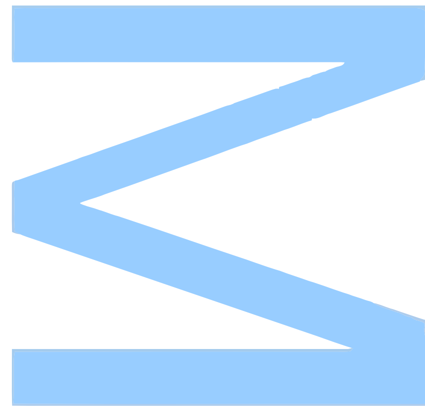
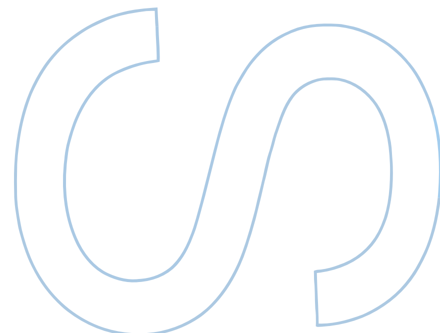
Flux:
A platform for mobile data sensing using personal devices



Nuno Miguel Alves da Silva
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2017

Orientador
Luís Lopes, Professor Associado,
Faculdade de Ciências da Universidade do Porto

Co-Orientador
Eduardo Marques, Professor Auxiliar Convidado,
Faculdade de Ciências da Universidade do Porto



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/ ______/ ___________

MSc

To my family and friends...

Acknowledgements

I would first like to thank my dissertation advisors, Professor Luís Lopes and Professor Eduardo Marques for their assistance and dedicated involvement in every step, throughout the process.

Thanks to Tadeu Freitas and Diogo Machado, for their insightful comments and encouragement.

Most importantly, I would like to thank my parents, my brother and sister for supporting me and providing a continuous encouragement throughout my years of study, with a lot of patience. This accomplishment would not have been possible without them.

The work carried out in this dissertation was supported by the SMILES project (NORTE-01-0145-FEDER-000020) / Norte2020.

Resumo

Dispositivos móveis, como por exemplo telemóveis, estão a tornar-se cada vez mais poderosos, em que combinados com o incremento do numero de sensores embutidos, expandem sua aplicabilidade. Em particular, estes sensores podem ser usados para aplicações de recolha de dados. No entanto, a utilização de dispositivos móveis para recolher dados pode revelar-se um desafio. Primeiro, temos a dificuldade de enviar tarefas de recolha de dados para os dispositivos e subsequente agregação dos dados capturados. E em segundo lugar, esses dispositivos não são dedicados para captura de dados, logo é preciso ter em conta o consumo de recursos no dispositivo. Esses requisitos foram parcialmente abordados em redes de sensores wireless, por isso torna-se relevante avaliar esses sistemas.

Nesta dissertação, descrevemos Flux, uma plataforma para captura de dados, através de tarefas de recolha de dados reconfiguradas dinamicamente em dispositivos móveis. Realiza a atribuição de tarefas periódicas em tempo real para os dispositivos presentes numa região geográfica, e faz a respetiva recolha dos dados, tornando-os acessíveis como fluxos de dados, disponibilizados por um publish/subscribe broker. As tarefas de recolha de dados são programadas usando a Flux Task Language e compiladas para byte-code que é executado numa máquina virtual de baixo consumo de recursos.

Implementamos um protótipo do Flux e avaliamos a influência do serviço nos dispositivos móveis, o que mostrou o baixo consumo de recursos. Em seguida, realizamos dois casos de estudo. O primeiro, onde um grupo de voluntários percorreu uma área de especifica usando smartphones e tablets, para recolher dados sobre do sinal da rede wireless. O segundo caso de estudo demonstrou a capacidade do sistema em reconfigurar dinamicamente as tarefas nos dispositivos, com base na sua localização.

Abstract

Mobile devices, such as smartphones or wearables are becoming more powerful, that combined with the increment of available embedded sensors, expands their applicability. In particular, sensing capabilities can be put to use for mobile data sensing applications. Nevertheless, the exploit of mobile devices for sensing can prove a challenge. First, there is the adversity of disseminating the sensing tasks and the subsequent aggregation of the captured data. And second, these devices are not dedicated for sensing, so the overhead on the device must be as low as possible. These requirements have been partially addressed in Wireless Sensor Networks so it becomes relevant to evaluate these systems.

In this dissertation we describe Flux, a platform for dynamically reconfigurable data sensing using mobile devices. It performs on-the-fly injection of periodic tasks on devices present in a geographical region, and gathers the sensing data, making it accessible as data streams by a publish/subscribe broker. Sensing tasks are programmed using the Flux Task Language and compiled to byte-code that is executed by a low-footprint virtual machine.

We implemented a prototype of Flux and assessed the overhead of the service on the mobile devices, which showed a low-footprint pattern. Then we conducted two case-studies, one where a group of volunteers walked over a survey area using smartphones and tablets, to take measurements of the Wifi signal. The second case-study demonstrated the ability of the system to dynamically reconfigure the task pool on the devices, based on their location.

Keywords: Mobile Data Sensing, Mobile Crowd-Sensing, Software Architecture, Domain-Specific Language, Virtual Machine, Android.

Acronyms

DSL Domain Specific Language.

GPS Global Positioning System.

HDOP Horizontal Dilution Of Precision.

MCS Mobile Crowd Sensing.

MDS Mobile Data Sensing.

SONAR Sensor Observation aNd Actuation aRchitecture.

SSID Service Set Identifier.

VM Virtual Machine.

WSN Wireless Sensor Networks.

Contents

Resumo **III**

Abstract **IV**

List of Tables **VIII**

List of Figures **IX**

1 Introduction **1**

1.1 Motivation 2

1.2 Problem Statement 3

1.3 Contributions 4

1.4 Outline 5

2 Related Work **6**

2.1 SONAR 6

2.2 Wireless Sensor Networks 9

2.3 Mobile Crowd Sensing Systems 12

2.4 Summary 14

3 The Flux Framework **16**

3.1 Framework Requirements 16

3.2 Architecture 17
3.3 Data Layer 19
3.4 Processing Layer 21
3.5 Client Layer 22
3.6 Message Flow 22
3.7 Summary 24

4 Implementation **25**

4.1 Programming framework 25
4.2 Communication 26
4.3 Gateway 27
4.4 Broker 29
4.5 Android Service 31
4.6 Client interface and Gateway Manager 34

5 Evaluation **37**

5.1 Resource Consumption 37
5.2 Wifi coverage case-study 39
5.3 Gateway roaming case-study 43
5.4 Summary 48

6 Conclusion **49**

6.1 Discussion 49
6.2 Future Work 50

List of Tables

4.1 Technologies used. 26

4.2 Commands available on the shell-based client. 35

4.3 Commands available on the gateway manager interface. 36

5.1 Resource consumption. 38

5.2 Byte-code size and execution time. 39

5.3 Android device characteristics 42

List of Figures

1.1 Dynamic task reconfiguration 4

2.1 SONAR architecture. 7

3.1 Flux architecture. 18

3.2 Dynamic task reconfiguration 19

3.3 Android Service. 20

3.4 Message flow - new deployment 23

3.5 Message flow - task management 23

3.6 Message flow - new node. 24

4.1 Specification of a geographic region for a deployment. 28

4.2 Android Service graphical user interface. 32

4.3 Web client. 35

5.1 Survey area for the Wifi coverage case-study. 40

5.2 Data plots for collected data 43

5.3 Survey area for the roaming case-study. 44

5.4 Location of the samples. 47

5.5 Average of microphone readings by corridor (absolute reading). 48

Chapter 1

Introduction

The use of sensors to monitor physical or environmental phenomena has been advantageous in several applications. Systems that take advantage from this concept can diverge from a simple gathering of information to gain knowledge about the environment, to systems capable of automated interaction based on values of the sensed data. With this demand for better systems to sense data, the Wireless Sensor Networks (WSN) emerged consisting of embedded devices distributed spatially (called nodes), usually low-cost and low-power. Each of these devices uses a set of multiple hardware components: a radio transceiver, a microcontroller, an energy source, one or multiple sensors and, if needed, some actuators. From the set of nodes, it is common to see at least one that acts as a gateway to the network and is responsible for disseminating the sensing tasks as well as collecting the captured data from all the nodes. These devices are usually programmed using domain-specific languages [1, 2, 3] easing the complexity defining sensing tasks unique for each sensing scenario.

Nowadays the collection of data is becoming more relevant to different areas of application, leading to a change in the needed requirements to better fulfill the demand. On the other hand, the technological development also presents new opportunities and in that sense, there are millions of potential multi-sensor personal devices in our mobile devices.

Devices, such as smartphones or wearables, are becoming more and more advanced both in software as in hardware, including a rich set of embedded sensors, for example, the gyroscope, the accelerometer, the Global Positioning System (GPS), among others. These sensors primary objective is to improve the user experience, but since most mobile devices are programmable and their popularity is increasing, these devices

could have a great potential for data sensing. Thus, appears the paradigm of Mobile Data Sensing (MDS), though it is mainly used for individual sensing in applications areas such as health monitoring or social interaction.

Another sensing paradigm that focuses on human-centric computing is Mobile Crowd Sensing (MCS) where a group of people is tasked to collect and contribute data using their mobile devices resulting in a larger scale sensing. MCS provides a way to overcome several limitations such as installation costs or insufficient spatial coverage, significantly increasing the quantity and the quality of the collected data. However, there are a few constraints to what can be achieved as well as the limitations related to the fact that the devices used are not dedicated for sensing.

1.1 Motivation

Using mobile devices as a tool for sensing purposes can be advantageous. Even more if the data that is going to be gathered is human-centric, e.g. in areas were the human behavior is monitored, in commerce or in social interaction, because mobile devices are evermore present in peoples lives and are rarely switched off. This added to the computing, sensing and communication capabilities of these devices, it makes a promising concept to explore.

