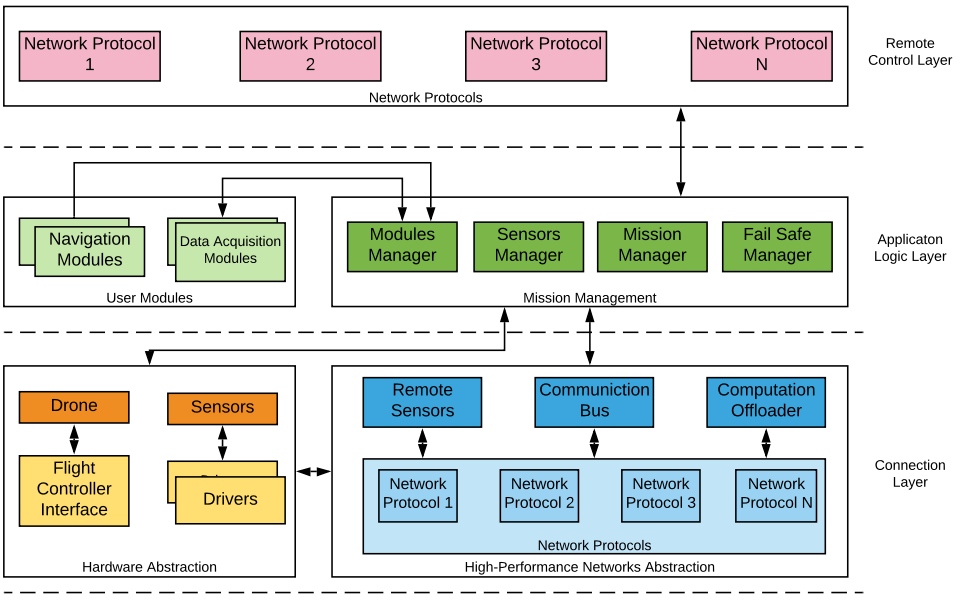


Figure 4.1: NG IDrOS Drone architecture.

form with external systems. NG IDrOS Drone can also be deployed on devices that are not UAVs. For instance, it can be deployed on a dedicated board to make a sensor exploitable by remote hosts. In this case, only the needed components and modules of the architecture would be instantiated. The architecture of this software component is layered into three separate levels: the *Connection Layer* holds the components needed to interface the system with the hardware — sensors and flight controller — and the functionality needed to exploit high-performance networks. The *Application Logic Layer* exposes a set of APIs to be leveraged by application developers to manage modules, missions and sensors. APIs from the Application Logic Layer are also exposed by the network protocols implemented in the *Remote Control Layer*, that provides functionality to remotely access the system through the internet. This network interface is also exploitable by external systems to implement machine-to-machine integrations.

4.2.1 Connection Layer

This layer holds the connectivity towards the flight controller, towards the locally connected sensors, and it also holds the functionality needed for the exploitation of high-performance networks. These features are grouped into two separate components: *Hardware Abstraction* and *High-Performance Networks Abstraction*.
The first one holds two modules: *Drone* provides the functionality to interact with the flight controller; *Sensors* provides an interface towards the locally installed sensors.

Specifically, the *Drone* module satisfies the requirements of the programming model related to the flight control that are defined in Section 3.2.4, and it provides the following functionalities:

- To abstract the communication towards the flight controller;
- To support multiple types of flight controllers by hiding the specific communication protocol;
- To provide high-level APIs to the application developer to takeoff, land, reach a specific GPS coordinate, retrieve telemetry information, change flight mode.

The high-level APIs provided by this module are intended to simplify the number of commands that usually must be sent to the flight controller to pilot the drone. They must also be agnostic with respect to the kind of drone and flight controller. This allows to write missions that can be reused in different deployment configurations without any change needed.

The *Sensors* module satisfies the requirements of the programming model related to local sensors that are defined in Section 3.2.2, and it provides the following functionalities:

- To list sensors that are installed onboard;
- To provide high-level APIs to access the descriptor of a sensor and to request for observations.

Each sensor that is installed on board needs a *Driver* to be loaded into IDrOS in order to allow the platform to support different type of sensors. All sensor drivers must implement the same methods that are specified by a common interface. The high-level APIs exposed by the Sensors module hide every implementation detail of the sensor, such as specific communication protocols. Also the fact that the sensor is of push or pull type is hidden to the application developer.
This module was already present in the original IDrOS architecture, in this new version we extend it to support descriptors. Moreover, while in the original IDrOS the *Sensors* module was directly exploited by application developers in writing drones missions, in this new architecture the access to sensors is further mediated by the *Sensors Manager* from the *Application Logic Layer*. This further mediation is needed to:

- Provide to the application logic layer a unified access point to both local sensors and remote sensors.
- Implement the providers comparison mechanisms and the best sensor selection mechanism.

The second component included in this layer holds three modules: *Remote Sensors* provides the functionality to interact with remote sensors and to publish local sensors as exploitable by remote hosts; *Communication Bus* holds all the functionality needed to create, join, send and receive messages through the communication bus; *Computation Offloader* provides the functionality needed to offload and inload part of the application logic at runtime. All these components, grouped in the *High-Performance Networks Abstraction*, access network transport using a specific interface.

The *Remote Sensors* module satisfies the requirements of the programming model related to remote sensors that are defined in Section 3.2.1, and it provides the following functionalities:

- To discover remote sensors by means of queries based on the properties contained in their descriptors;
- To retrieve properties from the descriptor of a remote sensor;

- To obtain measurements from a remote sensor;
- To make local sensors available to be exploited by remote hosts.

The exchange of information between the client hosting a sensor and the client accessing that sensor remotely can happen by means of a point-to-point connection or be mediated by an instance of NG IDrOS Central. As for the *Sensors* module in the *Hardware Abstraction* component, the APIs exposed by this module are directly exposed to the application developers, but they are further intermediated by the *Sensors Manager* module from the *Application Logic Layer*.

The *Communication Bus* module satisfies the requirements of the programming model defined in Section 3.2.3, and it is the module in charge of:

- Creating a new communication bus channel or join an existing one;
- Restricting the access to a specific bus channel by means of whitelisting or blacklisting mechanisms;
- Sending messages to other clients over the bus channel;
- Receiving messages sent over the bus channel by other clients.

The exchange of messages through a communication bus must use an instance of NG IDrOS Central as dispatcher. Messages are first sent to the central node, then they are forwarded towards all the relevant hosts. In this way, subscriptions for a specific bus channel are kept only on the central node. Plus, networks interconnections issues can be easily solved, as it is further detailed in Section 4.3.

The last component, *Computation Offloader*, satisfies the requirements of the programming model defined in Section 3.2.2, and it provides the following functionalities:

- To move user-defined functions towards a remote host;
- To forward the invocation of an offloaded function towards the offloading host;
- To return the result provided by the offloading host after a remote invocation;
- To inload previously offloaded functions on user request.

Both static functions — i.e. functions whose execution depends only on the input parameters and have not internal status — and dynamic functions — i.e. functions whose execution depends on the input parameters

and on an internal status — can be offloaded. This component is in charge of keeping track of the status of dynamic functions and to send it to the offloading host as well as restoring it when the function is inloaded again. This component must support all the offloading modes as they are described in Section 3.2.2.

All these modules access the high-performance network by means of a specific protocol. NG IDrOS must be able to support different protocols in order to guarantee the maximum degree of interoperability with other systems. Each different protocol may implement some peculiarities regarding non-functional aspects, such as security requirements or performance requirements.

4.2.2 Application Logic Layer

This architectural layer holds all the components and modules needed to allow the user to run application and missions on IDrOS. These components and modules implement all the requirements expressed for the programming model in Section 3.2, plus some additional requirements that are needed to implement the architecture in a real deployment setting. We decided to split a drone mission into two separate parts: a *Navigation Module* and one or more *Data Acquisition Modules*. In addition, we created a *Fail Safe Manager* module to provide to the application developers APIs to manage hazards and malfunctions. This layer is split into two separate components *Application Logic* and *User Modules*. This split between modules and components is reported in Figure 4.2.

The *User Modules* component holds the modules uploaded by the user into the system. These modules are then used by the *Mission Manager* to build up the mission to be executed by the drone. We decided to create a distinction between *Data Acquisition Modules* and *Navigation Modules*: they both are made of user-written code in which the application developer can exploit the APIs exposed by NG IDrOS, but the first ones are focused on the navigation part of the drone - i.e. piloting the drone - while the second ones are focused on the data acquisition part of the mission. Each mission

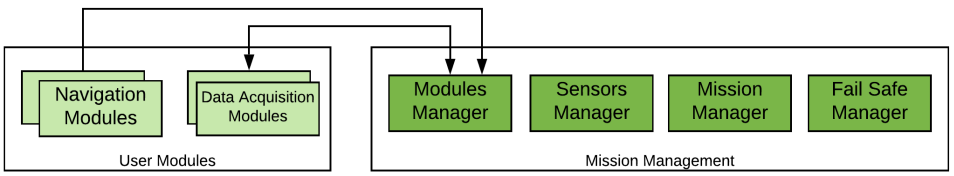


Figure 4.2: Application Logic Layer from the NG IDrOS Architecture.

is then build up of one and only one *Navigation Module* and zero to many *Data Acquisition Modules*, depending on the mission needs. The two types of modules are not isolated and can exchange information: a data acquisition module may request navigation status to the navigation module, and similarly the navigation module may request the advancement status of data processing to a data acquisition module.

The *Mission Management* component holds four different modules: *Modules Manager*, *Sensors Manager*, *Mission Manager*, and *Fail Safe Manager*. The *Modules Manager* provides to the software platform the following capabilities:

- To upload data navigation modules and data acquisition modules;
- To list currently stored navigation modules and data acquisition modules;
- To delete a navigation module or a data acquisition module;
- To upload a sensor driver;
- To list currently installed sensor drivers;
- To delete a sensor driver.

All these operations are allowed only when the drone is not executing a mission, since no hot-swap mechanism are in place for modules or drivers.

The *Mission Manager* must implement all the requirements of the programming model as defined in Section 3.2.4. It allows to:

- Select the modules included in the mission: one and only one Navigation Module, zero or more Data Acquisition Modules;
- Set a mission as cyclic or not cyclic;
- Start and stop a mission;
- During a mission, it enables the information exchange between the *Navigation Module* and the *Data Acquisition Modules*.

The functionalities provided by this module are mainly exploitable through a control network interface, as we are going to describe in Section 4.2.3.
The *Sensors Manager* module provides the access to both locally installed sensors and the remote accessed ones, providing a unified interface that hides any implementation detail between the two. This module implements all the logic about the sensor selection as described in Section 3.2.4.

The *Fail Safe Manager* is needed inside the architecture in order to handle

hazard situations or hardware failures that may compromise the integrity of the drone. A typical example is the presence of strong wind at high altitude that may cause the drone to precipitate. This module provides functionality to stop the execution of the mission and guide the drone safely to land. The Fail Safe Manager may be invoked by the remote interface and also be included in the code of a *Navigation Module*.

4.2.3 Remote Control Layer

This architectural layer allows NG IDrOS to be accessed remotely and to be integrated with other IoT applications. This layer exposes an interface towards all the components from the *Mission Management* module of the Application Logic Layer. This remote layer is accessed through the network by means of specific protocols, such as MQTT, COaP or HTTP. The architecture must support multiple protocols in order to ensure the integration with a number of different external systems. New protocols can be supported by implementing the specific interface that is used to integrate the *Remote Control Layer* and the *Application Logic Layer*. A running instance of NG IDrOS Drone must be able to expose the APIs provided by this layer using the implemented protocols.

The high decoupling between this layer and the rest of the architecture allows to add new supported protocol without any change needed to the rest of the software platform.

4.3 NG IDrOS Central

This part of the software platform works as central node among drones in order to enable some functionalities for high-performance networks exploitation.
This component can be deployed on a dedicated virtual host (such as on a Mobile Edge Computing virtual machine provided by the 5G infrastructure), on a drone (together with an instance of NG IDrOS Drone), in a cloud environment (such as Amazon Web Services), or on any other kind of physical host. It can be deployed as part of a specific drones mission or as general purpose instance, serving multiple missions during its lifetime.

The role of this component is to behave as a coordinator and as shared repository between clients. Specifically, this software component works as:

- Repository of remotely accessible sensors;
- Resolver for remote sensors queries;

- Dispatcher of measurement requests and measurement values for remote sensors;
- Execution host for offloaded functions;
- Repository of communication bus channels;
- Dispatcher of messages sent over communication bus.

The architecture of this software component is split into two layers: the *Connection Layer* and the *Application Logic Layer*.

The *Connection Layer* holds the protocols to exploit the high-performance networks, similarly to the same layer present in the architecture of NG IDrOS Drone. In order for two clients to interact, regardless the fact that they are instances of NG IDrOS Drone or instances of NG IDrOS Central, they must access the network through the same protocol, or through protocols that are compatible.
The other requirement for two clients to interact is that they must be able to reach each other inside the local-area, metropolitan-area, wide-area network they are connected to. The role of NG IDrOS Central is also to overcome possible network limitations that clients can experience, such as NAT limitations, blocked ports or missing interconnection between networks. For example, it can work as public node exposed at a globally reachable address, or as bridge node between two local isolated networks. In Section 4.4 we describe some examples regarding the usage of NG IDrOS Central to overcome network limitations.

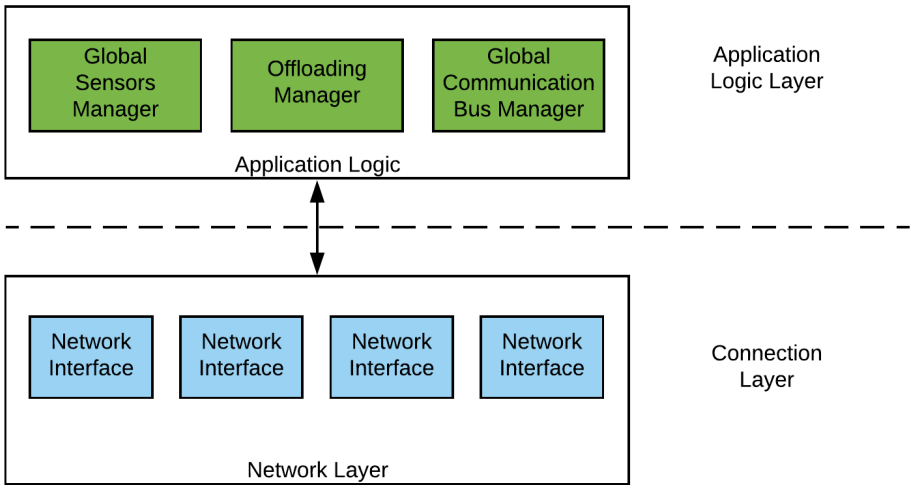


Figure 4.3: The NG IDrOS Central architecture.

The *Application Logic Layer* is composed by three modules, each of which is related to a macro-functionality that exploits high-performance networks: *Global Sensors Manager*, *Offoading Manager* and *Global Communication Bus Manager*.

The functionalities provided by the *Global Sensors Manager* module are:

- To keep a repository of sensors that are exposed by NG IDrOS Drone clients;
- To resolve queries issued over the exposed sensors;
- To retrieve dynamic properties of a sensor descriptor from its local host;
- To forward measurements requests and measurements values between a remote sensor consumer and a remote sensor provider.

Clients that want to make a local sensor available for remote exploitation must send the sensor descriptor to a central node. Clients that want to connect to a remote sensor must first issue a query to the central node in order to discover it. Once the query results contain a sensor that satisfies the needs of the issuing client, it can choose to connect directly to the sensor host or to use the central node as intermediate bridge.
Two connection modes to remote sensors are supported:

- *Direct* a peer-to-peer connection is open between the clients. The network configurations must allow such scenario;
- *Centralized* the NG IDrOS Central instance works as broker of read requests and measurements values among clients.

The functionalities provided by the Offloading Manager are:

- To receive the code of offloaded functions;
- To handle requests of remote invocations towards offloaded functions;
- To run offloaded functions on request;
- To send back the execution result and possible execution errors.