However, there are some obstacles that must be taken into consideration. Although the hardware of a typical mobile device such as a smartphone can be several times more powerful than a node used in WSN, there is the problem that these devices are not dedicated for sensing, so the overhead of running such applications must be as low as possible mitigating the impact on the user experience. Another relevant requirement is the infrastructure for managing the sensing tasks. This includes disseminating different tasks to the nodes, collect the captured data and make it available to the person responsible for handling and processing the data.

In a way, some of the major problems can be addressed using knowledge from the development of typical WSN. WSN has come a long way, introducing techniques to better cope with sensing challenges and improve the approach needed to collect large amounts of data. In that sense, we intend to reuse some of the work done in the Sensor Observation aNd Actuation aRchitecture (SONAR) project [4] that tackles some of the common obstacles, and then develop the MCS framework from there.

1.2 Problem Statement

To enable general-purpose sensing while using mobile devices, we intend to develop the Flux framework, by taking advantage of the progress achieved in SONAR.

SONAR is a WSN framework that enables the definition of sensing tasks and the respective dissemination to the devices that will perform the actual sensing, also providing the necessary infrastructure to aggregate and present the collected data to interested users. SONAR has an architecture with three layers: the data layer that represents the components that perform the data collection and the sensing tasks management, the processing layer that is composed by a broker that receives the published data, and the client layer composed of modules that received captured data using a publish-subscribe interface provided by the broker.

Our aim is to adopt the key features that compose the SONAR framework and are compatible to be used as part of an MCS system. More specifically, we want to use the publish/subscribe system that handles the data aggregation and distribution for the clients, and the domain-specific programming language for defining sensing tasks that will be disseminated on the mobile devices.

There are however several challenges to address when adapting a WSN platform for mobile data sensing using personal devices. In this case, all the layers in the SONAR framework need to be updated and improved, with special emphasis to the data layer that needs to be completely rethought so it can handle the differences between the typical nodes used in a WSN framework and the mobile devices.

Furthermore, in WSN the physical location of nodes is static, whereas personal devices move as they are carried by users. We see this as an advantage that enables the possibility of a specific mobile device to participate in different sensing activities. In other words, we pretend a system that is capable of dynamic reconfiguration of the sensing tasks running on the mobile device based on in its own location, as shown in Figure 1.1. We envision regions that require a specific set of sensing tasks and, as the mobile device moves through different geographic regions, it selects automatically which tasks should be running.

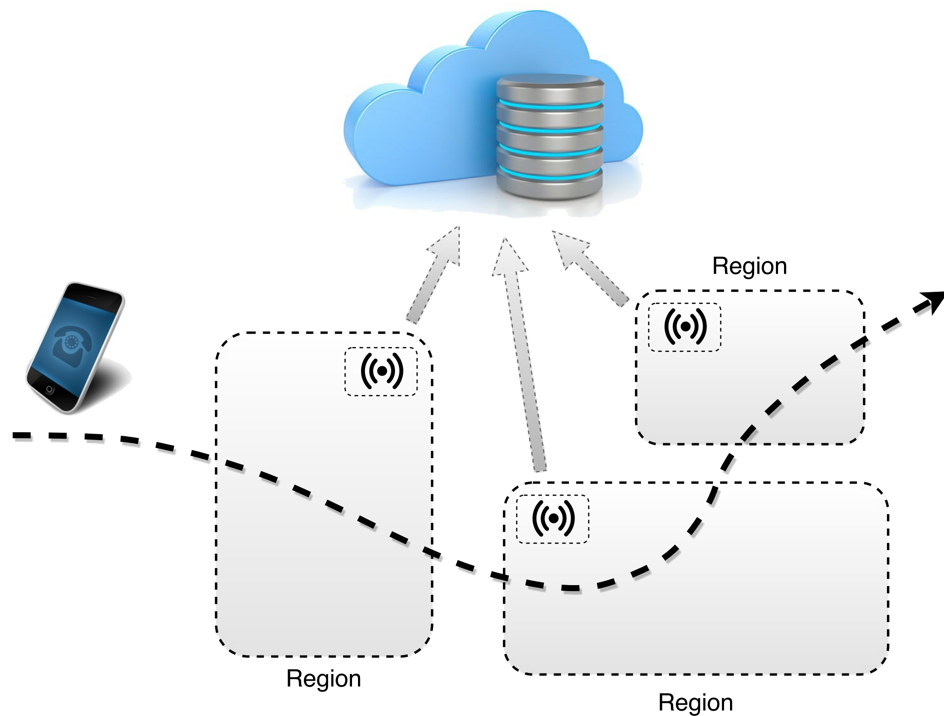


Figure 1.1: Dynamic task reconfiguration

In summary, the overall research questions are:

Is it possible to build a general-purpose platform to acquire data from sensors present on mobile devices? If so, can it be implemented without affecting the user-experience since the hardware is not dedicated to data acquisition? Can it profit from the mobility of devices? How does it compare to other proposals in the state-of-the-art?

1.3 Contributions

With the work described in this dissertation we made the following contributions:

- The implementation of an Android service for running sensing tasks, based on a VM and a domain-specific programming language intended for WSN. Moreover, introducing the additional components for the correct operation of the service: a task scheduler, a sensor/actuator interface, and a connection handler;
- A framework for handling the task dissemination to the mobile devices, depending on the location of the devices. Also, providing the necessary infrastructure for gathering the captured data from the devices and forward it to the clients

that made a subscription. Primarily as a live data stream, but also giving the option for accessing older data;

- A client interface for accessing the data streams from the sensing tasks that can be used with a browser and an internet connection. Provides the visualization of the data in live charts that are updated as the data arrives;

- A validation of the developed system, by performing a resource consumption evaluation of the Android service, to ensure the low-footprint profile. In addition, two case-studies were carried out, one using volunteers that performed a Wifi signal coverage survey. And a second case-study that proved the dynamic task pool reconfiguration based using only the device location.

1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 presents an overview of the SONAR framework followed by an enumeration of the more relevant related work regarding WSN that take advantage of virtualization and systems that employ the paradigm of MCS. Chapter 3 details the architecture of the developed framework. Chapter 4 presents the details for the implementation achieved. Chapter 5 presents an evaluation to the performance overhead of running the Android Service on the device and an overall validation of the framework through two case studies. Finally, on Chapter 6 we present our remarks about the developed system and give some future work.

Chapter 2

Related Work

In this Chapter, we present the most relevant state-of-the-art for this dissertation including a description of the SONAR framework. In Section 2.1 we depict the SONAR framework architecture and the respective task language utilized to define sensing tasks. Section 2.2 lists several WSN frameworks that take advantages in virtual machines. In Section 2.3 are listed some systems that use the concept of Mobile Crowd Sensing.

2.1 SONAR

SONAR is a general purpose WSN framework. It implements virtualization on the nodes, allowing the dissemination of tasks in the form of byte-code, generated from a simple domain-specific programming language. This framework consists of a publish/-subscribe model separated in a three-layer layout as shown in the Figure 2.1: Client layer, Processing layer, and Data layer.

The Data Layer consists of multiple deployments that are characterized by a mesh of nodes and the respective adapter and gateway, where each deployment can be configured to have its own set of tasks. The adapter receives connections from the administrative client interface allowing the management of the deployment and also forwards the messages coming from the gateway to the SONAR P/S Engine. The gateway is equipped with a radio device to enable the communication between the adapter and the mesh of nodes. The nodes are responsible for the data acquisition and controlling the actuators.

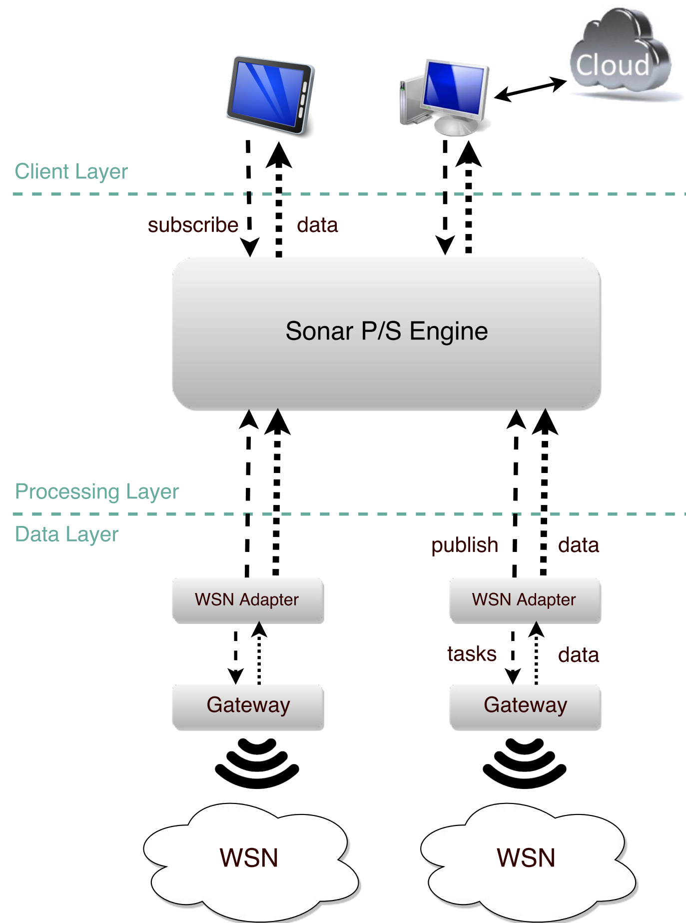


Figure 2.1: SONAR architecture.

The Processing Layer is composed of the SONAR P/S Engine that acts as a message broker. The main purpose is to forward the data streams from the Data Layer to the respective subscribers and to keep information about all the available deployments and associated tasks.

The Client Layer connects to the SONAR P/S Engine and allows the user to list the available deployments and the respective running tasks. The user can also subscribe or unsubscribe the data streams associated with each task.

2.1.1 Domain Specific Language (DSL)

The tasks are written in a domain-specific language designed for periodic sensing tasks. This is a statically-typed language that is compiled to a machine-independent byte-

code, thus abstracting the hardware of the machine where the VM will run. The DSL is quite constrained in order to provide guarantees of safe execution also providing a predictable memory footprint. The period for the task activation is not defined by the task code, instead, it is configured when the task is installed.

The DSL code for a task is structured in five sections. The first two sections define the sensors and actuators that can be used in a task, where each of them is defined by a signature, one or more argument types and the return type. The `init` block declares and initializes task variables that persist in memory across task invocations. After that is indicated the definition for the message that is sent by the task, describing for each field a type, a label, and a description of the units used. The `loop` block that contains the actual instructions, that execute every time the task is activated.

The DSL code supports sensor reading, actuator control, variable assignments, basic arithmetic, conditional branching and data transmission. Some of this is shown in the task displayed in Listing 2.1 that is intended to collect the temperature and humidity values.

Listing 2.1: DSL task for collecting temperature and humidity values.

```
sensors {
  temperature: void -> float,
  humidity: void -> float
}

init {
  float x = 5.0;
  float y = 0.0;
}

[ float @ "temperature:Celsius",
  float @ "humidity:%" ]

loop {
  x = temperature();
  y = humidity();
  radio [x,y];
}
```

A task expressed in the DSL is compiled onto abstract byte-code that will be interpreted by a VM installed on the nodes in charge of the data collection. The byte-code runs on a typical stack-based VM, i.e., each byte-code operation pops operands from

a stack and/or pushes its results onto to the stack. It has four segments: `header`, `data`, `stack`, and `text`. The `header` segment contains the total size of the byte-code and the offset to the text segment. The `data` section contains all the space for the program variables, corresponding initial values, and other program constants. The `stack`, whose size is calculated at compile time, is used for data manipulation. Finally, the `text` section contains the actual instructions to be executed.

Listing 2.2 shows a human readable representation of the byte-code resulting from the compilation of the task in Listing 2.1.

Listing 2.2: Representation of the byte-code generated from the task shown in 2.1.

```
.total      45
.offset     7

.data
.0   5.0
.4   0.0
.8   x
.12  y

.stack
.16  0
.20  0
.24  0

.text
.0   rd  0  0
.3   st  2
.5   rd  1  0
.8   st  3
.10  ld  2
.12  ld  3
.14  rad  2
.16  ret
```

2.2 Wireless Sensor Networks

In this Section, we focus in WSN systems but more specifically on the ones that use virtualization for handling the sensing tasks. Implementing a Virtual Machine (VM) comes with a cost to both memory and processing performance but this is becoming

less of an issue since the hardware is gradually improving his capabilities.

The use of virtualization provides multiples advantages namely portability, and flexibility. In the context of WSN, this can be quite relevant since it can provide, for example, an abstraction to the platform hardware and provide a standard programming interface for different target devices. This enables, for instance, the possibility of increasing the number of nodes without being too restricted by the hardware specifications. Running byte-code in a separate layer of the operating system, on the device, allows to dynamically load the VM program and help deal with the challenges of fault tolerance.

There are already some solutions that implement VM's for WSN's with different approaches and features.

SwissQM [5] is an implementation of a WSN that uses a gateway node and one or more sensor nodes. The gateway serves as an interface to the sensor network where it processes the user queries and replies with data streams. On the nodes is installed a stack-based Virtual Machine on top of TinyOS [6] where it interprets a specific byte-code that is generated on the gateway, based on the user-submitted query for the data acquisition tasks. The queries are written in a generic high-level declarative programming model that supports both SQL and XQuery. To disseminate the programs in the WSN, they are split into several fragment messages and sent as payload over the broadcast layer. Programs are executed at a configured interval. The byte-code instruction set is independent of the sensor platform and it defines a set of instructions that allows stack, arithmetic and control operations, load/store instructions, as well as instructions for sensing, transmission and data aggregation.

Maté [7] is another Virtual Machine designed to work on top of TinyOS. This VM is a stack-based binary code interpreter, that implements two stacks, one for instructions that control the flow of the program and another for all other instructions. The programs are written in TinyScript [7], a simple BASIC like language. The instructions supported can be divided into three groups: instructions for arithmetic operations and activation of sensors/actuators, instructions for memory access and instructions for branch operations. It is also possible to define user instructions but they can only be established when Maté is installed on the nodes. To disseminate a program through the nodes, Maté uses packages denominated as capsules that can accommodate at most 24 instructions, so when needed, a program can be divided in multiple capsules. To forward them, it is only required one instruction that uses the built-in ad-hoc routing

algorithm. During the process of diffusion of the capsules, to determinate if a given node needs to install a new application code, it only needs to verify the version number present in every capsule.

Agilla [8] provides a style of mobile-agent programming, where each agent can proactively migrate their code and state across the network. Unlike the previously mentioned systems, in which the entire network runs the same task or tasks, a given agent (task) can be executed only on some nodes of the sensor network. In other words, each node of the network can execute several agents or even none, if not justified. That is why Agilla addresses nodes by their location. To facilitate the mobile agent interactions, each node keeps two data abstractions: a neighbor list and a tuple space. The tuple space is a shared memory for communications between mobile agents. The agents are defined in a stack-based architecture in which their programming is done through an assembly-like language. On the nodes, Aggila runs on top of TinyOS.

Scylla [9] is a VM that supports inter-device communication, power management, and error recovery. Supports the migration of the running task to other devices while preserving its state. This VM does not interpret the received task byte-code, instead, it compiles the Scylla byte-code into native machine code. To facilitate the on-the-fly compilation, most of the Scylla instructions can be directly mapped onto instructions of modern microprocessors and microcontrollers.

Before receiving a task, the node receives information about the task requirements so the VM can decide if it is possible to compile and run the task. This information can be the amount of memory needed, the requirements of energy to run the task or the sensors required. A task is composed of the application code that contains the Scylla byte-code, the memory image with the data to be loaded into memory prior to the task execution, and the structure of the fault handler.

WSN has been meaningful for sensing work so it is important to review the considerations that they offer on the matter. On a brief comparison of these systems with our framework, they all focus on execution sensing tasks over VM but they differ in some points. SwissQM and Maté were targeted to be deployed on static nodes, so there is no notion of location when running a sensing task inside a deployment. Agilla does not force a sensing task to run on the entire deployment and even can address the location of the nodes but again, assumes that nodes are geographically static. Scylla

also does not provide a framework to handle changes on the location of the sensing nodes.

2.3 Mobile Crowd Sensing Systems

Following the concept of MCS, there are already several systems that try to exploit the use of mobile devices. These systems are integrated into several areas, for example, health, environment, and traffic monitoring, among others. However, there are several aspects to consider in the development of applications that have the objective of collecting data, highlighting the impact that these have on the mobile device in terms of processor utilization and energy consumption.

In terms of MDS, it becomes important to acknowledge a few characteristics about the sensing applications, for example, the user participation. In Participatory Sensing, the user is directly involved and in Opportunistic Sensing, the user does not have to actively participate. Another relevant characteristic is the type of sampling: continuous sensing, when the data is being constantly recorded, or event-triggered when the data is collected after a certain occurrence. There are also a few other challenges such as data validation and how to send the data to aggregate.

SmartRoad [10] is a crowd-sensing road system for mapping traffic regulators, such as traffic lights and stop signs. It aims to avoid expensive road surveys and provide data that can improve both safety and help compile fuel-efficient routes. It resorts to a smartphone based Crowd Sensing system that in a participatory way, collects data from the GPS sensor. This system is intended to minimize the user's intervention as much as possible, and with that, the user only needs to install the application and start it when it is necessary to collect data.

This system uses a client-server framework where the smartphone main function is to acquire data and send it to the server. Then on the server, the data is processed to detect and identify the traffic regulators. Some pre-processing is applied to the raw data to reduce the bandwidth required to send it to the server and thus reducing possible data communication costs. Another feature is that to reduce errors due to poor sensor quality, environmental noise or even improper handling of the phone, the information of multiple vehicles is combined in a selective way which can improve the results. This system also explores a way of motivating users to join in by using the collected data as input for navigation systems and assisted driving. There is also the

possibility of visualization of the detection and identification results via web service interface.

NoizCrowd [11] proposes the use of smartphone sensors to collect noise levels from a region and generate noise models. For the generation of these models it is necessary a great amount of data and to collect a sufficiently large data set this system used a crowdsourcing initiative. This system consists of four components. An application located in the smartphones that are responsible for recording the noise levels using the microphone, also adding the location of the sample through the GPS sensor. A warehouse layer to store all the data received from the application. A module that, using the collected data, generates the noise models. And a layer that allows the visualization and export of the generated data to the user.

The data collection consists of recording the mean noise levels in decibels, as well as measuring peaks, at intervals of only a few seconds. The data is then sent to the system's data storage via a web service. The data storage is done in a database based on arrays with three dimensions: latitude, longitude and time. Because of a large amount of data, these are compressed and stored in a sparse array, that is, only cells that contain values are written to disk.

Medusa [12] is a programming framework for general purpose crowd-sensing. It defines a sensing objective as a task that is provided by a requestor and is carried out by volunteers that act as workers. Medusa has the principle that the workers can receive an incentive to contribute sensor data. Each task is defined by a XML-based domain-specific language, that provides a high-level abstraction for specifying a sequence of stages or steps on the sensing task, for example taking a video or uploading the generated data.