Several instances of NG IDrOS Central may be available for a client at the same time. Each client must be able to select towards which host the function will be offloaded. The *Offloading Manager* must support both static and dynamic functions, which means it must be able to restore the internal status of a dynamic function when it is received, and it must be able to dump and send back such status when the function is inloaded back.

The *Global Communication Bus Manager* offers the following features:

- To keep a repository of created communication bus channels, together with their members;
- To add and remove clients from a bus channel;
- To implement exclusion logic as from the requirements of the programming model;
- To receive and forward messages inside the bus.

As it is already stated in Section 3.2.3, no special mechanisms are implemented to guarantee the in-order delivery of messages or the errors correction. For this reason, no specific architectural components are present at the moment to handle this kind of needs.

An instance of NG IDrOS Central must be able to choose which of the modules from the application logic layer to instantiate, since not all features may be required at the same time. For example, an instance of NG IDrOS Central can be deployed on a drone only to coordinate a communication bus. In this scenario, only the *Global Communication Bus Manager* will be instantiated in order to save physical resources on the host.

4.4 Deployment Settings

The content of this section has the purpose of showing and describing some deployment settings that are supported by the architecture of NG IDrOS. We are going to highlight the role of NG IDrOS Central in different network conditions, as well as to highlight how the high modularity and functional decoupling that drove the design of the architecture make NG IDrOS flexible to support several deployment modes.

Each deployment setting is associated to a high-level sequence diagram in order to describe how the components inside the architecture interact, and how instances of NG IDrOS Core and instances of NG IDrOS Central interact.

4.4.1 Mobile Edge Computing

4.4.1.1 Two separate local networks

The deployment setting shown in Figure 4.4 is the example of a possible deployment that exploits the hosting capabilities provided by the Mobile Edge Computing available in the 5G infrastructure. Three different networks are present: *Network A* is the local network generated by the 5G Antenna A; *Network B* is the local network generated by the 5G Antenna B; *Mobile Edge Computing Network* is the network where mobile edge virtual machines

are hosted.
The following interconnections exist between the three networks:

- Network A and Mobile Edge Computing Network are interconnected;
- Network B and Mobile Edge Computing Network are interconnected;
- No interconnection exists between Network A and Network B.

A drone is connected to Network A, running an instance of NG IDrOS Drone. A pull sensor is connected to Network B, running an instance of NG IDrOS Drone that remotely exposes the sensor. No architectural components for flight control or mission management are instantiated as part of NG IDrOS Drone Instance #2, but only the ones related to remote sensors sharing. Since Network A and Network B are not directly interconnected, no client-to-client connection can happen between the drone and the sensor.

The sequence diagram represented in Figure 4.5 shows an interaction between the two clients in whom the drone can remotely access the sensor. Since the two networks are not interconnected, the only connection mode is centralized, leveraging the NG IDrOS Central instance as bridge and as dispatcher.

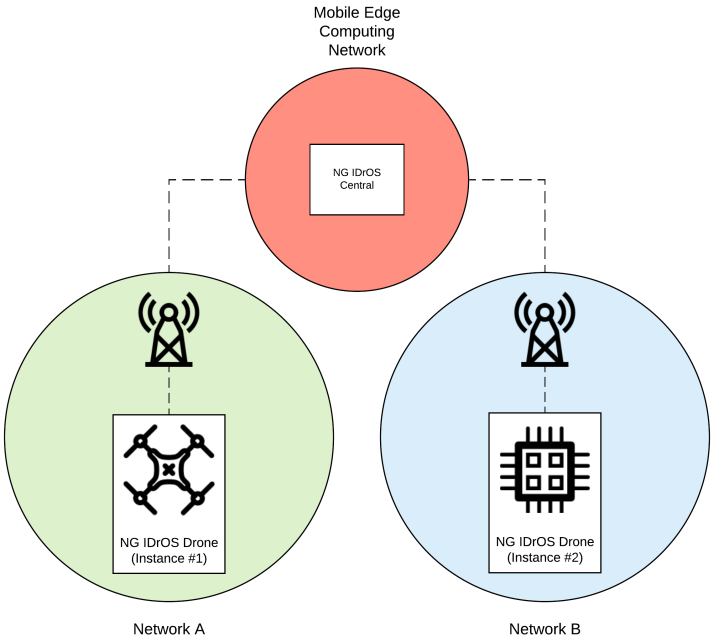


Figure 4.4: Example of NG IDrOS deployment on Mobile Edge Computing with clients in isolated networks.

The diagram also shows the interaction between the components and the modules inside the architectures.

The diagram starts with the sensor being published in the global repository hosted into the NG IDrOS Central instance. The second part consists of the query request, resolution and results processing. The last part of the diagram starts from the Sensors Manager of the Application Logic Layer of the instance #1 of NG IDrOS Drone requesting a measurement to the Remote Sensors module: the request is sent to the central node, and is

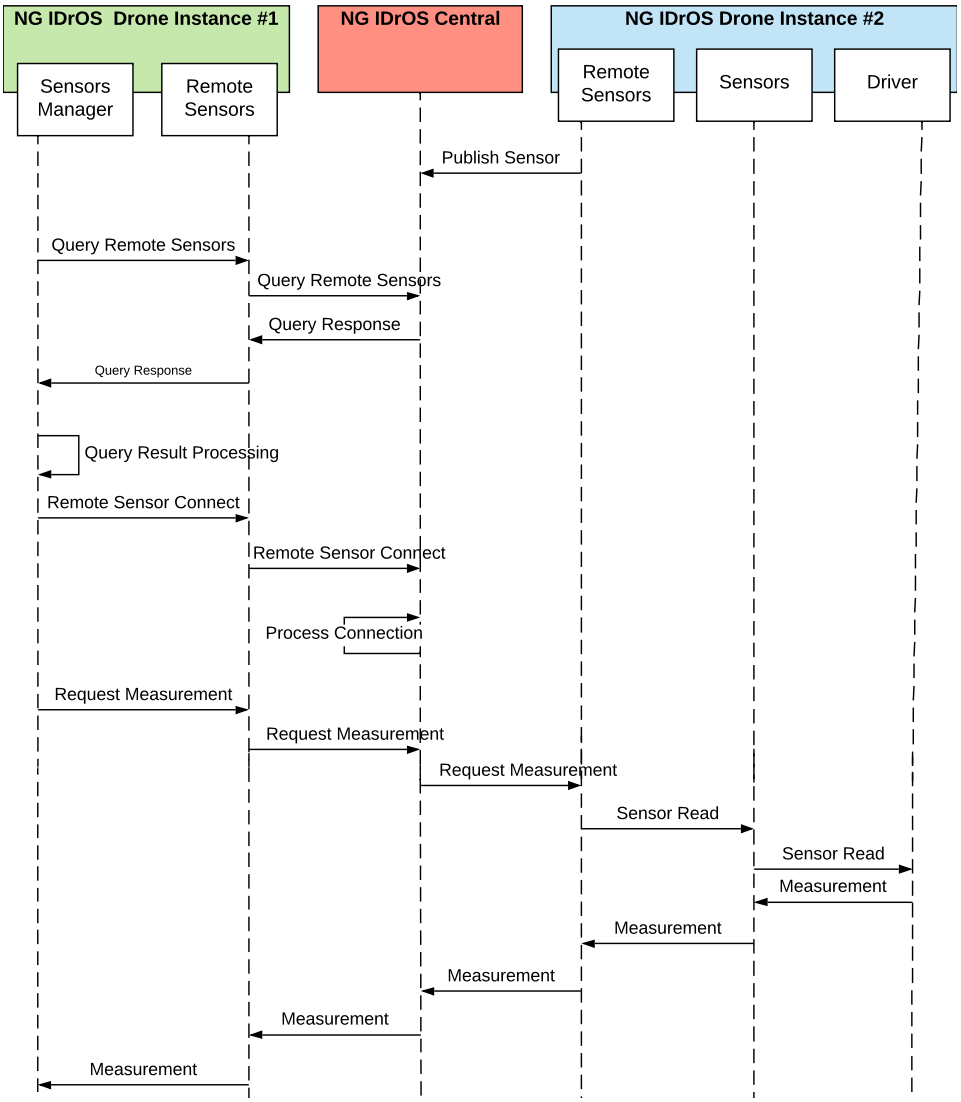


Figure 4.5: Sequence diagram of remote sensor exploitation inside the deployment setting of Figure 4.4.

then forwarded to the Remote Sensors module from the Connection Layer of NG IDrOS Drone instance #2. Internally, the request goes to the Sensors module of the Hardware Abstraction component and finally arrives at the driver. Once the measurement is ready, the return value follows back the entire chain and is returned to the Sensors Manager of the first NG IDrOS Drone instance.

In this specific deployment setup, the instance of NG IDrOS Central is working as bridge between the two networks. With such setup it is possible not only to connect hosts from two isolated networks, but also to overcome issues coming from NAT restrictions or client isolation inside the network.

This connection mode adds to the end-to-end latency of the process the overhead coming from the intermediation of the central node. However, only one network connection must be issued to the central node: this saves physical resources at the nodes. Moreover, the central node can avoid issuing a new read request towards the remote sensor if it is already waiting for a return value for a previous request. By implementing this requests caching mechanism, the number of requests is further reduced.

4.4.1.2 Interconnected Local Networks

The deployment setting shown in Figure 4.6 is the example of a possible deployment that exploits the hosting capabilities provided by the Mobile Edge Computing inside the 5G network. Three different networks are present: *Network A* is the local network generated by the 5G Antenna A; *Network B* is the local network generated by the 5G Antenna B; *Mobile Edge Computing Network* is the network where mobile edge virtual machines are hosted. The following interconnections exist between the three networks:

- Network A and Mobile Edge Computing Network are interconnected;
- Network B and Mobile Edge Computing Network are interconnected;
- Network A and Network B are interconnected.

The sequence diagram represented in Figure 4.7 shows a remote sensor exploitation based on a client-to-client connection. Differently from the sequence diagram of Figure 4.5, once the query result is processed, the instance of NG IDrOS Central is no more involved in the measurements requests and exchanges. This connection mode reduces the overhead introduced by the bridging of the central node, however it significantly increases the amount of connection and requests that each client must handle. A high number of requests may lead to bottlenecks for low computational power devices.

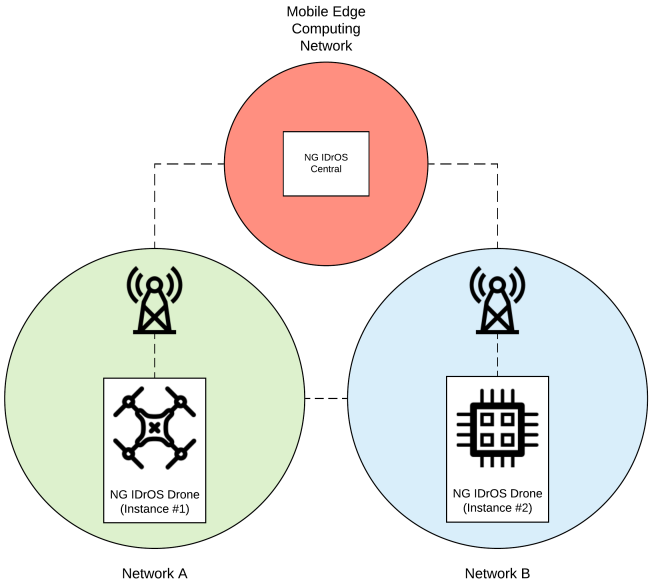


Figure 4.6: Example of IDrOS deployment on Mobile Edge Computing with two clients in interconnected networks.

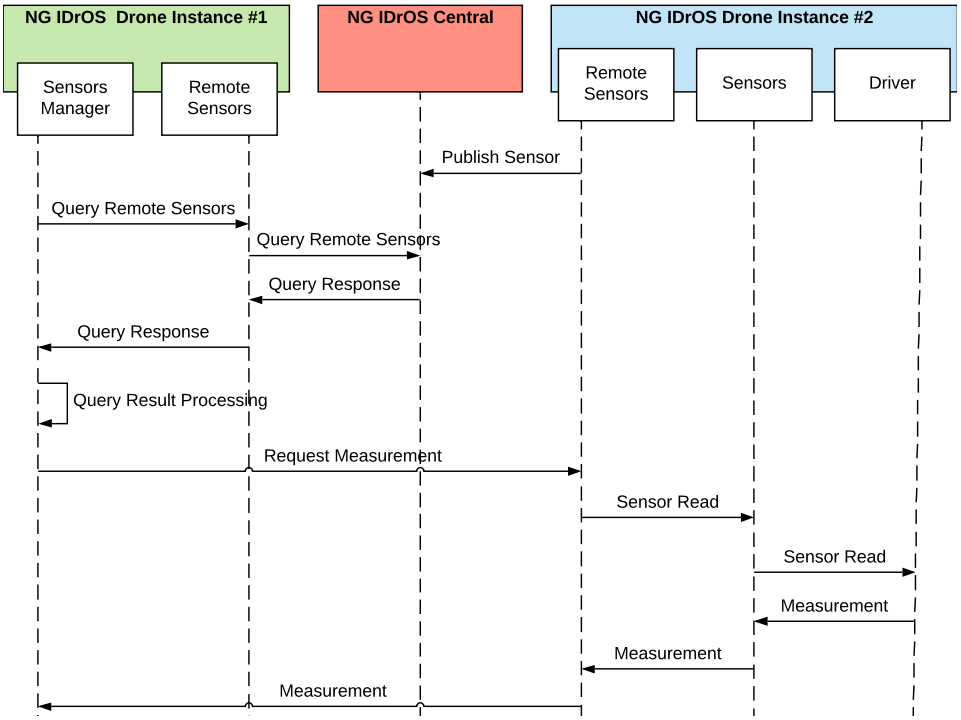


Figure 4.7: Sequence diagram of remote sensor exploitation through a client-to-client connection inside the deployment setting of Figure 4.6.

4.4.2 Local Host

The deployment setting represented in Figure 4.8 shows three drones deployed in the same local network provided by a 5G antenna, and an instance of NG IDrOS central deployed in the Mobile Edge Computing environment. Each drone runs an instance of NG IDrOS Drone, and only one of the three also runs an instance of NG IDrOS Central.
In this deployment setting, the NG IDrOS Central instance #1, running on the Mobile Edge Computing environment, is leveraged by NG IDrOS Drone instance #1 for computation offloading. The instance of NG IDrOS Central #2 is used as dispatcher for a communication bus involving both the three drones.
Since the perimeter of the communication bus is limited to the three drones in the same network, having the coordinator hosted in the same network guarantees a reduced latency. The computational power available at the host must be sufficient to support both an instance of NG IDrOS Central and an instance of NG IDrOS Drone at the same time, as well as to support a higher number of connections at the same time.

The sequence diagram represented in Figure 4.9 shows two distinct interactions among the clients present in the aforementioned deployment setting. The first one involves NG IDrOS Drone #1 and the instance of NG IDrOS Central deployed in the Mobile Edge Computing environment. It starts with the Application Logic Layer requesting to offload a function to the Computation Offloader of the Connection Layer. The request, together with the code and the actual internal status of the function are forwarded to the Offloading Manager of the NG IDrOS Central instance. Then, a remote invocation flow is shown.

The second interaction shows the creation of a communication bus, and a message exchange over it. Also in this case the interaction starts with the request being issued by the Application Logic Layer of NG IDrOS Drone instance #1, but in this case it is sent by the Communication Bus to the Global Communication Bus Manager of the NG IDrOS central instance #2.

The diagram does not show the possible interaction of drone #2 joining the communication bus as well. In this case, since the instance of NG IDrOS Central that is coordinating the bus is hosted by the same client, the communication is going to leverage the loopback interface of the host.