This framework consists of a distributed runtime system separated into two main components: the Medusa Cloud Runtime and the Medusa Runtime on the Smartphone. The Cloud Runtime is responsible for receiving and parse the tasks, keep track of the different generated instances for each task and manage the associated workers. This uses the Amazon Mechanical Turk [13] system as a backend. The Runtime on the Smartphone is in charge of receiving the tasks from the Cloud Runtime and running them in a sandbox environment. It is also responsible for downloading the stage binary's when is necessary, access the sensors and transfer the respective data.

Sensus [14] is a general-purpose system for MCS-based human-subject studies. The aim is to support scheduled and sensor-triggered surveys and integrate the survey response with data from the embedded sensors on the participant mobile device. This MCS system is composed by the application that runs on the mobile devices and a cloud storage. To create a sensing task the researchers use the mobile application. Using the platform Amazon Mechanical Turk [13], the task is disseminated to the study participants as an encrypted JSON file. Each participant then decrypts the sensing task and loads it into the Sensus mobile application. When the task is complete, the collected data is submitted to Amazon Web Services Simple Storage Service [15] for retrieval and analysis by researchers.

Device Analyzer [16] is a mobile application, developed at the University of Cambridge, that collects data from mobile phones usage and transmits it to a central server where the dataset is kept and analyzed for pattern extraction. The scope of the gathered data focused on several areas such as frequency and duration of interactions of the user with the phone, availability of a data communication, energy consumption, among others. The authors mention that several patterns emerge from the data and can be used to implement recommendation systems, e.g., the best phone plan based on phone usage by the user and apps that may be of interest.

2.4 Summary

In this Chapter, we presented the SONAR framework that provided some relevant elements to the development of the Flux Framework. Followed by some WSN's that despite not being intended to use mobile devices as sensing nodes they share some characteristics in design.

In the last section of this chapter we described the most relevant related work in MCS systems, and in comparison to our system there are several differences. The differences that are most relevant: is the capability of reconfiguring sensing tasks without having to redesign and reinstall the application on the device, and the idea of minimizing the need for the user intervention on a sensing session.

In particular, the SmartRoad, the NoizCrowd, and the Device Analyzer are designed for a specific sensing task only, so it is not possible to change the type or the characteristics of the collected data. Medusa and Sensus are general-purpose frameworks and like our system, use domain-specific languages to define the sensing tasks. Medusa

and Sensus are designed with user interaction in mind, in particular, Medusa requires that the user first validates the sensing task that will be performed and the Sensus framework is specific for surveys that a user has to explicitly answer. In summary, these systems do not allow a dynamic reconfiguration of the tasks to be performed while minimizing the user intervention.

Chapter 3

The Flux Framework

In this chapter we present the organization of all components that form the Flux Framework and their relevance to the system. In Section 3.1 we start by enumerating the main requirements for the Flux Framework. In Section 3.2 we give a brief description of the main elements that compose the framework. Sections 3.3, 3.4 and 3.5 give a more detailed description for each of the layers.

3.1 Framework Requirements

The use of mobile devices for sensing purposes adds several concerns that have to be considered in the framework design when comparing to a typical WSN. First, it is not guaranteed that a mobile device always has an active data connection, either because the user is in an area without network access or simply because the user imposes periods without a connection. Similarly, a user can enter or leave a sensing session without notice. There is also the constraint that these devices are not dedicated to sensing. Finally, the definition and distribution of sensing tasks must be done in a straightforward way, also ensuring that the captured data reaches interested clients.

With these concerns in mind, we considered the following requirements for the Flux framework:

- to be able to disseminate sensing tasks to be performed by mobile devices and aggregate the captured data to be forward to interested clients;
- to allow the configuration of tasks to be executed in specific regions, where the

task pool on the mobile devices are automatically updated if they enter such regions;

- to operate without requiring user intervention, unless explicitly requested by the user, e.g., for management or monitoring;
- to be able to work on any type of mobile devices, provided that it is possible to implement the virtual machine for that devices;
- to have a low execution footprint in terms of resource consumption.

The SONAR framework meets some of the defined requirements so it is suitable to harness part of developed work to hasten the Flux development. However, the SONAR framework was developed in the context of WSN, so it is necessary to make several changes in order to comply with the remaining requirements.

The Data Layer needs to be redesigned, from the component that will manage the sensing tasks for each region, to the service that will run on the mobile devices. In particular, we need to migrate the SONAR VM to Android and build the other necessary components, such as a connection handler, a task scheduler, and a sensor/actuator manager since the access to the sensors in the mobile devices is done in a more elaborate way.

On the Processing Layer, we need to add two modules: one to store the captured data from the sensing tasks and the other to manage the assignment of deployments to the mobile devices depending on the location of the device, that is, verifies if the device is in a deployment region.

On the Client Layer, we need to add a client interface that can be accessed using a web browser with an internet connection.

3.2 Architecture

Flux is a system for gathering sensing data using mobile devices. Provides an infrastructure for the distribution of sensing tasks on the devices and also gathers the produced data. It implements a publish/subscribe model where clients can access data streams that are made available by a broker. The data streams are published on the broker by multiple gateways that collect the data generated by the mobile devices.

It follows a three-layer architecture represented in Figure 3.1. The Data Layer contains multiple deployments that are composed of a gateway and several mobile devices. The gateway handles the task injection on the devices and also retrieves the data. Its configuration is done by an administrator, that uses the gateway manager. In the mobile devices is installed the Android Service, that executes the tasks on a VM and manages all the details for participating in a deployment.

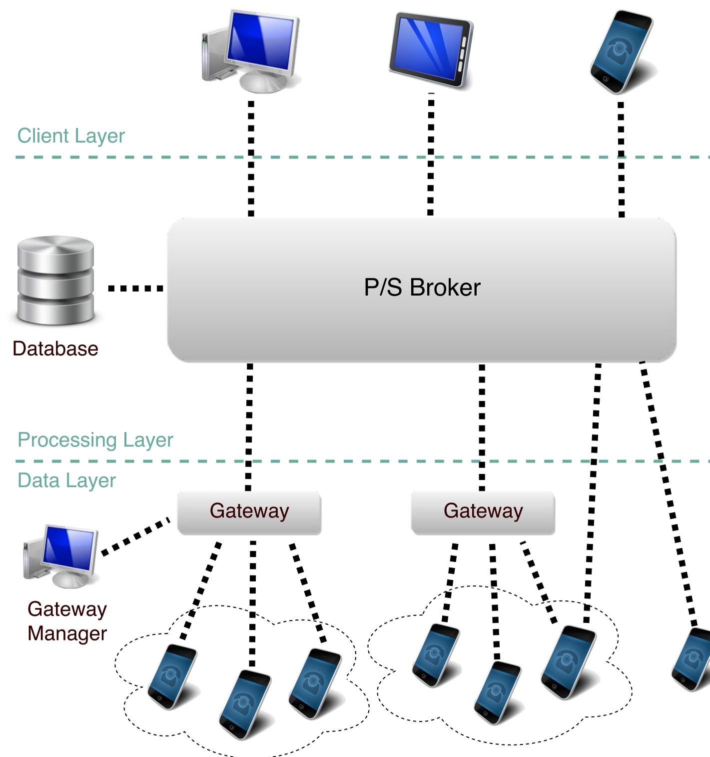


Figure 3.1: Flux architecture.

The Processing Layer receives all the data streams from the Data Layer and forwards the data to the respective clients. This layer has knowledge of all deployments and also manages the gateway attribution to mobile devices that are going to retrieve data. In addition, the broker stores all the received data streams in a database, allowing more flexibility for the clients to fetch data. The Client Layer is composed of different interfaces that grant access to the data streams.

Deployments can be defined to specific regions, by doing this a mobile device can only connect to the respective gateway if it is within the designated area. On the other hand, since the attribution of gateways is done dynamically, if a user crosses such regions, then the tasks performed on the device will be reconfigured to match the ones on each deployment. Figure 3.2 illustrates this feature.

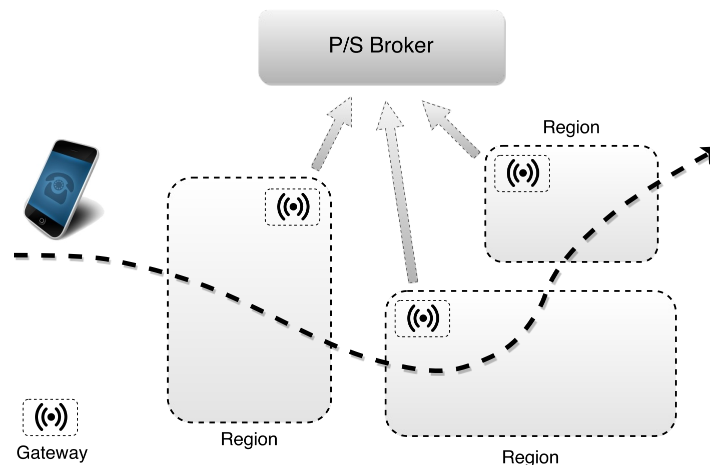


Figure 3.2: Dynamic task reconfiguration

3.3 Data Layer

The Data Layer is formed by all registered deployments on the framework where each deployment can only have one gateway associated with it. When a device is assigned to a deployment, then the respective gateway uploads the tasks to the device and keeps the device updated as long as it is associated with the deployment. The gateway also retrieves the data streams generated by the device and forwards them to the broker. The configuration of the gateways is done through a gateway manager that connects directly to the gateway. The mobile devices act as sensing nodes, executing the tasks provided by the gateway to whom they were assigned. The number of nodes is not fixed because the mobile devices can enter or leave the deployment at any moment, and when they leave, the tasks are removed.

3.3.1 Gateway

The gateway can be viewed as the main component for every deployment. It is responsible for assigning tasks to every node that enters it. A deployment can have multiple sensing tasks that are assigned to every device in the deployment, except if a device does not have all the required sensors and actuators for a specific task. In this case, that specific task is not loaded to that specific mobile device.

The gateway is constantly connected to the broker in order to keep the broker always updated. This information includes the description of the active tasks and the location

restrictions for the deployment if any. This connection also serves for the gateway to send the data generated on the deployment to the broker.

3.3.2 Android Service

The Android Service is installed on the mobile devices. It runs in the background, without the need for the user intervention. The service manages all the process in which a mobile device participates in the sensing. Including handling the reception, scheduling and executing tasks, and the sensor/actuator control. It is depicted on the Figure 3.3.

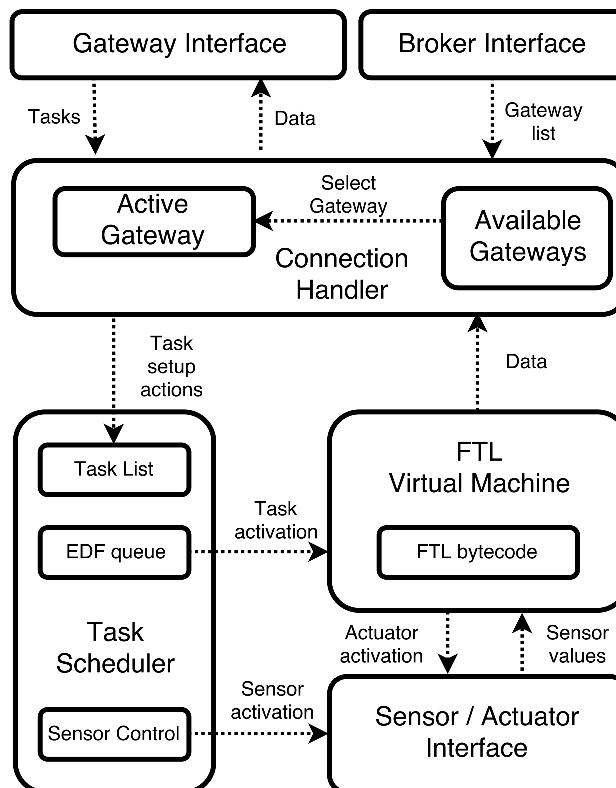


Figure 3.3: Android Service.

The most relevant components that comprise the service are:

- a connection handler that is responsible for contacting the gateway to download the tasks (byte-code and properties) that are assigned to the deployment that the device is currently associated with, and to upload the data captured when

performing the tasks. And to request the available gateways that the device can connect to.

- a Virtual Machine that runs the task byte-code. This is a compact VM, single threaded and run-time safe.
- a scheduler that handles the order for the task execution. It uses a priority queue ordered by the earliest-deadline-first scheduling algorithm. It loads the tasks byte-code into the VM and controls its activation based on the period associated with each task, also managing the task reschedule. The scheduler is also responsible for informing the sensor/actuator interface about the requirements for each task.
- a sensor/actuator interface responsible for abstracting the implementation details about the different sensors and provide a unique interface for all requests from the VM. It also handles the initiation of the sensors, since most of them need to be activated before getting a reading.

Besides the mentioned components, there is also an interface that allows the user of the mobile device to activate and deactivate the service. When the service is enabled, it also provides information about which gateway is currently connected to, and the tasks running on the device.

3.3.3 Gateway Manager

The configuration of a gateway is done using the gateway manager. It allows an administrator to register and unregister a gateway on the broker and change deployment settings, for instance, the geographic location where the tasks are intended to be performed. The task pool on the gateway is also managed using this interface, allowing to add/remove tasks and update the period of execution of a task.

3.4 Processing Layer

The Processing Layer is composed by the broker and acts as a central point between the Data Layer and the Client Layer. It maintains all the information about the available deployments, their registered tasks and the number of devices connected to each gateway. It also keeps a list of clients that are subscribing streams, so when data

arrives from the gateways, the broker knows to which clients send it. A client can request the broker, at any moment, to retrieve data that was collected prior to its connection, from the database connected to the broker.

The Broker also manages the Gateway assignment to the mobile devices. When a node wants to participate on the sensing, first sends a request to the Broker, for which the Broker answers with a list of available Gateways.

3.5 Client Layer

The subscription of data streams is done through the client interface. It provides a set of commands to list the available streams and their description and to subscribe/unsubscribe the desired data streams. Clients can use, a shell-like version, but also a web version that abstracts the use of commands. Both these clients connect to the Flux broker.

3.6 Message Flow

Messages are exchanged between the different components that make the Flux framework. These messages have different proposes, depending on the type of the operation that they represent. In the next figures, we describe the message flow for the more relevant scenarios of message exchange.

Figure 3.4 shows the message flow when a new deployment is registered. The gateway manager connects to the gateway that will handle the new deployment, sending the command with all the parameters necessary for the registration (1). The gateway contacts the broker announcing its configuration (2).

Figure 3.5 shows the message flow that happens in operations related to the management of the task pool of the gateway. When updating or removing a task, the gateway manager will send a command to the gateway (1), then the gateway will inform both the broker (2) and the nodes that are currently connected in the deployment (3). If a client is receiving the stream from the broker, it will also be notified (4) about the changes. When a task is added, the message propagation will be the same with the exception for the notification to the clients, since at that point, that task has no subscribers.

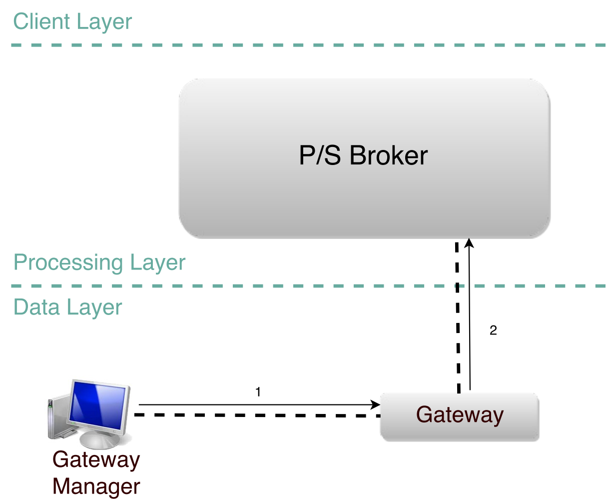


Figure 3.4: Message flow - new deployment

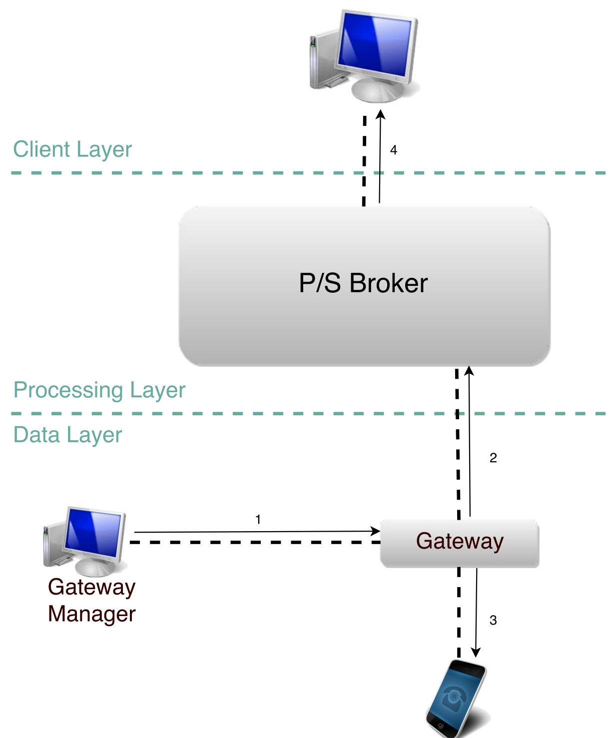


Figure 3.5: Message flow - task management

When a new mobile device wants to participate, first connects to the broker to get a list of available gateways, as shown in figure 3.6, annotation (1). Then the device receives the list of available gateways (2) and connects to one of the suggested gateways (3) and upon a successful connection, it receives the tasks defined for that deployment (4). The broker (5) and the clients (6) are notified about of the new node when they

receive the data that is generated on that node. When a mobile device changes from a gateway, this message exchange starts from the beginning.

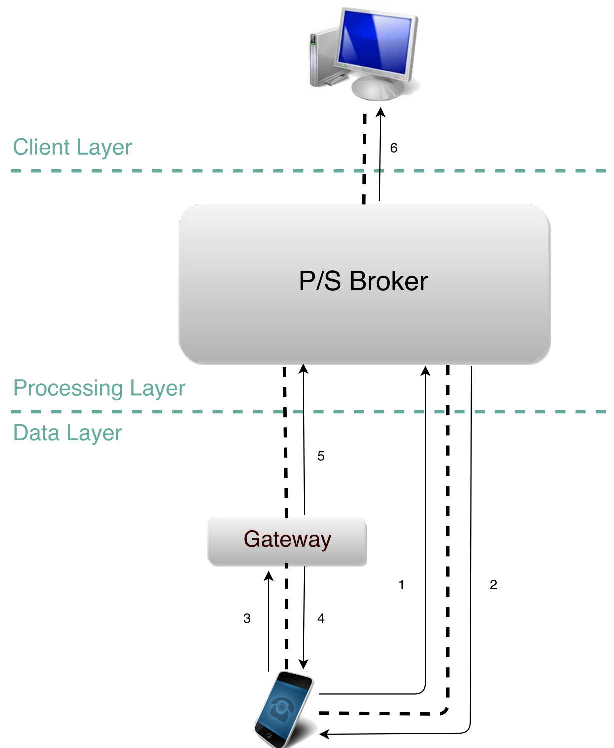


Figure 3.6: Message flow - new node.