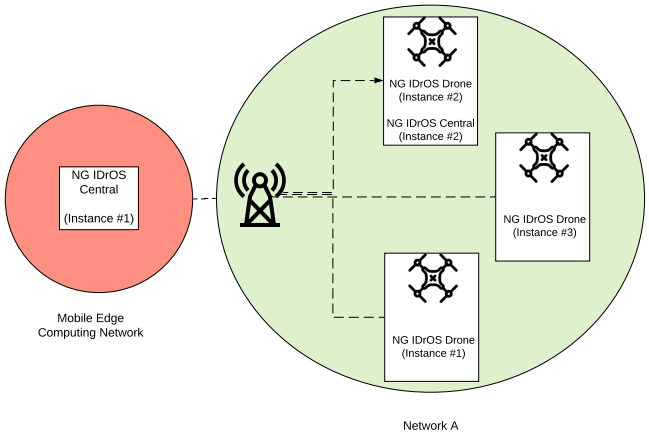


Figure 4.8: Example of IDrOS deployment with IDrOS NG Central deployed both on Mobile Edge Computing and on a local host.

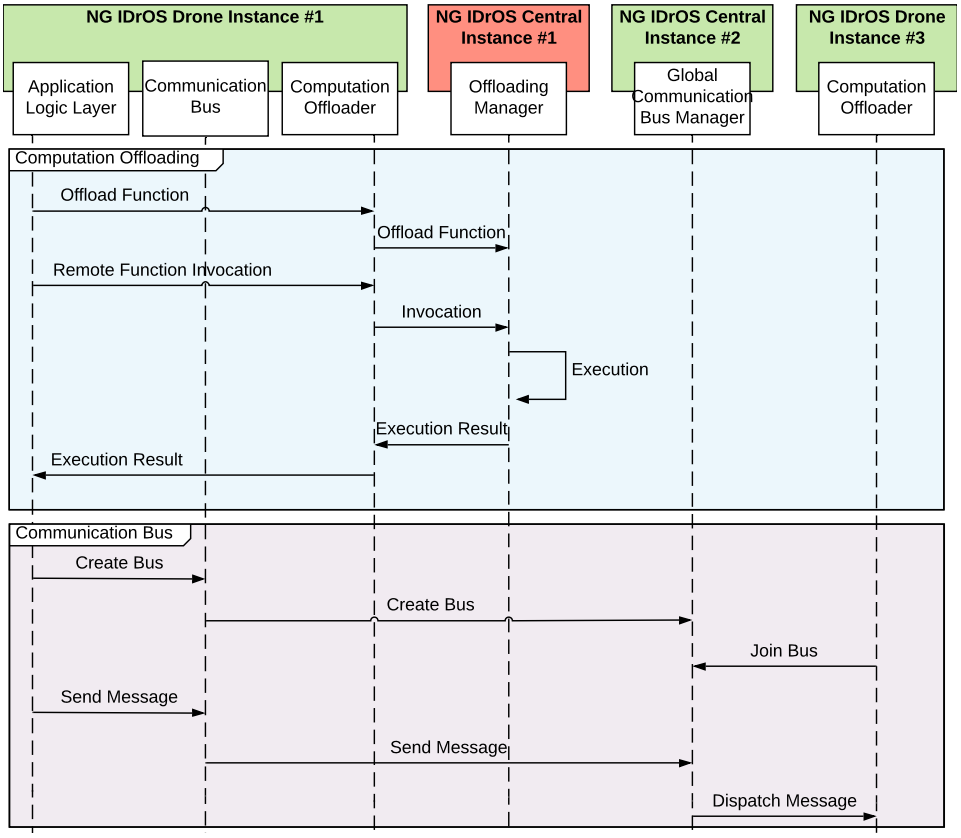


Figure 4.9: Sequence diagram showing a Computation Offloading interaction and a Communication Bus interaction inside the deployment setting of Figure 4.8.

4.4.3 Bluetooth Network

The descripiton of this last deployment setting, reported in Figure 4.10, has the purpose of empathizing the flexibility of NG IDrOS to be adapted to very different deployment scenarios.

Three drones are connected through a high-performance peer-to-peer Bluetooth network. One of these three hosts runs an instance of NG IDrOS Central together with an instance of NG IDrOS Drone. These two instances are able to interact using the loopback interface of the host, based on the IP protocol. The other two hosts run an instance of NG IDrOS Drone. All the three hosts communicate the one with the other using the Bluetooth protocol.
This deployment setting is particularly efficient to run drones mission where an internet connection is not present. Of course, missions must be adapted to take into consideration the limitations coming from the Bluetooth connection, such as the maximum distance between the drones or the increased transmission errors. However, for all the other aspects, the application logic does not need any further change.

The architecture we designed is able to support this deployment by just providing to all the instances a Network Interface based on the Bluetooth Protocol. No changes are needed to the other core parts of the platform.

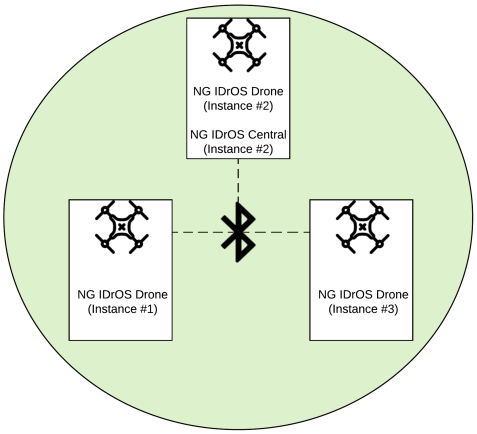


Figure 4.10: Example of NG IDrOS deployment involving a Bluetooth high-performance network.

Chapter 5

Implementation

The purpose of this chapter is to describe the details and choices that have driven the implementation of NG IDrOS. For each component and module of the architecture we are going to highlight the main implementation characteristics, also with respect to the way the component or the module is integrated with the rest of the system. The implementation we developed specifically targets UAVs as instances of high-mobility robots, and the 5G cellular network as instance of a high-performance network. For this reason, some parts of the implementation target UAVs and the 5G as well, and must be reviewed to port NG IDrOS to other robots or networks.

The first decision we took at the beginning of the development was to implement NG IDrOS using Python 3.7. This choice was driven by the high availability of libraries and by the high compatibility with different hardware platforms. These two characteristics guarantee a variety of different deployment settings, as well as a high degree of interoperability with other systems.
We then decided to implement both NG IDrOS Drone and NG IDrOS Central in one software package, and to use a configuration file to specify which one of the two must be instantiated. Thanks to this approach, just one software package must be shipped to hosts, and it makes easier to switch from one instance to the other, as well as to run both in parallel on the same host. From a development point of view, this allowed reusing most of the software components and will make it easier to maintain them. A drawback from this approach is the slightly increased overall size of the package.

5.1 Connection Layer

In this chapter we are going to report the implementation details for the Connection Layer, which is the lowest level of the NG IDrOS Drone architecture, that is reported in Figure 5.1. The role of this layer is to handle

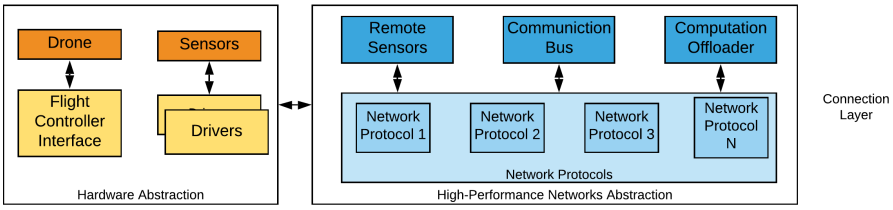


Figure 5.1: Control Layer from the NG IDrOS Architecture.

the interfaces with the flight controller, with the physical sensors and the exploitation of the network, as it is extensively described in Section 4.2. In this section we are also going to report the implementation of NG IDrOS Central, described in Section 4.3, because the two are closely related from the implementation point of view. Since we sdecided to implement a single software package including both the software components, this layer is actually implemented once and shared among the two.

5.1.1 Hardware Abstraction

5.1.1.1 Flight Controller

The *Drone* component has the purpose of providing a simplified set of APIs to the Application Logic Layer that can be easily exploited by application developers to pilot the drone. In order for this module to provide these APIs, it should be able to open, manage, and then close a connection towards the flight controller. A flight controller is made up of a software platform and of the hardware device into which the software is deployed. Several flight controllers exist: they mainly differentiate for the number of flight modes they support, the variety of drones they can control (e.g. quadcopters, hexacopters) and the variety of hardware platforms they can run onto. For the development of NG IDrOS, we decided to target ArduCopter [32], which is one of the most used and community-supported flight controllers. It can be deployed into a large variety of hardware platforms (e.g. Pixhawk [33], DJI Flight Controllers [34], and others), and it supports the MAVLink protocol [35] for communication.
The MAVLink protocol is based on the exchange of header-only messages and was originally designed for the communication between a ground station and the UAV. Being header-only means that the structure and the content itself of the message can be inferred from the structure of the header, and this is the main reason that makes MAVLink such a lightweight communication protocol. MAVLink supports a variety of different sensors, such as gyroscopes, accelerometers and magnetometers. It also allows to control some payloads, such as gimbals. It is a widely adopted protocol, and in-

tegrating it into NG IDrOS makes the platform natively compatible with several existing drones platforms. MAVLink can be exploited using a wired connection towards the flight controller or also using a TCP/IP network connection. During the implementation phase we leveraged a lot the TCP/IP connection mode because it can be used to connect to drone simulators, such as the DroneKit SITL simulator [36]. The same can be done during application development and testing. For a real deployment setting the direct wired connection should be preferred since it is more stable and faster.

We decided to build a MAVLink interface based on the *pymavlink* library [37]. Several other libraries are available for Python (e.g. MAVProxy [38] and DroneKit [27]) that provide a higher level of abstraction. However, we decided to use *pymavlink* since it is the reference implementation for the protocol. We used the interface built by Cantoni in his thesis [26] as starting point, and we adapted it to the new NG IDrOS architecture. The final set of APIs that the Drone component exposes to the application logic layer are reported in Listing 5.1.

```
1  def arm() -> void:
2  Arms the drone before the flight
3
4  def takeoff(altitude) -> void:
5  Takes off the drone and reaches the specified altitude
6
7  def land() -> void:
8  Makes the drone to land
9
10 def return_home() -> void:
11 Makes the drone to land at the place it has taken off
12
13 def set_flightmode(flightmode, vars) -> void:
14 Changes the current flight mode. The vars parameter is optional
       and applies to a subset of flight modes only.
15
16 def get_location() -> (lat, long, alt):
17 Returns latitude, longitude and altitude of the drone from the
       telemetry provided by the flight controller
18
19 def get_speed() -> (vx, vy, vz):
20 Returns the drone speed with respect to the three space axis
21
22 def heading() -> heading:
23 Returns the orientation of the drone
```

Listing 5.1: APIs exposed by the Drone component

The flight modes supported by NG IDrOS are:

- *Stabilize* This mode allows the pilot to manually control the roll and pitch of the UAV using a radio controller, but it uses the flight controller to self-level.

- *AltHold* This mode allows the pilot to manually control the roll, pitch, and yaw of the UAV using a radio controller, but it uses the flight controller to maintain a consistent altitude.

- *Guided* This mode allows the UAV to autonomously reach a specified position and altitude. In order to use this mode, a GPS sensor must be installed onboard.

- *Loiter & PosHold* These two modes use the flight controller to maintain the location, heading and altitude of the UAV.

The APIs we listed and the flight modes we decided to support consist of a subset of all the features supported by the flight controller and a subset of all the available flight modes. Thanks to the layered and decoupled architecture of NG IDrOS, both the APIs and the supported flight modes can be easily extended without impacting any other part of the software platform. It is just sufficient to extend the *Flight Controller Interface* and the *Drone* component.

5.1.1.2 Sensor

The purpose of the *Sensors* module is to provide an access interface towards the sensors installed onboard. A driver must be provided for each sensor that is installed onboard. The driver must include both the descriptor of the sensor and the implementation to request measurements.
Each sensor driver must implement a specific abstract class, named *Driver*, that defines the mandatory methods to be implemented, as well as the mandatory properties of the descriptor. This interface is reported in Listing 5.2.

The attributes belonging to the sensor descriptor are marked with the specific decorator *@descriptor*. Additional *custom properties* can be added to the descriptor just by using this decorator. The Driver also specifies if the sensor is remotely exploitable by other clients, by means of the *makeremote* decorator applied to the entire class.
Listing 5.3 provides an example of a driver for a *pull* temperature sensor, that is remotely exposed to other clients and that communicates with the hardware via the GPIO interface.

We decided to implement the sensor descriptor by means of methods inside the sensor driver because of the presence of dynamic properties, whose values are not static and may change at each invocation. A different approach, such as a structured XML file, would have increased the complexity needed to support such dynamic properties.

```
class Driver(AbstractClass)
    /* Sensor Descriptor */
    @descriptor @abstractmethod
    def identifier:
    Returns the identifier of the sensor

    @descriptor @abstractmethod
    def interval:
    Returns the interval of the sensor

    @descriptor @abstractmethod
    def type:
    Return the type of the sensor, PUSH or PULL

    @descriptor @abstractmethod
    def quantity:
    Returns the quantity mesured by the sensor

    @descriptor @abstractmethod
    def quantity_uom:
    Returns the unit of measurement of the quantity read by the
        sensor

    @descriptor @abstractmethod
    def accuracy:
    Return the accuracy of the sensor. It can be a static
        value, or a dynamic value

    @descriptor @abstractmethod
    def accuracy_uom:
    Returns the unit of measurement of the accuracy

    @descriptor
    def accuracy_type:
    Returns STATIC if the accuracy method is a static method,
        DYNAMIC otherwhise

    @classmethod
    def __description__(cls) -> dict:
    Returns an object containing all the properties of the
        descriptor

    /* Sensor Measurements */
    @abstractmethod
    def __read__(self):
    Requests a measurement to the sensor and returns its value
```

Listing 5.2: The Driver interface.

```
@makeremote
class ExampleTemperatureSensor(Driver):
    @descriptor
    def identifier:
    return "example_sensor"

    @descriptor
    def interval:
    return 0.5 //Corresponds to 2Hz

    @descriptor
    def type:
    return PULL

    @descriptor
    def quantity:
    return "temperature"

    @descriptor
    def quantity_uom:
    return "centigrade"

    @descriptor
    def accuracy_uom:
    return "centigrade"

    @descriptor
    def accuracy_type:
    return STATIC

    @descriptor
    def accuracy:
    return 0.1

    @descriptor
    def sensor_brand: // Custom static descriptor property
    return "ST"

    def __read()__: // Read the GPIO
    GPIO.input(4);
```

Listing 5.3: Example of sensor driver of a pull remotely exposed temperature sensor, with a custom descriptor property.

5.1.2 High Performance Networks Abstraction

The implementation of this component required most of the effort during the implementation phase, since it is related to the core contribution of this thesis work. In this section we are going to describe the *Remote Sensors*, *Communication Bus* and *Computation Offloader* modules of the architecture. Then, we are going to present the network protocol we implemented.

5.1.2.1 Remote Sensors, Communication Bus and Computation Offloader

All these three modules provide specific functionality related to a specific aspect of the high-performance networks exploitation. Our implementation approach was to define all the three as abstract classes that must be implemented by the network protocol.

The *Remote Sensors* module contains the methods that must be implemented for remote sensors discovery, connection, disconnection and measurements read. It also contains the class *Remote Sensor Driver* that proxies the access towards remote sensors. Listing 5.4 reports the methods of the Remote Sensor class, while Listing 5.5 reports the methods of the Remote Sensor Driver.

```
1  class RemoteSensors(AbstractClass):
2      @abstractmethod
3      def sensors_discovery(self, query, callback=None) -> List:
4          Invoked to issue a remote sensor query. It returns a
               possible empty list.
5
6      @abstractmethod
7      def remote_sensor_connect(self, remote_sensor) -> bool:
8          Invoked to connect to a remote sensor.
9
10     @abstractmethod
11     def remote_sensor_disconnect(self, remote_sensor) -> bool:
12         Invoked to interrupt the connection with a remote
               sensor, whenever it is no longer needed.
13
14     @abstractmethod
15     def remote_sensor_read(self, remote_sensor):
16         Invoked to acquire a measurement from the sensor.
```

Listing 5.4: The Remote Sensors module.

```
1  class RemoteSensorDriver(Sensor):
2      def __init__(self, descriptor, remote_interface):
3         A remote sensor must be initiated by means of its
              descriptor and the remote interface object
4
5      def __read__(self):
6          This method invokes the remote_sensor_read  method from
               the remote interface.
7
8      @classmethod
9      def __description__(cls) -> dict:
10         Returns an object containing all the properties of the
               descriptor
```

Listing 5.5: The Remote Sensor Driver.

The descriptor of a remote sensor requires an additional mandatory field named *remote_host* which keeps a reference to the physical host to which the sensor is physically connected. This is needed by the remote interface in order to know towards which host the requests must be issued.
In order for a Remote Sensor Driver to look exactly as a local sensor driver, when the class is instantiated, a set of methods is dynamically created from the names of the properties in its descriptor. In this way, not only the properties that we defined as mandatory can be accessed through a method that corresponds to the name of the property, but also the custom defined ones. The methods exposed by these two classes are not directly accessed by the application developers, but are exposed only toward the Global Sensors Manager of the application layer.
The *Communication Bus* module holds the abstract methods that must be implemented by the network protocol in order to provide the functionality to create, join and leave a communication bus channel, and to send and receive messages using it. Since these methods are directly accessed by the application logic layer, they hide the unnecessary implementation detail. These methods are reported in Listing 5.6.

```
1  class CommunicationBus(Sensor):
2      @abstractmethod
3      def create_bus_channel(channel_name, nodes_witelist,
           nodes_blacklist):
4          This method allows to create a new channel over the
               bus, using a blacklist or whitelist filtering
5
6      @abstractmethod
7      def join_bus_channel(channel_name):
8          This method is used to join a specific channel
9
10     @abstractmethod
11     def leave_bus_channel(channel_name):
12         This method is used to leave a specific channel
13
14     @abstractmethod
15     def get_bus_members(channel_name):
16         this method returns the list of members in a specific
               channel
17
18     @abstractmethod
19     def send_message(channel_name, message):
20         This method is used to send a message over a specific
               channel
21
22     @abstractmethod
23     def on_incoming_message(self, channel_name, message):
24         This callback is invoked when a new message is received
               in a specific bus channel
```

Listing 5.6: The Communication Bus module

The last module hold by this component is the *Computation Offloader*. It exposes methods to offload, inload and then invoke an offloaded function. These methods are directly exploitable by the application developer, and for this reason all the implementation details are hidden. For example, all the methods are valid for both static and dynamic functions, even though the mechanisms behind are radically different. This module is an abstract class whose implementation must be provided by the network protocol. Listing 5.7 reports the methods of this abstract class.

```
1  class ComputationOffloader(AbstractClass):
2      @staticmethod
3      def init_method_offloading(method)
4          Offloads the method given as input
5
6      @staticmethod
7      def dump_global_variables(method):
8          Retrieves the values of the state variables of the
               method given as input from the remote host
9
10     @staticmethod
11     def revoke_method_offloading(method):
12         Inloads back the method given as input
13
14     @staticmethod
15     def offloaded_method_invocation(method, argv):
16         Remotely invokes the method given as input, with the
               given input parameters
```

Listing 5.7: The Computation Offloader module

5.1.3 Calvin

Before describing the network protocol we implemented, it is necessary to present *Calvin* [39], which is the application environment upon which we decided to implement the protocol.

Calvin is an open-source project by Ericsson Research started in 2015 and currently under development. The purpose of Calvin is to reduce the fragmentation of communication protocols, platforms and environments that are used in IoT applications. The way it tries to reach this goal is by providing to application developers a different way of building and managing distributed IoT applications based on a combination of the Actors paradigm [40] and of the Flow Based paradigm [41].
At the core of Calvin there is the division of an application into four different aspects:

- *Describe* the functional parts that make the application, and design them to be reusable.

- *Connect* the components that build up an application.
- *Deploy* the components according to their connections.
- *Manage* the mapping of components to hardware during the lifetime of the application.

The *describe* aspect is based on the Actor model. An actor is a reusable software component that runs into a specific runtime. Actors can communicate with each other only by means of *tokens* exchanged over input and output *ports*. Providing data over the input port of an actor is the only way to influence its status, and this allows one to move and migrate actors across runtimes, because it makes possible to wrap the actor together with its status at a given moment. While processing data incoming from an input port, actors may produce outbound tokens sent over their output ports. The description of an actor is made up of its action, its port and of the conditions that trigger an action.
Connections among actors are represented by means of graphs where the actors are the nodes and there are directed arcs going from output to input ports. Connections are the only way to exchange tokens between input and output ports.

When defining an actor, no specific information for its deployment is provided, because this stage is dynamically handled. Also, no instructions are specified on how tokens should be transferred among them. During the deployment phase, actors are mapped to a *Calvin runtime* that is reachable by the user executing the deployment operation. This runtime immediately migrates the actor towards the most suitable runtime (that can be this initial runtime itself). Runtimes form a mesh network wherein actors can move among nodes. This functionality is based on tagging each runtime with specific attributes, that can be then specified as requirements by actors. An example of such a tag can be the availability of a time provider, and actors requiring a time provider will automatically be migrated to tagged hosts. An actor will be deployed following load-balancing criteria if it does not require specific capabilities. The way actors communicate depends on where they are deployed and it is inherited from the way runtimes are connected the one with the other. Mixed communication protocols are possible in the mesh: for example, half of the nodes may communicate using Bluetooth while the other half using the IP protocol. It is sufficient that a single runtime is able to use both the protocols to create a bridge and form a single mesh.
Once all the actors are deployed on a runtime that satisfies all their requirements, the application can start running and it enters into the manage phase. During this phase, actors are monitored and can migrate and can scale based on the needs.

Calvin applications are written using a specific language named *Calvin Script*. Each application starts with the definition of the actors involved

into the application. Then, the connections among their ports are defined. Listing 5.8 reports the example of Calvin application for a vending machine. The language used to define actors depends on the language used to implement the Calvin runtime they are going to be deployed into. Currently, the reference runtime is written using Python, and so are mostly of the available actors. Listing 5.9 is the example of an actor generating a random number inside an interval specified by a lower and an upper bound.

```
/* Actors Definition */
moneyhandler : MoneyHandler(currency="euro")
dispenser : Dispenser()
keypad : KeyPad()
itemdb : Database()

/* Actors Connection */
keypad.number > itemdb.choice
itemdb.value > moneyhandler.request
itemdb.choice > dispenser.choice
moneyhandler.ok > dispenser.ok
```

Listing 5.8: Example of Calvin application for a vending machine

```
class RandomNumber(Actor):
    """
    Produce random number (floating point) in range [lower ...
        upper)
    Inputs:
      trigger : Any token
    Outputs:
      number : Random number in range [lower ... upper)
    """
    @manage(['lower', 'upper'])
    def init(self, lower, upper):
        self.lower = lower
        self.upper = upper
        self.setup()

    def setup(self):
        self.rng = calvinlib.use("math.random")

    def did_migrate(self):
        //This method is fired soon after the actor migration
        self.setup()

    @condition(action_input=['trigger'],action_output=['number'])
    def action(self, trigger):
        return self.rng.random_number(self.lower, self.upper)

    action_priority = (action, )
    requires = ['math.random']
```

Listing 5.9: Example of Calvin Actor producing a random number

The definition of an Actor always starts with the statement of its input and output ports, together with optional descriptions for the ports and for the actor itself. The *manage* decorator (line 9) defines the variables that form the state of the actor. The *condition* decorator (line 22) is used to define which input ports trigger the action, and which ports are used for the output. At line 26, it is specified the order in which actions must be evaluated, while at line 27 the capabilities that the actor requires in order to be deployed on a runtime.
The *Calvin Runtime* is specified by means of a set of lightweight APIs of which multiple implementations can be provided. This component of the Calvin stack is partially platform-dependent, since it is stacked directly on top of the operating system, as it shown in Figure 5.2. Calvin runtime systems can be extended by adding new capabilities using a plug-in mechanism: this flexibility makes Calvin suitable to be adapted for application-specific scenarios, and it is the key feature that allowed us to build a network protocol upon Calvin.

At the moment of writing, two different implementations of the Calvin runtime exist:

- *Calvin Base* Is a Python implementation of the runtime and it is the reference of the APIs.
- *Calvin Costrained* Is a C implementation of the runtime that targets devices with constraints on memory, computing, and power consumption.

Since actors and applications are stacked on the platform independent part, as it is shown in Figure 5.2, they can agnostically be deployed on Calvin Base and Calvin Constrained instances. This flexibility makes Calvin deployable in a very wide range of hardware devices, and it is a relevant feature we are going to exploit in our design.

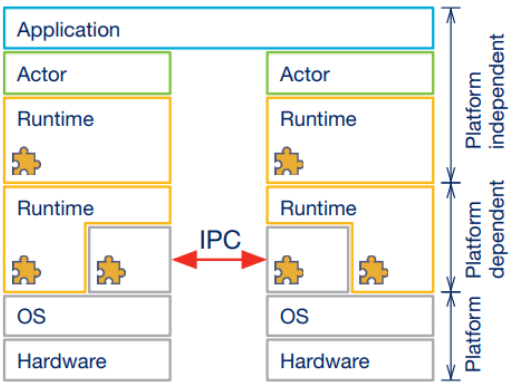


Figure 5.2: The Calvin software stack, taken from [39].

5.1.4 Calvin Network Protocol

We decided to build an NG IDrOS network protocol based on Calvin for five main reasons:

- To leverage the automatic deployment mechanism and the built-in migration features;
- To exploit the wide range of hardware devices into which a Calvin runtime (base or constrained) can be hosted;
- To benefit from the built-in creation of the mesh network;
- For the network-agnostic management of the connections, which is totally delegated to the operative system;
- For the built-in capability to create a mesh of nodes using different connection protocols, such as IP and Bluetooth.

These technical aspects allowed us to save a lot of effort during the development phase because we leveraged Calvin features and avoided to start an implementation from scratch for each of them. Together with the technical aspects, we decided to use Calvin because we agree with the goal of Calvin of reducing fragmentation of IoT, and we did not want to further contribute to this fragmentation by creating a new and isolated platform. Furthermore, using Calvin as network protocol allows NG IDrOS to be integrable in the growing ecosystem of Calvin applications.

However, using Calvin to develop our network protocol required a considerable effort because Calvin is distributed as a standalone runtime and no libraries or APIs exist to interact with it. The approach we followed was then to run an instance of Calvin runtime in parallel with an instance of NG IDrOS, and to exploit the command line interface to let NG IDrOS talk with Calvin. At NG IDrOS startup it is checked if a Calvin runtime is present on the host, and it is started if it is not already running. This creates a tight coupling between a Calvin runtime instance and an NG IDrOS instance, that can not work properly without a running Calvin runtime on the same host. A Calvin runtime instance is started using the *csruntime* command. The logs of the runtime are captured during its entire lifecycle to monitor for errors or malfunctioning.

The implementation of this Calvin based network protocol followed four distinct phases, that we are going to describe in this section:

1. Implementation of custom modules to extend Calvin base runtime capabilities;

2. Implementation of Calvin actors;

3. Implementation of generative programming capability for NG IDrOS to autonomously generate Calvin Script applications;

4. Implementation of an NG IDrOS class to implement all the abstract classes from the Hardware High-Performance Networks Abstraction component.

5.1.4.1 Implementation of Calvin Runtime Modules

The first step was to extend the standard capabilities of the Calvin base runtime by means of two ad-hoc module needed for NG IDrOS Central: *Central Sensors Manager Module* and *Central Bus Manager Module*. Calvin runtimes capabilities can be easily extended by means of a plug-in mechanism. Each new module, named *calvinsys module* must implement a specific abstract class that prescripts the methods to be implemented. It is then sufficient to place the file in the runtime modules folder and it will be automatically loaded at startup time and made usable by actors.

The *Central Sensors Manager* module implements the functionality described in Section 4.3 for the Global Sensors Manager module from the architecture: it works as repository of remote sensors, as resolver of queries and it forwards measurements requests and values among hosts. It maintains an internal state tracking the provider of each sensor, which host is connected to which remote sensor, and it keeps an internal buffer of sensors measurements. It is capable of caching read requests for pull sensors with the purpose of reducing the number of requests issued towards the sensor and of reducing the latency. When requests arrive while this module is already waiting for a response from the sensor, no additional requests are issued towards it: as soon as the measurement is available, it is distributed as response to all the queries arrived in the interval. This caching mechanism mitigates both the latency introduced by the sensor and by the network connection between the central node and the sensor host. Figure 5.3 reports a timeline showing an example of this caching mechanism. In order to forward the values produced by push sensor to each host connected to it, this module provides a method invoked by the sensor every time a new measurement is available.

This *calvinsys module* is also in charge of attaching a logic timestamp to each measurement coming from sensors: in this way it is possible to have temporal references without any effort to synchronize clocks among hosts. This module adds the *idros.host-central-sensors-manager* tag to the capabilities of the Calvin runtime, in order for actors to be correctly deployed into it.

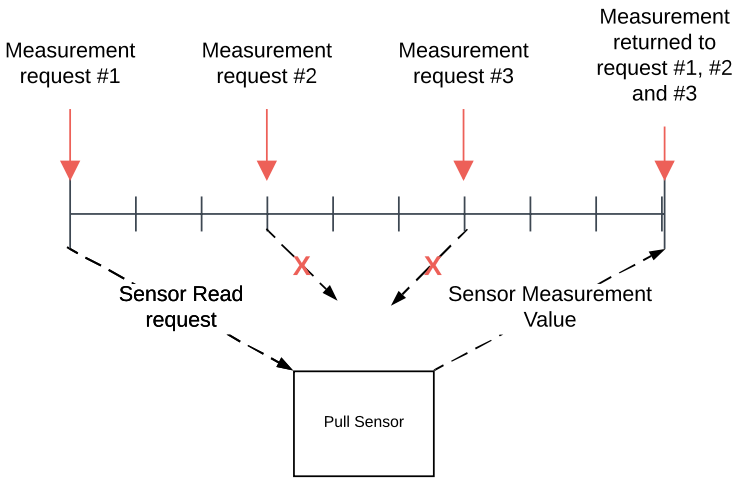


Figure 5.3: Example timeline of sensor requests caching. Requests #2 and #3 are not issued towards the sensor because the module is still waiting for the response of request #1. When the measurement is ready, it is returned to all the requests.

The Central Bus Manager module implements the functionality described in Section 4.3 for the Global Communication Bus Manager module from the architecture: it works as repository of existing bus channels and it keeps the list of memberships. It implements the blacklisting and whitelisting that prevent hosts to join a specific bus channel if they are not intended to. It provides a method to retrieve channel members and one to send messages. This module adds the *idros.host-central-bus-manager* tag to the capabilities of the Calvin runtime.

Both the *calvinsys modules* we described are distributed as part of NG IDrOS: at startup, it is checked if these two modules are already present in the Calvin runtime, otherwise they are installed.

5.1.4.2 Implementation of Calvin Actors

The second step was to implement the calvin actors needed to connect different NG IDrOS instances. We developed nine actors in total: five related to remote sensors, two to computation offloading and two to the remote bus. The actor model of Calvin led us to develop the entire network protocol using an asynchronous approach: each network interaction does not block the execution, and callbacks are triggered once results are ready. This is an approach we already declared as the best candidate to implement a network protocol before choosing Calvin, since it best fits the needs of a highly distributed environment.

Remote Sensors

For Remote Sensors capabilities we had to implement actors able to connect clients to the NG IDrOS Central instances where the Central Sensors Manager module is running. Figure 5.4 shows a scenario where all the Calvin actors for Remote Sensors are present. Figure 5.5 shows the connections between the input and the output ports of the actors.

For this purpose, the first actor we implemented was the *Central Sensors Manager Connector*. This actor is deployed into the Calvin runtime of an NG IDrOS Central instance by each NG IDrOS Drone client that wants to connect to it. It requires the specific capability tag *idros.host-central-sensors-manager* because, once deployed, it connects to the Central Bus Manager module. This actor has four input ports:

- *register* When the descriptor of a sensor is sent to this input port, the sensor is added to the remote sensors list of the Central Sensors Manager, and mapped to its hosting client.