3.7 Summary

In this Chapter we described the main aspects that compose the Flux framework by presenting the different components that defined each of the three layer architecture and the framework requirements that were identified. We also illustrated the message flow for the most relevant operations between the Flux components.

Chapter 4

Implementation

In this chapter, we present the most relevant details of the prototype implementation of the Flux architecture. Section 4.1 catalogs the main technologies and programming languages used in the prototype. In section 4.2 we describe how the various communication-related technologies were used in the implementation. The remaining sections depict the major implementation aspects for the distinct elements that form the Flux framework.

4.1 Programming framework

For implementing the Flux prototype, Java version 8 was the main programing language for all components, with the exception of the web client, that was implemented with HTML5 and JavaScript.

Several technologies were employed on the implementation of the prototype, the Table 4.1 summarizes the most relevant technologies used on the system.

The Apache Tomcat is an open source implementation of the Java Servlet, Java WebSocket technologies, Java Expression Language and JavaServer Pages, but for this implementation, we used the first two technologies.

Google Protocol Buffers was developed for serializing structured data in an efficient format [21]. It provides a language-neutral mechanism where we define the desired data structure for the serialization and, using the Protocol Buffers compiler, it generates the source code files for a variety of languages. This source files provide methods to write and read structured data.

Technology	**Used for**
Apache Tomcat [17]	Java Servlet Container
Plain TCP/IP Sockets	Communications Protocol
Google Protocol Buffers [18]	Serialization of structured data
SQLite [19]	Database
ANTLR [20]	Parser generator

Table 4.1: Technologies used.

SQLite is a library that implements a self-contained, transactional SQL database engine. It is designed to be a compact library, that requires low resources and does not need a separate server process.

ANTLR is a parser generator for reading, processing, or translating structured text or binary files. It is used to build languages, tools, and frameworks. In our system, was used to implement the DSL compiler.

4.2 Communication

The gateway and the broker were implemented as an Apache Tomcat web service. More specifically, the communication points on this components use WebSockets. The broker used two instances of WebSockets server endpoint, one for receiving connections from the clients and other for connections from the gateways. The gateways used a WebSocket server endpoint for listening for connections from the gateway manager.

The connection from the Android service with the gateway and the broker is performed using plain TCP/IP sockets. Both the gateway and the broker implemented the ServerSocket side allowing for multiple connections at the same time.

All messages that run through the system are serialized using Google Protocol Buffers version 3. For each point of communication, a different message structure was defined, since the transmitted data is distinct. A portion of the schema file devised is shown on Listing 4.1. The section `message AdminRequest` of the schema file describes the structure of the message that is transmitted from the gateway manager to the gateway. The field `type` describes the operation that the message is intended to, from a list of possibilities defined by the enumeration `Type`. The fields `task` and `deployment` are

sub-messages that serve to encapsulate information regarding the same element, in this case, a task and a deployment. The use (or not) of this fields depends if the defined operation requires it. The section `message Deployment`, as mentioned before, acts as an enclosure for the information that describes a deployment, containing the fields that are necessary to describe a deployment.

Listing 4.1: Fraction of the Protobuff Protocol schema file used.

```
message AdminRequest {
  Type type = 1;
  Task task = 2;
  Deployment deployment = 3;
  enum Type { REGISTER = 0; UNREGISTER = 1; LIST = 2;
    TASK = 3; KILL = 4; PERIOD = 5; RESET = 6; HELP = 7; }
}

message Deployment {
  string id = 1;
  string address = 2;
  int32 port = 3;
  string description = 4;
  repeated string ssid = 5;
  repeated float geoInfo = 6;
  repeated Task task = 7;
}
```

4.3 Gateway

The implementation of the gateway started by defining the data structures to accommodate the sensing tasks, that will be provided to the mobile devices, and the settings for the gateway.

When a gateway is registered on the broker, the address from which the mobile devices can contact the gateway is sent, along with the location restrictions for the deployment, if applied. The broker requires this information for assigning the gateways to the mobile devices.

The gateway keeps a list of tasks that are injected on the devices that are assigned to the gateway. For each task, the gateway saves the byte-code that executes on the VM, the associated period and the definition of the task output that describes the data

stream properties, e.g. the number of fields, their types, and description. In addition, the gateway parses the task byte-code to identify the sensors and actuators required for the task execution.

When a node connects to a gateway, first sends the actual state of its task pool and the available sensors and actuators on the device. With this, the gateway sends to the node, the list of tasks that need to be added or removed from the device, ignoring the tasks that the device cannot perform. A mobile device also connects to the gateway to delivering data that was captured, that is then forwarded to the broker.

To restrict a deployment to a region of influence, it is possible to use a geographic location and/or by the Service Set Identifier (SSID) of the wireless network that the mobile device is currently connected to. For the latest, multiple choices can be defined.

If a geographic location is set for the deployment, the area must be a rectangular form, defined by a geographic point in latitude/longitude, a bearing (counterclockwise from the north), a width and a height. Figure 4.1 depicts the parameters of defining the geographic region.

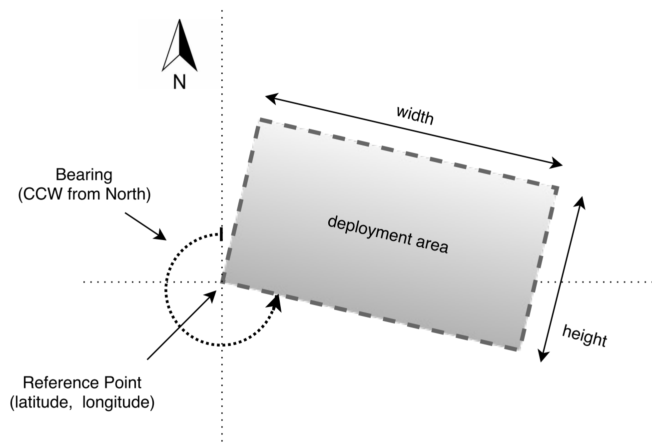


Figure 4.1: Specification of a geographic region for a deployment.

4.4 Broker

The broker tracks all the gateways and their task pools. For this, a permanent connection from the broker to all the gateways is kept, so when an administrator changes a setting on a gateway, the information is quickly updated on the broker.

The connection between the broker and the clients that are subscribing a data stream is also kept permanently open. So the broker can promptly forward the data streams to the clients when they arrive from the gateways.

In order to maintain all the necessary information from the deployments and the clients, the broker essentially contains two data structures:

- a deployment index that lists all the registered deployments. It keeps all the details about the deployment, e.g. id, address, port, but also information about the associated tasks. From the tasks, only description and configurations are stored, the byte-code and sensor/actuator requirements are not sent from the gateways to the broker.
- a table of clients that are subscribing one or more data streams. It associates the tasks to the connection session of the clients.

This information is required by the broker so that it can provide details for the clients to select the desired data streams, and to able to assign mobile devices to the more relevant deployments. Is also used to correctively forward the data streams to the respective subscribers.

The broker acts as a central component in the framework and in order to minimize the initial configuration, the gateways, the mobile devices, and the clients only need to configure the broker public address to join the system.

4.4.1 Gateway assignment to mobile devices

The broker manages the assignment of gateways to the mobile devices. When a mobile device requests a list of the available gateways, the request message also contains the current location of the node and/or the SSID of the Wifi network that the device is currently connected to. Though this information can be omitted if the mobile device opts to not reveal this information.

The broker answers with a list of addresses to the available gateways that match the information given by the mobile device. Put differently, if a gateway was registered with a geographic location and/or a list of Wifi SSID's, then the node has to be in the defined region and to be connected to one of the listed networks. As mentioned before, these restrictions on the gateway are optional and can be defined separately or simultaneously.

The provided list of gateways is ordered by the priority that the mobile device should follow when trying to connect to a gateway. In other words, the mobile device connects to the first given option, but if for some reason the connection is not successful, it tries the next, and so on.

The criteria for ordering the list of addresses is determined by comparing the settings from each of the gateways. Namely, a gateway has more priority if (more relevant first):

1. is restricted in both geographic coordinates and SSID.

2. is only restricted by geographic coordinates.

3. is only restricted by Wifi SSID.

4. has no restrictions about the location.

The choice to prioritize gateways that have more restrictions was made because it is most likely that these gateways will have fewer mobile devices assigned, so if any device meets the requirements to participate, it will be immediately selected.

If more than one gateway matches one criterion, then they are ordered by the number of mobile devices that are registered on each gateway. More specifically, the gateways that have a lower number of devices, come first. This way we can add some level of load-balance and redundancy since multiple gateways can be defined with the same constraints.

The fact that the broker is responsible for delivering the addresses of the gateways to the mobile devices, is also relevant in the sense that, this way, a mobile device only needs to know the address of the broker.

4.4.2 Database

Clients can request data from streams prior to when the subscription was made. This is possible since the broker keeps a record of all the data from the streams that were published by the gateways. When data from a new stream arrives at the broker, a table on the SQLite database is created. Then all the posterior data from that same stream is added to that same table. The fields on the table are defined to match the specific task output, determined by the definitions provided when an administrator injects a task on the gateway.

Whenever an administrator removes a task from a gateway or unregisters the gateway, all the data associated to it is deleted.

4.5 Android Service

The Android service was intended to run without the intervention of the user, but still leaving some level of control. It runs as a background service that in an automatic way, manages the gateway discovery, the task synchronization and execution, and the data commit to the respective gateway. This service was developed to be supported by devices with Android version 4.4 and higher.

The service is composed of multiple modules that handle specific functions. The more relevant are the connection handler, the task scheduler, the sensor/actuator handler and the virtual machine.

4.5.1 Graphical user interface

The service has a user interface for enabling/disabling the service as the user sees fit, as shown in Figure 4.2. It also gives the option for the user to enable/disable the use of GPS for gateway assignment, providing some sort of privacy for users concerned with exposing their specific location to the framework. Another option available is to disable automatic upload of data to the gateway and to cache the generated data from the tasks executing. With this, a user can choose which data connection to use for delivering the data to the gateway. A distinct objective of the interface is to show the user which are the current tasks being executed, their description and also which gateway they are currently assigned to.