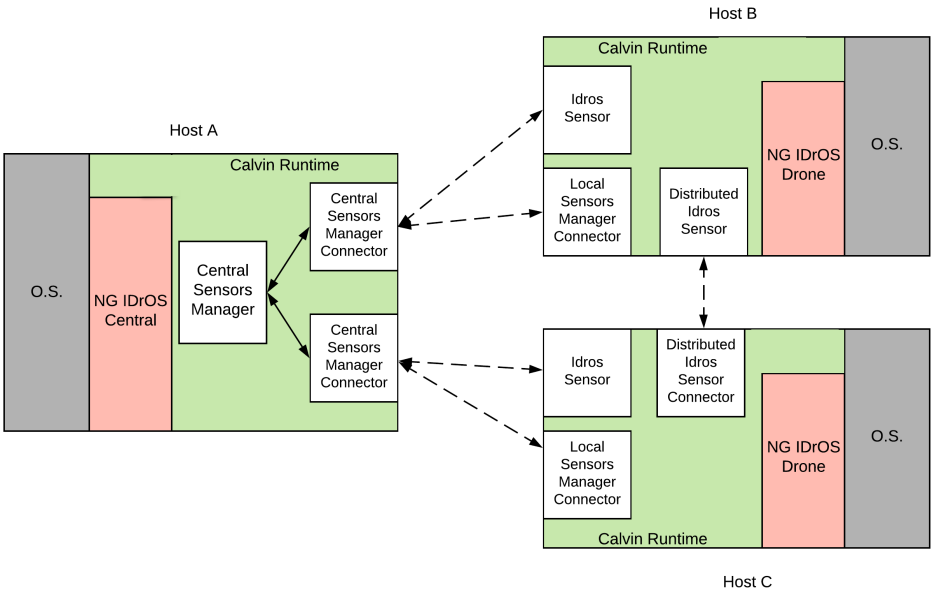


Figure 5.4: Host A runs an instance of NG IDrOS Central and hosts a Central Sensors Manager *calvinsys module*. On the same runtime, there are two Central Sensors Manager Connectors actors, one deployed by Host A and one by Host B. Both Host A and Host B have a remote sensor that is remotely exposed and is connected to the NG IDrOS Central via the Idros Sensor actor. Host C is connected to the remote sensor of Host B in a distributed manner, for this reason the Distributed Idros Sensor Connector actor is deployed on C, and the Distribute Idros Sensor actor is deployed on B.

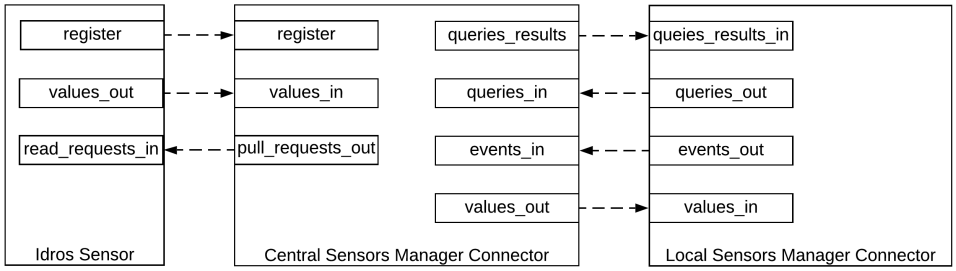


Figure 5.5: Calvin actors connection for centralized remote sensors.

- *queries_in* When a query is sent to this input port, it is resolved by searching into the list of remote sensors of the Central Bus Manager.
- *events_in* Inputs on this port are able to trigger three distinct events into the Central Bus Manager: remote sensor connection, remote sensor disconnection, and remote sensor read request. Each of these events is distinguished by a specific payload.
- *values_in* This port is used by remote sensors to send their measurements. Push sensors autonomously publish measurements over this port based on their frequency, while pull sensors will publish a measurement only after a specific request.

These are the input ports of the actor:

- *queries_result_out* This port is used to send out the results of a query.
- *pull_requests_out* This port is used to issue measurement requests against pull sensors.
- *values_out* This port is used to sent push sensors measurements to all the clients that are connected to them.

To allow an NG IDrOS Drone instance to communicate with the Central Sensors Manager Connector actor, we developed the *Local Sensors Manager Connector* actor. This actor has two input ports and two output ports:

- *events_out* This output port is used to forward three kinds of event:
  - Connect to a remote sensor;
  - Disconnect from a remote sensor;
  - Get a sensor measurement.
- *queries_result_in* This input port is used to receive results from remote sensors queries.

- *values_in* This input port is used to receive measurement values after read requests.

- *queries_out* This output port is used to send remote sensors queries.

A Calvin actor is also deployed for each sensor that is available for remote access. This actor is deployed on the same host where the sensor is physically hosted, and it is connected to the Central Sensors Manager Connector on the NG IDrOS Central. These are the input and output ports of the actor:

- *read_requests_in* This port is used to receive read requests for the sensor.

- *register* This port is used to register the sensor at the central node by means of the sensor descriptor.

- *values_out* This port is used to send sensor measurements. Push sensor autonomously send measurements based on their frequency of the sensor. Pull sensors send measurements only after an explicit request.

These three actors described so far implement all the functionality needed for the centralized connection mode of remote sensors. Two more actors were developed to support the distributed remote sensors connection. Figure 5.6 shows the connections among the ports of these two actors.

The *Distributed Idros Sensor* actor is deployed on the same host where the sensor is connected, and works as an interface towards it. This actor has only one input port and one output port:

- *read_requests_in* This input port is used to receive read requests. It is used for pull sensors only.

- *values_out* This output port is used to send sensor measurements once they are ready.

The actor to be deployed on the host that wants to connect to the remote sensor is named *Distributed Idros Sensor Connector* and has one input port and one output port:

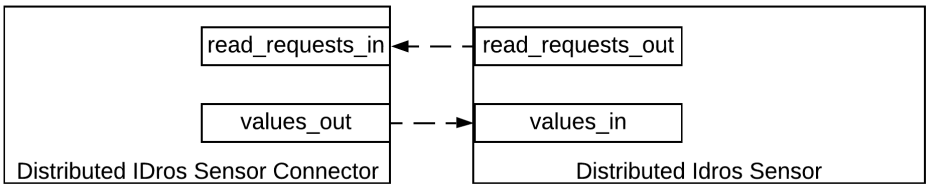


Figure 5.6: Calvin actors connection for distributed remote sensor connection.

- *values_in* This input port is used to receive measurements from the sensor.
- *read_requests_out* This output port is used to issue measurement requests towards pull sensor.

Computation Offloading

For computation offloading we developed two actors: one is deployed on the remote host that receives and runs the function, the other one on the offloader host. No *calvinsys modules* are needed to extend the Calvin base runtime capabilities for this functionality because it can be implemented just using Calvin actors. Figure 5.7 reports the deployment setup of the two actors, as well as the connections between their ports.

Computation Receiver is the actor deployed on the remote host. It implements the functionality needed for the *Offloading Manager* component of the NG IDrOS Central architecture, described in Section 4.3. The function to be offloaded is directly injected into the actor by means of generative programming techniques [42]: the source code of the function to be offloaded is extracted from the offloader host, manipulated and injected into the Python code of the actor. The manipulation is needed to extract the variables that form the state of dynamic functions. In order to identify the variables that form the internal state of a function, we implemented a custom Python type hint named *makeglobal*. Each variable marked with this type hint is added to the state of the function, and its value will be transferred back and forth when the function is offloaded and inloaded.
Listing 5.10 is an example of a function to be offloaded. Functions can be offloaded by marking them with the specific *@offload* decorator (line 1) or by passing them to the method exposed by the Computation Offloader class. When a function is marked with the decorator it is offloaded at startup time.

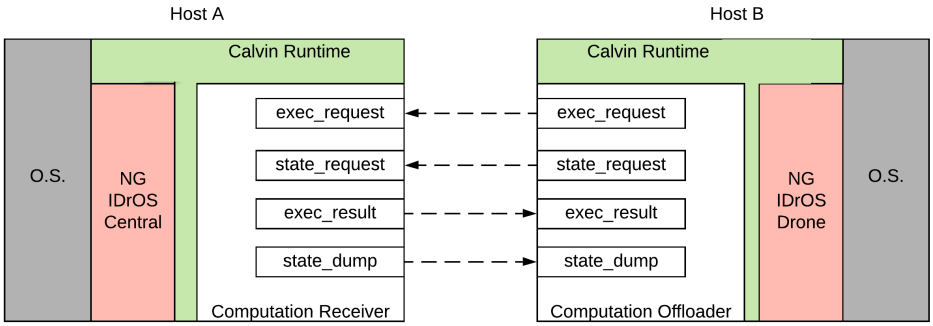


Figure 5.7: Calvin actors deployment and connections for computation offloading capability.

Using the method exposed by the Computation Offloader the function can be offloaded at any time during the execution of the application.

```
1 @offload
2 def function(input_params){
3     state_variable : makeglobal = value
4     non_state_variable = value
5
6     //Function logic
7 }
```

Listing 5.10: Source code example of function to be offloaded

This actor has two input ports and two output ports:

- *exec_Request* Tokens on this input port trigger the execution of the offloaded function. If the method has input parameters, they are sent as part of the request.
- *state_request* Tokens on this input port will trigger the output of the state of the function on the corresponding output port. This port is relevant for dynamic functions only.
- *exec_result* This output port is used to send execution results following a remote invocation.
- *state_dump* This port is used to send the state of the function after a dump request. This port is relevant for dynamic functions only.

The second actor we implemented is named *Computation Offloader* and it is deployed on the host that requests the computation offloading. It mirrors the ports of the Computation Receiver actor, as it is reported in Figure 5.7.

Communication Bus

In order to develop the functionality related to the communication bus, we implemented two actors: one is deployed on the NG IDrOS Central instance that runs as coordinator, the other one is deployed on the client that accesses the bus. The NG IDrOS Central instance holds an actor for each clients that is connected to the bus. Figure 5.8 represents the deployment and connections between these two actors.

The *Central Bus Connector* requires the capability tag *idros.host-central-bus-manager* to be deployed on a Calvin runtime because it establishes a connection towards the Central Bus Manager *calvinsys module* that effectively implements the bus coordinator. This actor has two input ports and one output port:

- *event_in* Tokens on this input port trigger three different events, depending on the payload:
  - Create a new bus channel, optionally specifying a blacklist or a whitelist of clients;
  - Join a bus channel, specified by the channel name;
  - Leave a bus channel, specified by the channel name.

  When a create, join, or leave request is received, it is forwarded to the Central Bus Manager module, where the application logic is actually implemented.

- *msg_in* This input port is used to receive messages for a specific bus channel. When an inbound message arrives, it is forwarded to the Central Bus Manager module.

- *msg_out* This output port is used to forward messages to all the members of a channel. Tokens on this port are triggered by the Central Bus Manager module, which keeps the list of participants to each channel.

The *Bus Connector* actor is instead deployed on the host that is connecting to the bus, and it mirrors the ports of the Central Bus Connector, as it is shown in Figure 5.8.

5.1.4.3 Implementation of automatic Calvin Script generation

The third step of the implementation phase addressed the development of NG IDrOS capabilities to autonomously write Calvin Script applications. The approach we followed derives from the generative programming approach [42]: actors are declared, and their ports are connected programmatically, then the application is deployed in the mesh of nodes using the local Calvin runtime as starting point. This generative programming approach

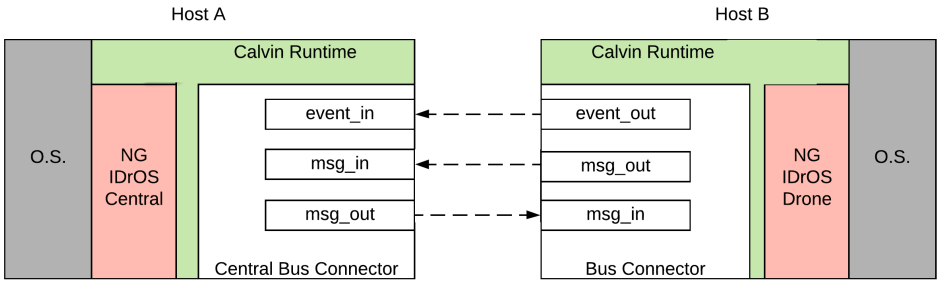


Figure 5.8: Calvin actors deployment and connections for communication bus functionality.

is used also to declare where actors must be deployed by generating a requirements JSON file, deployed together with the application, that specifies the deployment requirements. Deployment requirements can be expressed both by specifying the capability tags that the hosting runtime must have, or by directly specifying the target host by its identifier. The first approach is used to deploy actors on the Calvin runtime of the NG IDrOS Central instances. The second one is used to force the deployment of an actor on a specific runtime, and to inhibit its migration towards other runtimes.
This generative programming approach allows to create Calvin applications that exactly fits the application logic needs, and consequently to deploy only the actors that are really needed. For example, no applications are created and no actors are deployed if the drone mission does not include any function to be offloaded.

The deployment of the application is performed using the *csdeploy* command provided by the Calvin base runtime. This command takes as input a Calvin Script file and a JSON requirements file. The output of this command is captured and parsed to detect errors and implement corrective actions. At this point, the built-in capability of Calvin deploys the actors on the runtimes present in the network that satisfy the requirements. If the deployment fails because not all the requirements can be fulfilled — e.g. because the mesh of nodes is not complete, or because it doesn't include suitable nodes at all — an error is returned and corrective actions can be taken by the application logic.

5.1.4.4 Implementation of the Abstract Classes

The fourth and last step of the implementation phase addressed the implementation of the abstract classes we described in Section 5.1.2: we included everything in a single class containing the methods from all the abstract ones, and named it *Calvin Remote*. In order for this class to communicate with the actors deployed on the local Calvin runtime, we implemented a Remote Method Invocation (RMI) mechanism over the loopback interface using the Pyro4 Python library [43].
We decided to use this library because it reduces the implementation effort. It is sufficient to expose the methods over the loopback interface and to invoke them as they were standard methods. This approach allows passing back and forth input values and return values, but also to propagate exceptions when they happen. Even though other lower level approaches, such as socket based ones, may grant a lower latency in this internal communication, they would have required much more implementation effort to guarantee the same capabilities. It is a small overhead we decided to pay, and that can be easily changed in the future.

5.2 Application Logic Layer

In this section we describe the modules contained in the middle layer of the NG IDrOS architecture, the *Application Logic Layer*, that is reported in Figure 5.9. This layer manages user-built modules and provides functionality to start and manage a drone mission. The *Sensors Manager* component is in charge of providing a unified access interface to both local and remote sensors, as well as to implement the requirements of the programming model described in Section 3.2.1

The *Modules Manager* provides functionality to upload, delete, and list navigation modules and data acquisition modules. These functionalities are mainly exploited through the *Remote Control* layer of the architecture, and remotely exposed through one or more network protocols. We decided to implement these upload, list and delete functionality in the easiest possible way: modules are uploaded into a specific folder, and from such specific folder they are loaded or deleted. We decided not to implement more complex mechanisms, such as local databases, because at the moment there are no additional information to be stored except for the files themselves. Both *Navigation Modules* and *Data Acquisition Modules* must implement an abstract class that prescribes the existence of the *start()* method: this is invoked by the *Mission Manager* when the mission begins. This module is also in charge of providing capabilities to upload, delete, and list sensor drivers. The exact same approach is applied to sensor drivers too: they are loaded into a specific folder of the system, and from there they are loaded into the system or deleted.

At startup time, the *Sensors Manager* module scans each file in this specific folder and instantiates the sensors. For *push sensors*, separated threads are spawned to acquire the measurements at the given sensor frequency. These values are stored into a local buffer which is used to provide values to measurement requests. This module is capable of managing local or remote sensors agnostically, since they implement two abstract classes containing the exact same methods, as it is described in Section 3.2.1.
This module keeps an internal mapping between each sensor and the quan-

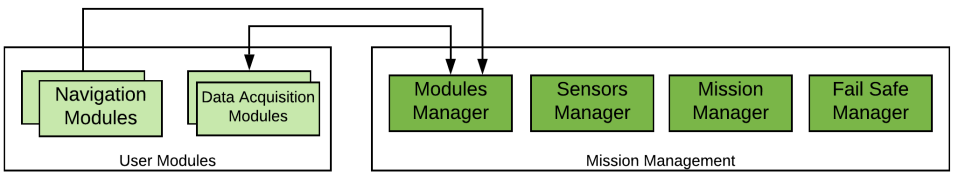


Figure 5.9: The Application Logic Layer from the NG IDrOS Architecture.

tity they measure. The idea is that application developers, in writing their missions, do not access a specific sensor, but request to the *Sensors Module* the observation of a specific quantity. Since multiple sensors may be available to provide the same quantity at the same time, this module provides capability to compare sensors and to select the best provider.
In the current NG IDrOS implementation, two sensors are comparable if they measure the same quantity, if they have the same unit of metric, and also if their accuracy is provided with the same unit of metric. The comparison is simply based on the value of the accuracy read from the sensor descriptor. Figure 5.10 reports an example highlighting the concept of comparable sensors.

The current implementation of the *Sensors Manager* supports the three working modes described by the programming model:

- *All* In this working mode, an observation is retrieved from all sensors providing the requested quantity, together with their accuracy. Comparable sensors are compared and for each tuple {quantity unit of measurement, accuracy unit of measurement} the best observation is returned.

- *Conservative* In this working mode, every time a new sensor is registered to the Manager, its accuracy is retrieved and compared with the one of the other comparable sensors. The best provider is stored for each tuple {quantity unit of measurement, accuracy unit of measurement}. Only the best provider of a quantity is queried. The best provider is computed again every time the best provider is removed from the

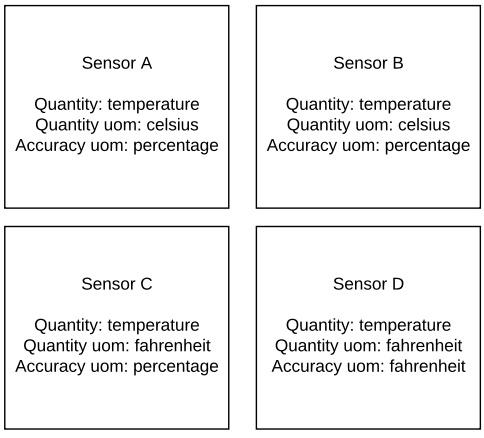


Figure 5.10: Example of Sensors Comparison. Sensors A and B are comparable because their quantity unit of measurement and accuracy unit of measurement are the same. Sensor C and Sensor D are not comparable because their accuracy unit of measurement is different.

Manager.

- *Exclusive* In this working mode, the client explicitly specifies the provider of a quantity.

The first working mode always ensures that the best providers are selected to provide a quantity. This requires to query all sensor at each request and this may cause a series of drawbacks, such as an increased power consumption or a high number of connections opened towards remote sensors. The second working mode reduces these drawbacks coming from the need of querying all the sensors, but it may cause suboptimal providers to be used as sources of observations. The last working mode gives the flexibility to the application developer to explicitly specify the provider of a quantity, actually empowering him to write custom logics to switch between providers.

The *Mission Manager* exposes five methods:

- *add_module(module)* Used to add a navigation module or a data acquisition module to the mission. This operation is permitted only when a mission is not already running.

- *remove_module(module)* Used to remove a previously added module to the mission. This operation is permitted only when a mission is not already running.

- *view_modules()* Shows which modules were added to the current mission.

- *start_mission()* Starts the mission by invoking the start() method on each navigation module or data acquisition module that was added to the mission.

- *mission_status()* Used to identify if a mission is currently running on the drone.

These methods are exploited mainly by accessing them through the Control Layer of the architecture.

The last module to be described in this layer is the *Fail Safe Manager*. We inherited this module from Cantoni's IDrOS implementation [26] and we limited our efforts in making it compatible with NG IDrOS, since it already provides the functionality needed. This module offers two methods that are able to override all the commands sent by the navigation module: one to land the drone, one to make him return to the takeoff position. The purpose of this module is to provide functionality to stop the execution of any kind of mission and to preserve the integrity of the UAV.

5.3 Remote Control Layer

This layer of the architecture provides a remote interface that can be accessed through different internet protocols. This interface exposes the modules of the Application Logic Layer and can be leveraged to create machine-to-machine interactions with other systems, as well as to remotely control the drone and its mission.

The original IDrOS implementation includes a binding to two different internet protocols: MQTT and CoAP. It provides a sophisticated mechanism based on handlers to parse incoming requests.

Since this layer does not represent the core part of this thesis work, we decided to reduce the implementation effort for this layer and to simplify it with respect to the first version of IDrOS.
We removed both the MQTT and CoAP binding, and we created a very simple socket server to handle requests. This socket server accepts requests built using the JSON format [44] with a very specific structure, which is directly mapped to the methods exposed by the modules of the *Application Logic Layer*. In this JSON payload, it must be specified which is the target module, which method of this module must be invoked, and which are the input parameters. Listing 5.11 reports the structure of the JSON format, while 5.12 reports an example of a payload to start the drone mission after 5 seconds.

```
{
    module: "module",
    method: "method",
    input_parameters: [
        {
            name: "name",
            value: "value"
        },
        ...
    ]
}
```

Listing 5.11: JSON structure for Remote Control Layer payloads.

```
{
    module: "MissionManager",
    method: "start_mission",
    input_parameters: [
        {
            name: "t_offset",
            value: "5000"
        }

    ]
}
```

Listing 5.12: Example of JSON payload

Chapter 6

Evaluation

In this chapter we present the evaluation tests we designed and executed to measure the performance of the network protocol we implemented using Calvin. We decided to focus the evaluation phase on the network protocol because the performance of this architectural component influences the performance of the entire software platform that is built around it.

To perform the experiments, we decided to avoid the usage of simulators because they would not allow to test the actual implementation of NG IDrOS, but they would require to rewrite entire parts of the software.
We decided to perform the experiments by deploying the real NG IDrOS implementation on a real hardware. The testbed we built, reported in Figure 6.1, simulates a real-world deployment setting of NG IDrOS:

- Host A is an instance of NG IDrOS Central running in the Mobile Edge Computing infrastructure, providing storage and computation resources.
- Hosts B and C are two instances of NG IDrOS Drone running a mission.
- Host B and C can not directly communicate because they are in isolated networks. They leverage Host A as bridge between the two networks.
- They are connected to the internet using a 5G connection.

This testbed simulates one out of all the possible deployment scenarios for NG IDrOS, and it includes only three hosts. The drawback of running the tests on real hardware is the limited scalability that prevented us to test different deployment settings or scenarios including more than 3 hosts.

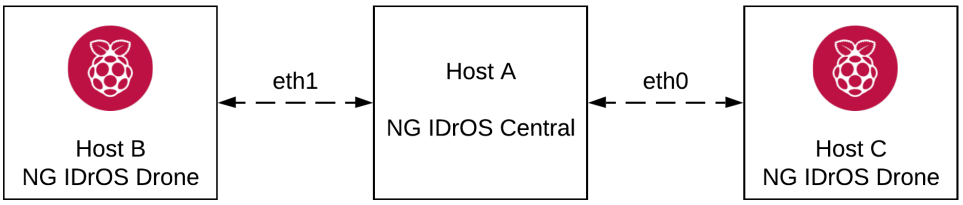


Figure 6.1: The testbed used for Calvin network protocol evaluation.

We used the *NetEm* tool available on Linux to simulate the performance of the 5G cellular network [45]. This tool allows to software simulate network conditions by imposing several parameters, such as:

- The baseline latency;
- The packets loss rate and its distribution;
- The packets corruption rate;
- The packets duplication rate.

Specifically, we imposed over Eth1 and Eth0 connections of Host A a latency of 4 milliseconds, which is the upper-bound of the theoretical target latency of 5G, as we reported in Section 2.1.2. We leveraged the same tool also to simulate the packet loss over the links: we tested every scenario imposing a packet loss rate from 0% to 10% with a 2% step and a random distribution. We decided to limit our investigations to a maximum of 10% because it is expected to be the maximum packet loss rate of the radio link at around 1000 connected clients [46].

Also the hardware equipment of the hosts have been chosen to simulate the performance of the deployment setting we described.
Host A is a server machine that simulates a virtual machine inside the Mobile Edge Computing infrastructure. It is equipped with:

- Intel Xeon CPU E3-1270 v5 @ 2.60GHz.
- 64GB DDR4 RAM @ 2400 MHz.
- 2 x Gigabit Ethernet network card adapters.

Hosts B and C are two Raspberry Pi model 3B+, which is a common companion computer for UAVs. This specific model of the Raspberry is equipped with:

- ARM Cortex-A53 @ 1.4GHz.
- 500MB SDRAM.

- Gigabit Ethernet network adapter.

Since most of the performance metrics are related to latency measures, the clocks of the three hosts must be perfectly synchronized. For this purpose, we configured Host A to be an NTP server [47] for the two networks. This guarantees an accuracy in clock synchronization of ± 0.3 milliseconds.

The following sections describe the tests and performance metrics we measured for the three features we implemented to exploit the high-performance networks: *Remote Sensors*, *Computation Offloading*, and *Communication Bus*.

6.1 Remote Sensors

The performance metric we want to measure to evaluate the performance of the remote sensors functionality is the latency overhead introduced by the NG IDrOS stack. The sum of the latency introduced by the software and the latency introduced by the network transport is the end-to-end latency of a remote sensor read. The purpose is to verify if the end-to-end latency is suitable for real-time remote sensor exploitation.

To perform this experiment, we implemented a software emulated sensor with a zero second interval, that is able to output a measurement immediately after a request. This sensor is physically connected to Host B and it is remotely exploitable. We then implemented a drone mission to run on Host C that connects to this sensor and requests measurements. Since the two hosts are in two isolated networks, each measurement request is sent to Host A and is then forwarded to the sensor. In the same way, each output value is first sent to the central node and is then returned to the application. This deployment setting is inspired by a real-world deployment in which an instance of NG IDrOS Central acts as bridge between two isolated networks to form a single mesh of nodes.

In this scenario, each measurement request goes through two hops to reach the sensor. In the same way, each measurement value goes through 2 hops to reach the application. Hence, the baseline latency added by the network transport is equal to $4x4$ milliseconds = 16 milliseconds.

We implemented the drone mission to execute from one to 250 measurement requests per second with a step of five, and logged the time interval between the read request and the moment the sensor value was returned to the application. We executed each experiment by imposing a packet loss rate in the interval [0%, 10%] with a step of 2%. To give a better statistical relevance to the results, we repeated each experiment 5 times. Table 6.1 and Figure 6.2 report the results averaged over the 5 repetitions.

Packet Loss [%]	Amount of Remote Reads	Average Latency [ms]	Standard Error [ms]
0	18375	26.89	0.153
2	18375	27.80	0.152
4	18375	28.20	0.153
6	18375	28.75	0.157
8	18375	29.12	0.147
10	18375	29.93	0.158

Table 6.1: Remote sensor read experiments results.

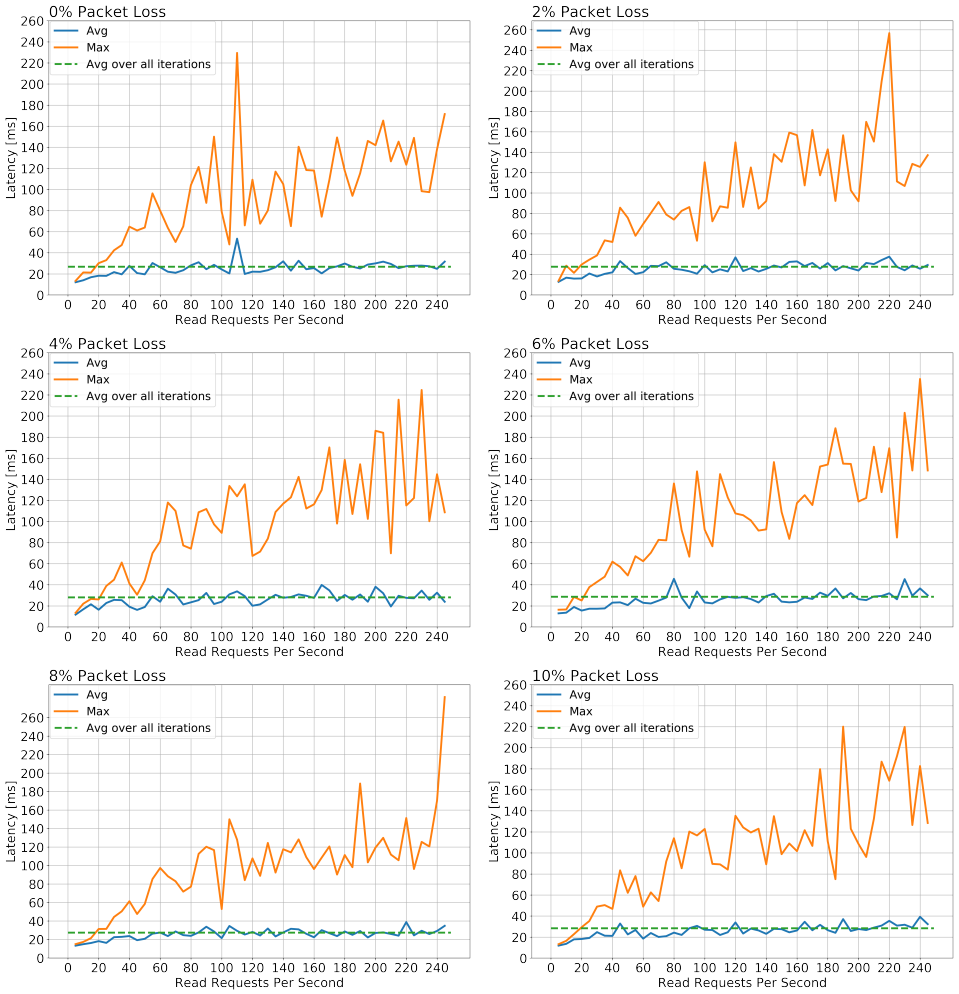


Figure 6.2: Remote sensor read experiments results.

From these results we derive the following conclusions:

- The average end-to-end latency of a remote sensor read, in the packet loss rate interval we tested, is $\sim$ 28 milliseconds. Being the network transport baseline latency equal to 16 milliseconds, we can conclude that the software stacks accounts for the 43% of the end-to-end latency.
- The end-to-end latency in the interval we tested is slightly sensitive to the packet loss rate. The end-to-end latency measured at 10% packet loss rate is 11.3% higher than the one measured at 0% packet loss rate.
- The end-to-end latency in the interval we tested is slightly sensitive to the read requests rate.

The root cause for the last two behaviours listed above is the double caching mechanism that is implemented in the software stack: both at NG IDrOS Central and at NG IDrOS Drone level, when observation requests arrive while the system is already waiting for the response of a previous request, no new queries are issued towards the sensor. As soon as the measurement is available, it is returned to all the pending requests. This mechanism is able to mitigate the latency introduced by both the TCP retransmissions caused by the packet loss and by the high number of read requests in parallel.

6.2 Offloading of Static Functions

We decided to evaluate different performance metrics for the offloading of static functions and for the offloading of dynamic functions. For this reason, we implemented two separate experiments to test the two features.

For static functions offloading, we decided to evaluate two metrics:

- The execution time at increasing invocation rate.
- The execution time at increasing network packet loss rate.

The purpose is to verify if there are conditions in which the overhead introduced by the software stack makes the offloading disadvantageous with respect to a local execution.

The experiment we designed involves Host B and Host A: the first one offloads the function, the second one receives and executes it. This experiment is inspired by a real use case, where a low computational power host exploits a more powerful one to execute a complex part of the application logic.
The function to be offloaded was designed to emphasize the performance difference between the hosts. Listing 6.1 reports the function we implemented, while Table 6.2 reports the average execution times of the function for each host. The execution of the test function on the Raspberry Pi is on average 15.5 times slower than its execution on the server machine.

```
1 def test():
2     var acc = 0
3
4     for i in range(0, 4000):
5         for q in range(0, i):
6             acc += (i*i*i)
7
8     return acc
```

Listing 6.1: Evaluation function for static computation offloading.