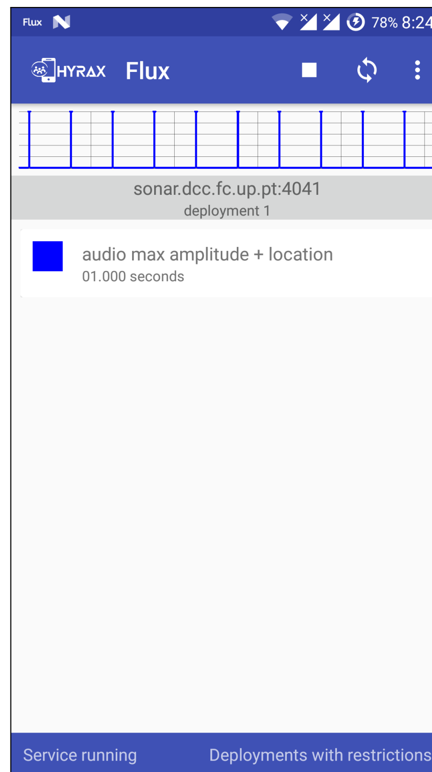


Figure 4.2: Android Service graphical user interface.

4.5.2 Connection handler

The connection handler is responsible for interacting with the broker when getting the list of available gateways and also with a gateway when joining the respective deployment. More specifically, the connection handler implements three different methods. One to connect to the broker, requesting for a list of available gateways, sending also information about the current location of the device (if permitted by the user). A second to connect to the gateway and request the tasks defined for the deployment, where it first sends the available sensors and actuators on the devices. This method also serves to test the connection with the gateway or gateways provided by the broker. The last method is responsible for connecting to the defined gateway for delivering the data collected by the tasks. The data is buffered rather than continuously sending the data. Time and buffer size limits may be set and fine-tuned if desired, so the data transmission to the gateway occurs periodically.

A mobile device can only belong to one deployment at a given time. The connection handler is also responsible for periodically check with the broker if the gateway that is currently being used is the most relevant. It also takes into consideration if the current

deployment has a geographic region and/or Wifi SSID restriction, defining listeners for when the restrictions location are no longer fulfilled. In this case, the connection handler requests a new list of available gateways.

4.5.3 Scheduler

The task scheduler uses a priority queue for storing the tasks on the mobile device. It orders the tasks for execution, based on the earliest-deadline-first scheduling algorithm. The scheduler is responsible for loading the task with the shortest time for execution on the VM (byte-code) also setting that time for the VM activation. Upon the complete execution of the task, the scheduler recalculates the time until the next activation based on the periodicity of the task and reorders the queue. The scheduler is also responsible for keeping the sensor/actuator interface updated about the requirements of the tasks and the time until needed.

4.5.4 Sensor/actuator interface

The sensor/actuator interface is responsible for answering the requests from the VM, whether they are for retrieving a sensor value or enable an actuator. It provides an interface for the platform, where the details from the various Android application programming interfaces are hidden from the VM.

The sensor/actuator interface also balances time a sensor needs to stay active. Active sensors may consume significant battery power, and on the other hand, their repeated initialization/shutdown may cause unnecessary latency. In particular, initialization may imply high latency until valid readings are obtained (for example, the GPS). Using the information provided by the scheduler about the time for next activation and periodicity for each task, it implements an activation/deactivation strategy for the sensors. For a task with a small period (high-frequency), below a certain threshold, the sensors it uses are enabled before the first task activation and henceforth left on. Otherwise, for a task with a larger period (low-frequency), the sensors it uses are turned on and off respectively before and after each task activation. In the latter case, to avoid stale reads when the task is activated again, the module also takes care to schedule the sensor activation for a configurable amount of time before the deadline of the next task activation is reached.

As mentioned, the threshold values for the sensor activation/deactivation depends on

the sensor type, where these values were selected based on the behavior of the sensors on some of the devices that were used in the case-study in Section 5.2. For the GPS sensor was defined that it should be deactivated if the location values are only needed in intervals superior to 2 minutes, and should be reactivated 30 seconds before the execution of the next task that required the information about the device location. For the remaining sensors, was defined a threshold for deactivation of 5 seconds and reactivation of 600 milliseconds before the next time that a value is required.

4.5.5 Virtual Machine

The virtual machine handles the task execution. In a first moment, the VM is loaded with the task byte-code that will be carried out and on the moment defined by the scheduler to initiate the execution, the VM starts performing the task. When a value from a sensor is needed (or initiate an actuator), the VM interacts with the sensor/actuator interface. For sending data, it contacts the connection handler for temporarily storing the data. When the execution is done, the scheduler is notified for rescheduling the task and load the next on the VM.

4.6 Client interface and Gateway Manager

To access the data streams provided by the broker, it is possible to use one of the two available types of client interfaces: a web client that can be accessed by using a web browser and a shell-based client.

The web client grants a more flexible access to the available data streams since it only needs a web browser. All the expected operations are available: list the deployments and the associated tasks, and subscribe/unsubscribe the data streams. The subscribed streams are presented in the form of charts for an immediate visualization, where it was utilized the Google Charts [22] for its implementation. Figure 4.3 shows a web client that subscribed two tasks, but is only interested in viewing one value of each task.

The shell-based client offers the same functionalities as the web client with the addition that it can provide access to the data stored in the database in the broker. A more comprehensive list of the available commands is shown in Table 4.2.

The gateway manager is very similar to the shell-based client in terms of implemen-

Figure 4.3: Web client.

Command	Description
`list`	Lists all the available deployments and tasks.
`sub TASK_ID DEP_ID`	Subscribes the task TASK_ID from deployment DEP_ID.
`unsub TASK_ID DEP_ID`	Unsubscribes the task TASK_ID from deployment DEP_ID.
`query TASK_ID DEP_ID T1 T2`	Requests data from the database from time T1 to time T2.
`help`	Lists the available commands.

Table 4.2: Commands available on the shell-based client.

tation. It provides a list of commands that enable the administrator to specify all the settings of the gateway. A list of the available commands is shown in Table 4.3. From the commands mentioned, the command `register` needs a more detailed explanation. When registering a deployment, the only mandatory arguments are the address and the port that define the link for the gateway to receive the connections from the

mobile devices. This is the address that is provided by the broker when a mobile device requests a list of gateways to connect to. After the administrator enters these details, is prompted to enter the restrictions for where the deployment will be available, namely the geographic location and the list of SSID's, which are optional.

Command	**Description**
`list`	Lists all the registered tasks on the gateway.
`register ADDRESS PORT DESC`	Registers the gateway, accessible from the address ADDRESS and port PORT on the broker.
`unregister`	Unregisters the deployment.
`task PERIOD PATH DESC`	Adds the task in PATH (local), with the period PERIOD and the description DESC.
`period TASK_ID PERIOD`	Changes the period of task TASK_ID to PERIOD.
`kill TASK_ID`	Removes task TASK_ID from the gateway.
`reset`	Removes all tasks on the gateway.
`help`	Lists the available commands.

Table 4.3: Commands available on the gateway manager interface.

When a new task is injected on the gateway, it must be already compiled. So the administrator needs to compile the DSL task, using the DSL compiler.

For both clients and gateway manager, the user must provide the address of the broker or gateway, respectively, since the framework is meant to be easily accessible from anywhere that has access to the Internet.

Chapter 5

Evaluation

This chapter presents an evaluation of the Flux framework. We begin with a benchmark analysis of the Android service in terms of resource consumption in Section 5.1. We then present results for a case-study experiment, where real users carried mobile devices across a certain area to measure Wifi signal coverage in Section 5.2. Finally, we report on a small-scale experiment where we tested the ability of mobile devices to roam between different Flux gateways.

5.1 Resource Consumption

We conducted an evaluation of the Android service in terms of resource consumption. In terms of physical resource utilization, we measured the CPU utilization, the RAM consumed and the amount of data transferred from the mobile device to the gateway, in five different configurations for the task pool. Additionally, for the involved tasks and configurations, we measured the byte-code size of tasks and the time of execution per task activation.

5.1.1 Setup

The mobile device used was a Google Nexus tablet running Android 6.0 with 2 GB of RAM and a dual-core 2.3 GHz CPU, plus a gateway and broker installed on a 4-core machine with 12 GB of RAM that was connected to the same network as the mobile device. This was done to mitigate exterior interference on the communication between the device and the gateway, as we wished to evaluate the performance of the service

in isolation.

The gateway task pool was setup using five distinct configurations. The first configuration had no tasks running, with the purpose of measuring the footprint of the service when idle. The other four configurations resulted from successively increasing the number of running tasks by one, and doubling the frequency of each new task by a factor of two. The four tasks were: the task used on the Wifi coverage case-study in Section 5.2 running at 1 Hz, an atmospheric pressure sensing task at 2 Hz, a gyroscope sensing task at 4 Hz, and an accelerometer sensing task at 8 Hz.

For each configuration, we then conducted 5 monitoring sessions of a 2-minute run of the service using the Android Debug Shell (adb). In terms of resource consumption, we sampled the CPU utilization and RAM usage in 1-second intervals, plus the total of the TCP/IP data transmitted by the service in each 2-minute interval.

5.1.2 Results

The results for the resource consumption (with the corresponding 95% confidence intervals) are shown in Table 5.1, in terms of average CPU and RAM usage during the interval, as well as the average network bandwidth used to send the sensed data.