Host	Number for executions	Average execution time [s]
Host A	1000	0.45
Host B	1000	7

Table 6.2: Average execution times of test function on Host A and Host B.

Since the execution time on the Raspberry Pi is 7 seconds, we tested the invocation rate in the range [1, $60/7 = 8$] invocations per minute. We repeated each experiment imposing a network packet loss rate in the range [0%, 10%] with a 2% step. We measured the time interval between the remote invocation and the moment the result value was returned. In these measurements, we have not included the time needed to offload the function — i.e. to transfer the source code — since the test function is static and does not include an internal state which size may influence the offloading

time. Moreover, the offloading of the source code is an operation executed only once during the lifetime of a NG IDrOS application. The evaluation of the offloading and inloading times is performed in section 6.3 for dynamic functions.

The graphs reported in Figure 6.3 show the result of the experiments. It is clear that it is always convenient to offload the function instead of running it locally. The benefit is higher with the increasing number of invocations per minute because the server machine has more cores to handle the executions in parallel.

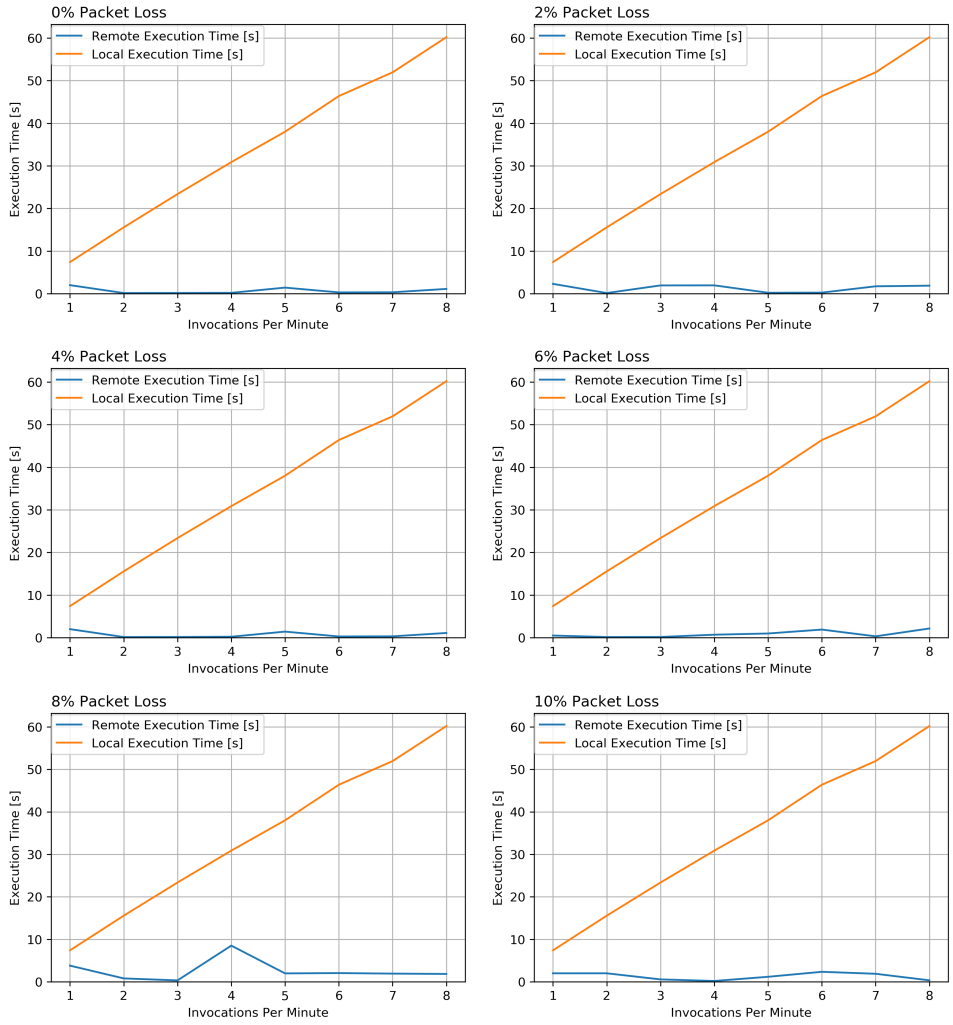


Figure 6.3: Comparison between the local execution time and the remote execution time of the test function.

The packet loss rate does not significantly affect the end-to-end execution time because the network time, including the TCP retransmissions, is negligible with respect to the execution time. On average, the execution time accounts for the 95% of the end-to-end latency.

To further validate these results, we decided to compare the network protocol with a classic Remote Method Invocation implementation. We used the Python library *Pyro4* to remotely expose the test function from Host A to Host B, and we tested the same execution rate range imposing a fixed 4% network packet loss rate. As per the original experiment, the measurements do not include the time needed to offload the source code. In fact, in this experiment the source code to be executed is directly hosted in the server machine and it is only remotely invoked.

The results of this comparison experiment are reported in Figure 6.4: the performance of NG IDrOS is slightly better than the performance of Pyro4. On average, the execution time on NG IDrOS is ∼450 milliseconds shorter than the execution time on the Pyro4 based application. To have a better statistical evidence of this comparison, we performed the experiment 10 times and averaged the results. The standard error of NG IDrOS mean is 0.169 seconds, the standard error of Pyro4 mean is 0.33 seconds.

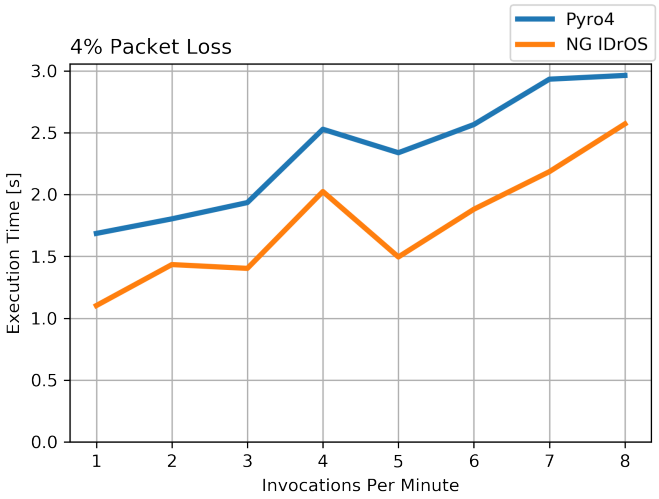


Figure 6.4: Comparison between NG IDrOS network protocol and a Remote Method Invocation implementation based on Pyro4.

6.3 Offloading of Dynamic Functions

The performance metrics we decided to measure to test the offloading of dynamic functions are:

- The offloading time, that is the time needed to move the source code and the function state towards the receiver host.
- The inloading time, that is the time needed to acquire back the internal status of the function.

The purpose is to verify how time performance is impacted by the size of the internal state and by the packet loss. In particular, when the state is moved among peers it requires some time to be serialized and then deserialized and restored. The latency introduced by these operations sums with the latency introduced by the network transport. Moreover, while the offloading of the source code is an action that happens few times — in most cases just once — during the lifecycle of a NG IDrOS application, the state may be transferred multiple times. To measure these metrics, we modified the experiment described in Section 6.2 by adding an internal state to the test function. The pseudocode of the final evaluation function we used for the experiments is reported in Listing 6.2. The function logic is not influenced by the internal status in order to keep its execution time constant: this performance metric is not relevant for these experiments, while it was analysed in Section 6.2.

```
1 def test():
2     internal_status = [Int] * (StateSize)
3     acc = 0
4
5     for i in range(0, 4000):
6         for q in range(0, i):
7             acc += (i*i*i)
8
9     return acc
```

Listing 6.2: Evaluation function for dynamic computation offloading.

To simulate the internal status of the function, we implemented an array of variable size (line 2). In our experiments, we increased the dimension of the array from 0 to 1000 integer elements, varying its size from 0 to 4000 bytes. We tested how the offloading time and the execution time change also with respect to the packet loss rate in the range [0%, 10%] with a 2% step. The results of the experiments are reported in Figure 6.5.

From the graphs, we can derive the following conclusions:

- The number of state variables in the range we tested does not significantly affect the inloading and offloading times, since there is no increasing trend in the average times.

- The average inloading time is higher than the average offloading time because it is influenced by the time needed to restore the variables, and in our experiments it is driven by the performance of the Raspberry.
- The packet loss rate in the interval we evaluated does not significantly affect the inloading and offloading times, because the network transfer time, including TCP retransmissions, is negligible with respect to the time needed to serialize, deseralize and restore the status.

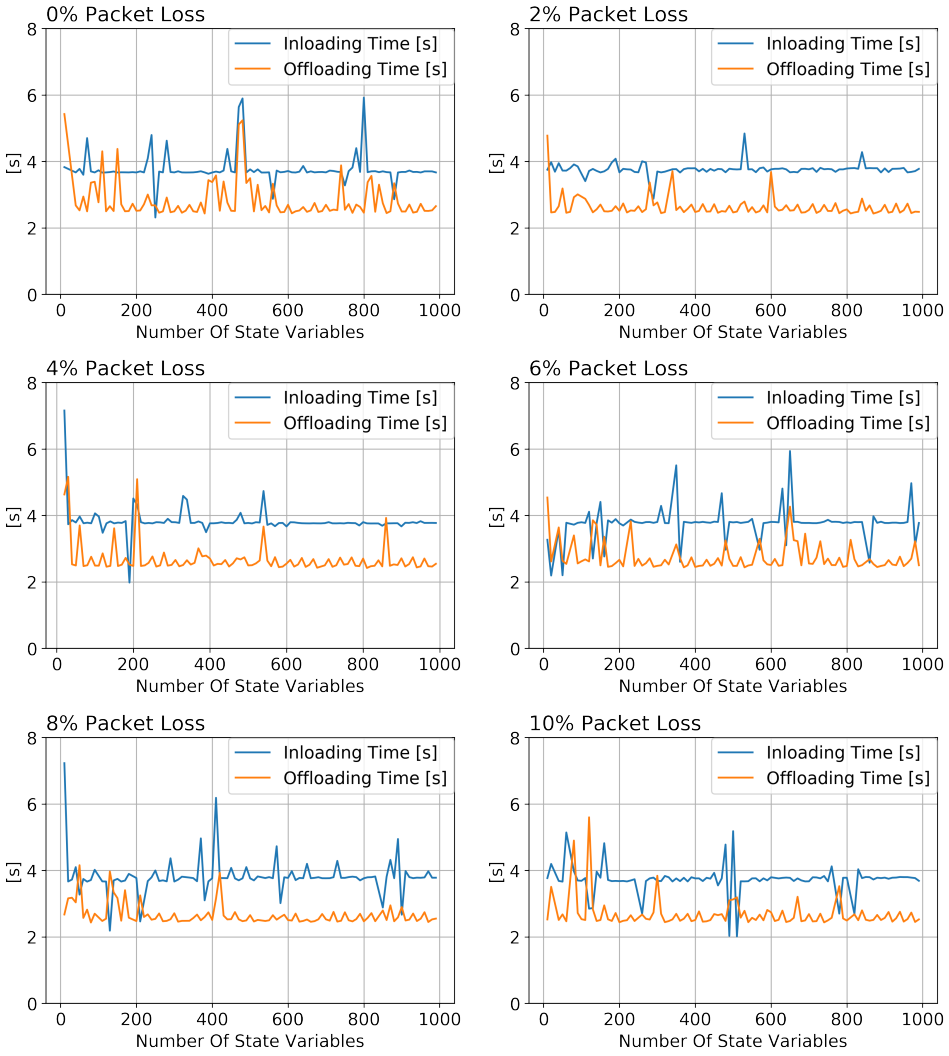


Figure 6.5: Evaluation of Inloading Time and Offloading Time with respect to state size and network packet loss. Deployment errors are not shown.

We also compared the inloading time and offloading time with respect to a standard Remote Method Invocation application. We used the Pyro4 library to expose, from Host A to Host C, three functions:

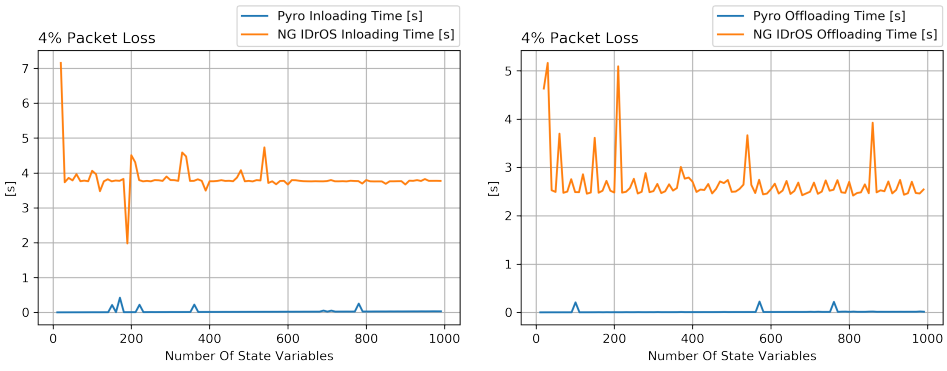
- One to transfer the source code of the function, together with its internal status;
- One to remotely invoke the source code;
- One to dump the internal status of the function.

We executed the comparison imposing a fixed 4% network packet loss rate, and the results are reported in Figure 6.6. From the graphs, we can conclude that the performance of NG IDrOS are four times worse than the Pyro4 implementation. We identified two main root causes for this:

- The endpoints used in the Pyro4 implementation are statically typed into the application, while Calvin, before starting the offload, dynamically deploys the actors. The deployment phase is the major factor that causes this performance gap.
- Pyro4 has a better serialization and deserialization mechanism to transfer the variables composing the status of the function.

These experiments allowed us to evaluate also the reliability of the deployment mechanism of Calvin actors. We stressed the Calvin runtime by offloading, running, and inloading functions for 1000 times in a row and in a short time range.

Table 6.3 reports the occurrences of deployment failures with respect to the size of the function state and the packet loss. The maximum failure rate in our experiments is equal to 0.006%. The failure rate, although it is very low, is correlated with the increase of the packet loss rate.



(a) Inloading Time Comparison. (b) Offloading Time Comparison.

Figure 6.6: Comparison of Inloading and Offloading times between NG IDrOS and comparison application implemented using Pyro4.

	0% packet loss	2% packet loss	4% packet loss	6% packet loss	8% packet loss	10% packet loss
0-200 variables	2	1	1	2	2	2
200-400 variables	0	1	0	2	1	1
400-600 variables	0	0	0	1	1	0
600-800 variables	0	0	0	0	0	1
800-1000 variables	0	0	0	1	1	1
Total	2	2	1	6	5	5

Table 6.3: Occurences of Calvin application deployment failure with respect to number of state variables and network packet loss.

6.4 Communication Bus

To evaluate the implementation of the network protocol that addresses the communication bus functionality, we decided to compare the performance of NG IDrOS with a publish-subscribe application.
The performance metric we decided to measure is the end-to-end transmission time of a message, that is the time difference between the moment a message is received and the moment it was sent. We excluded from the analysis the creation of the bus channel and the channel join, because they are operations executed only once in the lifetime of an application. We designed an experiment to evaluate how this performance metric is impacted by the number of messages sent per second and by the network packet loss rate.

On NG IDrOS side, we implemented two drone missions:

- *Mission A* At startup time, this mission creates a bus channel with a specific name and waits for a member. As soon as a new member joins, it starts sending messages over the bus. It sends from one to 250 messages per second and logs the timestamp of each message right before sending it.

- *Mission B* At startup time, this mission joins a specific bus channel and waits for messages. It logs the time for each received messages.

We deployed Mission A on Host B and Mission B on Host C. The message broker is provided by the NG IDrOS Central instance on Host A. This de-

ployment scenario is inspired by a read world scenario in which the message broker is hosted on a server machine — e.g. inside the Mobile Edge Computing infrastructure — that works as bridge between two isolated networks.

To implement the comparison publish-subscribe application, we have chosen to rely on the MQTT protocol. In the same deployment setup of Figure 6.1, we hosted a MQTT broker on Host A, and two MQTT clients on Host B and C. For the broker we used *mosquitto* [48], which is one of the most used implementation of the MQTT broker. For the two clients, we implemented a sender and a receiver application based on the *paho mqtt* [49] Python library. We set the Quality Of Service for the message exchange at level 2. This QOS level guarantees the highest reliability in MQTT, since it ensures that each message is received once and only once by each client subscribed to a topic. The behaviour of the application is exactly the same of the NG IDrOS mission.

In this deployment scenario, the hops count for each message is 2. Hence, the baseline network latency is equal to $2x4$ milliseconds = 8 milliseconds. Figure 6.7 reports the result of the experiments. From the graphs we can conclude that:

- The MTTQ implementation is 2 times faster than the NG IDrOS implementation.
- Both the applications show a slightly increasing trend related to the increase of the number of messages sent per second, present in all the graphs except for the 0% packet loss rate.
- Both the applications show an increasing trend in the average end-to-end transmission time related to the increase of the packet loss.

The root causes we identified for the behaviours described above are:

- MQTT performs better than NG IDrOS because Calvin exchanges tokes among actors with a polling mechanism: the presence of tokens at input ports is checked periodiocally at a given interval. This introduces a small delay from the moment a token is available on the port and moment the corresponding action is triggered. Moreover, this is impacted by the performances of the Raspberry Pi, since a thread is spawned to check over each input port.
- Both in MQTT and NG IDrOS, the end-to-end time is slightly affected by the message rate because both the applications rely on the TCP/IP protocol, which guarantees the in-order delivery of messages. This implies that with a high rate of messages per second, if one is lost and needs to be retransmitted it will delay the delivery of all the following ones.

- Both in MQTT and NG IDrOS, the end-to-end time is sensitive to packet loss because they both rely on the TCP/IP protocol. A higher packets loss leads to an increase of packets retransmission. However, the impact on the end-to-end latency is limited because, since the connection between the clients is intermediated by the broker, the retransmission affects only one network hop.

Moreover, the average end-to-end time of NG IDrOS shows a higher variance with respect to MQTT. The cause of this behaviour is the polling mechanism of Calvin, that affects the general performance due to the high number of thread spawned.

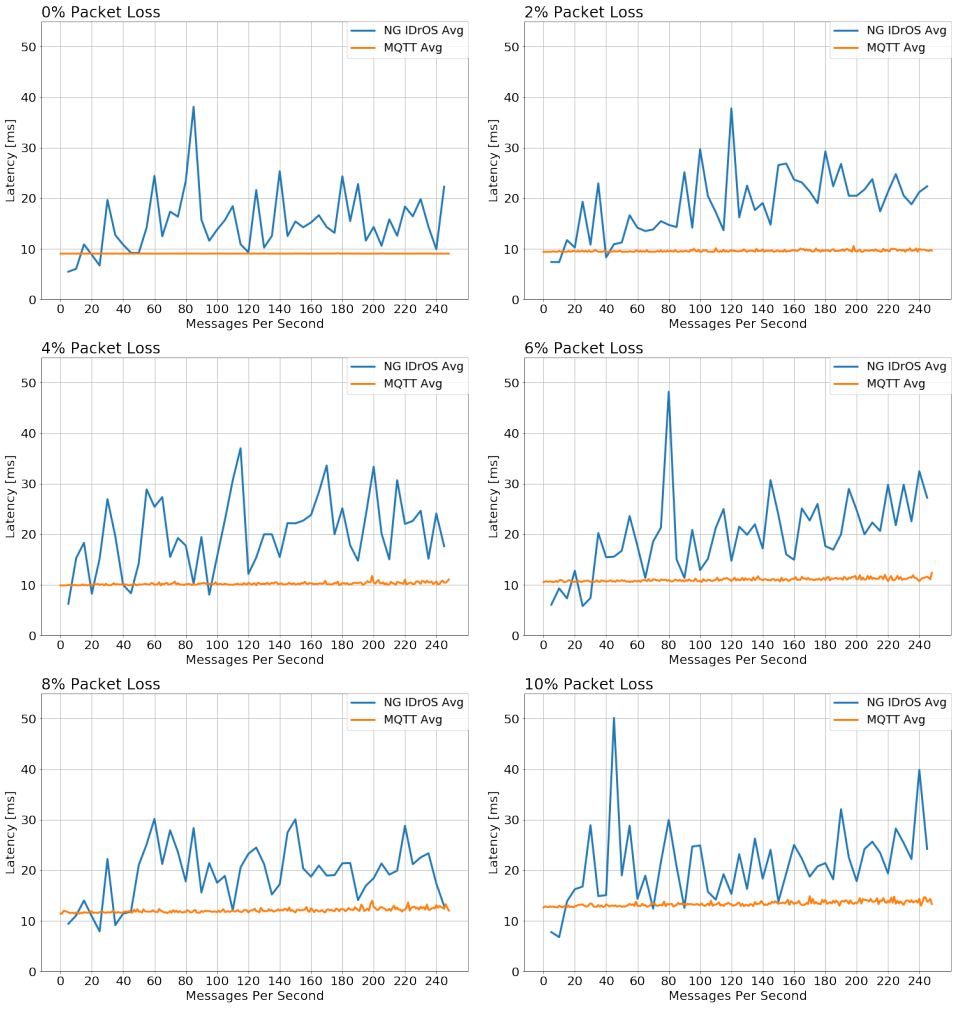


Figure 6.7: End-to-end message delivery time comparison between NG IDrOS Communication Bus and a MQTT application.

Chapter 7

Conclusion

In this thesis work we investigated the possible outcomes of the integration between high-performance networks and high-mobility robotic vehicles.
With a specific focus on UAVs and wireless networks, we explored two parallel directions: to solve the current state-of-the-art issues and limitations, and to identify new application scenarios in which UAVs can be involved.
The outcome of this analysis clearly indicates that it is not sufficient to connect UAVs to a faster network to benefit from the improved network performance. A proper software platform must support this integration. Hence, we identified three macro-functionalities for which the improved network performance represents a key enabler element.
The capability to share sensors among clients, that we named *Sensors Sharing*, mainly exploits the ultra-low latency. The capability to dynamically offload some application parts, that we named *Computation Offloading*, leverages the high bandwidth and low latency. And finally, the capability to open communication channels among clients that are steady enough to support core functionality, that we named *Communication Bus*, is mainly based on the high reliability of the network.
We derived a list of functional and non-functional requirements to describe these features. These requirements drove the development of a programming model, whose pillars are the support the extreme client mobility, the compatibility with different hardware platforms, and the exploitation of high-performance networks.
We designed the architecture of a software platform — New Generation IDrOS — to support this programming model. The core concepts underlying this architecture are the layering of the functionality and the decoupling of the components. These two characteristics guarantee the possibility to add and extend parts of the architecture without impacting the others.
This architecture then served as a blueprint to develop a working implementation of the software platform, that can be deployed on real UAVs.

The core part of this implementation is represented by the network protocol because it is directly connected to the exploitation of high-performance networks.
The network protocol we implemented is based on Calvin, an open-source application environment that mixes the Actor paradigm with the Flow Based paradigm. The core purpose of Calvin is to simplify the way IoT applications are developed, and thus to reduce the fragmentation of programming languages, protocols, and environments. We decided to build the network protocol over Calvin not only to leverage some technical aspects, but also to adhere to its philosophy: reducing fragmentation in IoT.
The evaluation experiments we performed over this network protocol demonstrated that its performance is adequate to support the purposes of the software platform, even though there are several aspects that can be improved. The latency added by the NG IDrOS stack is on average lower than the latency introduced by the network transport, and also the comparison with respect to similar implementations shows that the performance of NG IDrOS is in line with the performance of similar implementations.

There are several aspects that we set aside during this thesis work, and they represent good starting points for future works.
The principal aspect is related to security. Both the programming model and the architecture completely miss any reference to security. We believe that *security by design* is a necessity in our context and it would be of primary relevance when extending NG IDrOS.
A different aspect that can be investigated is the extension of the computation offloading model to the full code mobility paradigm. The current design of the software platform does not support the ability to migrate the computation to multiple peers, but it is limited to a one-to-one exchange. Supporting the complete offloading model would lead the way to other interesting features, such as autonomous load-balancing or failure recovery. This last one is another missing aspect in this thesis work. Especially the possibility to recover NG IDrOS Central instances from failures would be crucial in a real application scenarios.
An additional aspect we believe would be a relevant future work, is related to error detection mechanisms for messages exchange over the communication bus. Our original idea was to support the application of user-defined validation mechanisms to be applied to incoming messages. This would further improve the reliability of the communication channel and would make it even more suitable to support core parts of the applications.

Bibliography

[1] R. E. Hattachi and J. Erfanian, "NGMN 5G White Paper," tech. rep., NGMN Alliance, Feb. 2015.

[2] "Study on Socio-Economic benefits of 5G Services Provided in mmWave Bands," tech. rep., GSMA, Jan. 2019.

[3] "Understanding 5G: Perspectives on future technological advancementsin mobile," tech. rep., GSMA Intelligence, Dec. 2014.

[4] "Emerging Trends in 5G/IMT2020," tech. rep., ITU, Sept. 2016.

[5] C. Anderson, "How i accidentally kickstarted the domestic drone boom," *Danger Room Wired. com, http://www. wired. com/dangerroom/2012/06/ff—drones/all*, pp. 1–10, 2012.

[6] B. Rao, A. G. Gopi, and R. Maione, "The societal impact of commercial drones," *Technology in Society*, vol. 45, pp. 83 – 90, 2016.

[7] Yael Maguire, "High altitude connectivity: The next chapter." https://engineering.fb.com/connectivity/high-altitude-connectivity-the-next-chapter/. Accessed: 2019-10-27.

[8] S. Katikala, "Google project loon," *InSight: Rivier Academic Journal*, vol. 10, no. 2, pp. 1–6, 2014.

[9] R. J. Hall, "An internet of drones," *IEEE Internet Computing*, vol. 20, pp. 68–73, May 2016.

[10] N. H. Motlagh, M. Bagaa, and T. Taleb, "Uav-based iot platform: A crowd surveillance use case," *IEEE Communications Magazine*, vol. 55, no. 2, pp. 128–134, 2017.

[11] "UAS: The Global Perspective," tech. rep., UVS International, 2011.

[12] A. Solodov, A. Williams, S. Hanaei, and B. Goddard, "Analyzing the threat of unmanned aerial vehicles (UAV) to nuclear facilities," *Security Journal*, Apr. 2017.

[13] ITU, "Minimum requirements related to technical performance for IMT-2020 radio interface (s)," *Report ITU-R M.2410-0*, 2017.

[14] J. G. Andrews, S. Buzzi, W. Choi, S. V. Hanly, A. Lozano, A. C. Soong, and J. C. Zhang, "What will 5G be?," *IEEE Journal on selected areas in communications*, vol. 32, no. 6, pp. 1065–1082, 2014.

[15] I. Neokosmidis, T. Rokkas, M. C. Parker, G. Koczian, S. D. Walker, M. S. Siddiqui, and E. Escalona, "Assessment of socio-techno-economic factors affecting the market adoption and evolution of 5G networks: Evidence from the 5G-PPP CHARISMA project," *Telematics and Informatics*, vol. 34, no. 5, pp. 572 – 589, 2017.

[16] M. T. Beck, M. Werner, S. Feld, and S. Schimper, "Mobile edge computing: A taxonomy," in *Proc. of the Sixth International Conference on Advances in Future Internet*, pp. 48–55, Citeseer, 2014.

[17] A. Reznik, A. Sulistio, A. Artemenko, Y. Fang, D. Frydman, F. Giust, S. Lv, U. Sheikh, Y. Yu, and Z. Zheng, "MEC in an Enterprise Setting: A Solution Outline," tech. rep., ETSI, Sept. 2018.

[18] "5G vision (the next generation of communication networks and services)," *The 5G infrastructure public private partnership [Available Online at: https://5g-ppp.eu/wpcontent/uploads/2015/02/5G-Vision-Brochure-v1.pdf]*, 2015. Accessed: 2020-03-12.

[19] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenbach, and F. Giust, "Mobile-edge computing architecture: The role of mec in the internet of things," *IEEE Consumer Electronics Magazine*, vol. 5, pp. 84 – 91, Oct. 2016.

[20] M. Gharibi, R. Boutaba, and S. L. Waslander, "Internet of Drones," *IEEE Access*, vol. 4, pp. 1148–1162, 2016.

[21] K. Wyrobek, "The Origin Story of ROS, the Linux of Robotics." https://spectrum.ieee.org/automaton/robotics/robotics-software/the-origin-story-of-ros-the-linux-of-robotics, Oct. 2017. Accessed: 2020-01-26.

[22] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009.

[23] "ROS on MAVs with MAVLink." https://www.ros.org/news/2011/03/ros-on-mavs-with-mavlink.html, Mar. 2017. Accessed: 2020-01-26.

[24] S. Gupta and U. Durak, "Restful software architecture for ros-based onboard mission system for drones," in *AIAA Scitech 2020 Forum*, p. 0239, 2020.

[25] K. DeMarco, M. E. West, and T. R. Collins, "An implementation of ros on the yellowfin autonomous underwater vehicle (auv)," in *OCEANS'11 MTS/IEEE KONA*, pp. 1–7, Sept. 2011.

[26] D. Cantoni, "System Support for Internet-connected Drones," Master's thesis, Politecnico di Milano, Italia, 2015.

[27] "Dronekit." https://dronekit.io/. Accessed: 2019-02-11.

[28] H. D. Mathias, "An Autonomous Drone Platform for Student Research Projects," *J. Comput. Sci. Coll.*, vol. 31, p. 12–20, May 2016.

[29] M. Compton, P. Barnaghi, L. Bermudez, R. García-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. L. Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, and K. Taylor, "The SSN ontology of the W3C semantic sensor network incubator group," *Journal of Web Semantics*, vol. 17, pp. 25 – 32, 2012.

[30] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Transactions on software engineering*, vol. 24, no. 5, pp. 342–361, 1998.

[31] A. Banks and R. Gupta, "MQTT Version 3.1.1," *OASIS standard*, vol. 29, p. 89, 2014.

[32] https://ardupilot.org/copter/. Accessed: 2019-03-11.

[33] L. Meier, "Pixhawk." https://www.pixhawk.org, 2017. Accessed: 2019-02-24.

[34] "DJI Drones." https://dji.com/. Accessed: 2019-03-11.

[35] "MAVLink." https://mavlink.io/en/about/overview.html. Accessed: 2019-03-11.

[36] "SITL Simulator (Software in the Loop)." https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html. Accessed: 2019-03-11.

[37] "Pymavlink." https://github.com/ArduPilot/pymavlink. Accessed: 2019-03-11.

[38] "MAVProxy." https://ardupilot.github.io/MAVProxy/html/index.html. Accessed: 2019-02-11.

[39] P. Persson and O. Angelsmark, "Calvin-Merging Cloud and IoT.," in *ANT/SEIT*, pp. 210–217, 2015.

[40] C. Hewitt, "Actor model of computation: scalable robust information systems," *arXiv:1008.1459v38*, 2015.

[41] J. P. Morrison, *Flow-Based Programming: A new approach to application development*. CreateSpace, 2010.

[42] K. Czarnecki, K. Østerbye, and M. Völter, "Generative programming," vol. 2548 of *Lecture Notes in Computer Science*, pp. 15–29, Springer, 2002.

[43] "PYRO - Python Remote Objects." https://github.com/irmen/Pyro4. Accessed: 2019-03-11.

[44] "JSON." https://json.org. Accessed: 2019-03-27.

[45] S. Hemminger *et al.*, "Network emulation with NetEm," in *Linux conf au*, pp. 18–23, 2005.

[46] R. Singh and R. Garhewal, "Smart M-Health Continuous Monitoring System Using 5G Technology," Dec. 2018. Research Proposal.

[47] "Network Time Protocol project." http://www.ntp.org/. Accessed 2020-03-28.

[48] "Eclipse Mosquitto." http://mosquitto.org. Accessed 2020-03-28.

[49] "Eclipse Paho - MQTT and MQTT-SN software." http://eclipse.org/paho/. Accessed 2020-03-28.