Tasks	**CPU (%)**	**RAM (KB)**	**Net. (bytes/s)**
None ($\emptyset$)	0.17 ± 0.04	9731 ± 2.8	6.2 ± 1.1
WS	0.25 ± 0.06	9851 ± 3.3	86.3 ± 15.8
WS + AP	0.32 ± 0.06	9864 ± 3.3	139.0 ± 25.0
WS + AP + GS	0.58 ± 0.08	9877 ± 3.8	288.9 ± 52.4
WS + AP + GS + AS	1.39 ± 0.10	9915 ± 5.6	576.6 ± 104.0

WS: Wifi survey (1Hz); AP: atmospheric pressure sensing (2 Hz);
GS: gyroscope sensing (4Hz); AS: accelerometer sensing (8 Hz)

Table 5.1: Resource consumption.

Overall, we can observe that the service has a very low footprint for all the measures we considered. On average, CPU usage is below 2% in all configurations, the RAM used is under 10 MB, and the consumed network bandwidth is less than 1 KB/s. Moreover, the implementation scales well as the number of tasks increase: the CPU and RAM overhead of adding one more task at double the frequency is almost negligible,

whereas the consumed network bandwidth increases naturally owing up to the need of transmitting more sensed data.

In addition to resource consumption, we also measured the performance of byte-code execution within the virtual machine. This was done in terms of the average execution time per activation for each of the four benchmarking tasks. This was done for the last evaluation configuration, the one with all tasks enabled. The results (with the corresponding 95% confidence intervals) are shown in Table 5.2, that lists the byte-code size and average execution time in milliseconds per each of tasks.

Task	**Size (bytes)**	**Exec. time (ms)**
Wifi survey (WS)	137	4.55 ± 0.20
Atmospheric pressure sensing (AP)	26	0.23 ± 0.01
Gyroscope sensing (GS)	74	0.44 ± 0.02
Accelerometer sensing (AS)	74	0.42 ± 0.01

Table 5.2: Byte-code size and execution time.

Again, a low-footprint pattern is observed. The Wifi survey task, with larger code size, is the most time-consuming but still takes less than 5 milliseconds on average to run. All other tasks run in less than 0.5 milliseconds, on average.

5.2 Wifi coverage case-study

We conducted a controlled real-world experience where Wifi service quality was surveyed over a certain area. Volunteer users carried Android devices and walked through prescribed paths along the survey area, while the Android service executed an DSL task to collect GPS-referenced Wifi signal data and streamed that data to a Flux gateway.

The survey area, depicted in Figure 5.1, has a dimension of roughly 100×150 meters, and comprises a department building in our university (A in the figure) plus walkways in a garden north of the same building. The figure also depicts an outline of the paths followed by volunteer users carrying mobile devices, covering corridors of the first floor within the department building plus walkways outside.

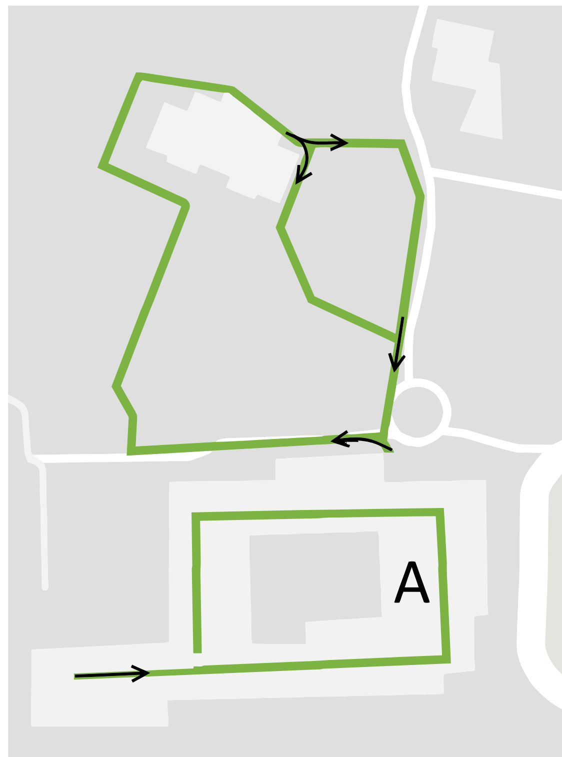


Figure 5.1: Survey area for the Wifi coverage case-study.

The Wifi network subject to monitoring is the eduroam installment at our university, the most commonly used campus network by students and staff. Listing 5.1 shows the task used for this case-study. The task collects the following data: the Wifi signal strength for the current connection, the number of Wifi networks, and GPS location data. The sampling period was set to 4 seconds.

Listing 5.1: DSL task for Geo-referenced Wifi survey.

```
sensors {
  (...)
  WIFI_SIGNAL_LEVEL: void -> int,
  NUMBER_WIFI_NETWORKS: void -> int,
  (...)
  LOCATION: int -> float
}

init {
  int n_wifi = 0;
  int wifi_level = 0;
  float lat = 0.0;
  float long = 0.0;
  float alt = 0.0;
  float acc = 0.0;
}

[ int @ "Number of wifi networks: Number",
  int @ "Wifi signal strength: dBm",
  float @ "Geographic location - latitude: degrees",
  float @ "Geographic location - longitude: degrees",
  float @ "Geographic location - altitude: meters",
  float @ "Geographic location - accuracy: meters"]

loop {
  n_wifi = NUMBER_WIFI_NETWORKS();
  wifi_level = WIFI_SIGNAL_LEVEL();
  lat = LOCATION(0);
  long = LOCATION(1);
  alt = LOCATION(2);
  acc = LOCATION(3);
  radio[n_wifi, wifi_level, lat, long, alt, acc];
}
```

5.2.1 Setup

For the experiment, we used a CentOS Linux virtual machine (CentOS VM) with 2 cores and 1837 MB of RAM, hosted on an OpenStack cloud infrastructure. An Apache Tomcat application server instance runs on the VM, hosting a Flux gateway and a Flux P/S broker. The CentOS VM is accessible over the Internet, allowing devices running the Android service to install tasks (and relay data) from (to) the gateway,

and external clients to access the P/S broker.

For measurements we used a total of 23 devices, divided into two groups: 9 Google Nexus tablets running Android 6.0 that we provided the volunteers for use, plus 12 personal smartphones owned by the volunteer themselves from various vendors and running assorted Android versions, predominantly Android 6.0 (the 9 Google tablets + 9 smartphones), but also 7.0, 5.1, and 4.4 (one device per each version). Table 5.3 summarizes the basic characteristics of these devices.

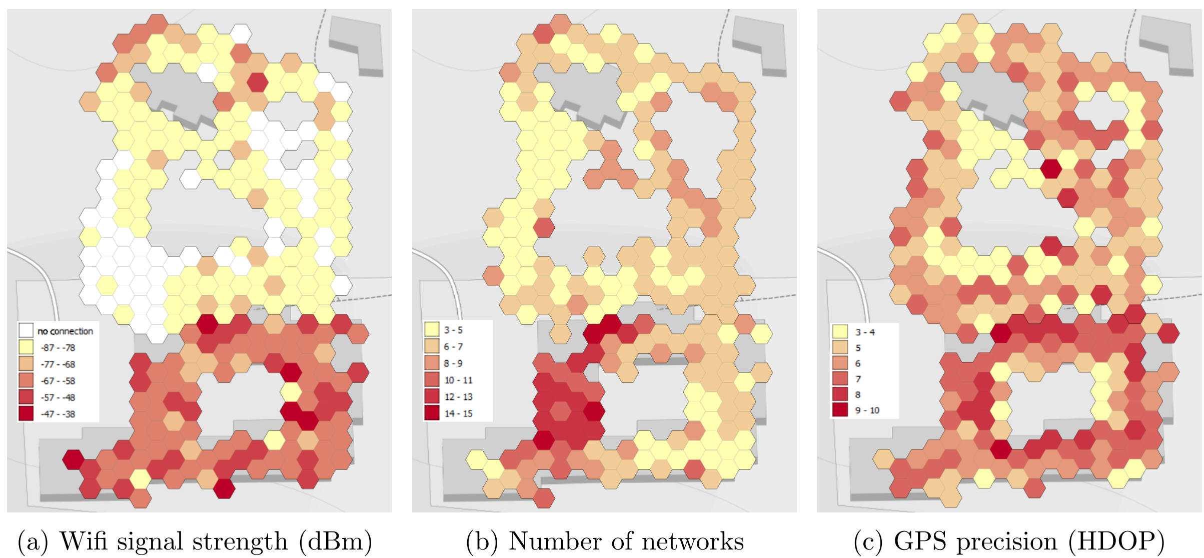
Type	**Version**	**Vendor**
Tablet	6.0	Google (9)
Smartphone	7.0	Samsung (1)
	6.0	Asus (1), Huawei (1), Lenovo (1), LG (1), OnePlus (2), Vodafone (1), Wiko (2)
	5.1	Xiaomi (1)
	4.4	Alcatel (1)

Table 5.3: Android device characteristics

The Android service was installed in each of the devices, followed by an automatic download and installation of the DSL task for the survey by the service itself, as soon as it got a connection to the gateway.

5.2.2 Results

After setup, the volunteers conducted 33 trips along the prescribed survey paths, resulting in the collection of 2726 data sample measurements, 1193 inside the department building and 1533 outside. For data analysis, we filtered out measurements for which the GPS Horizontal Dilution Of Precision (HDOP) exceed 10 meters, reducing our data set to 710 samples (59% of the original) inside the building and to 1212 samples (79% of the original) outside. Figure 5.2 depicts the filtered data set as geo-referenced "heat maps", in terms of the eduroam Wifi signal strength (5.2a), the number of detected Wifi networks (5.2b), and the GPS HDOP (5.2c). In the plots, rendered using QGis [23], the colors depict the average measure for data points within each hexagon that forms the heat map.



(a) Wifi signal strength (dBm) (b) Number of networks (c) GPS precision (HDOP)

Figure 5.2: Data plots for collected data

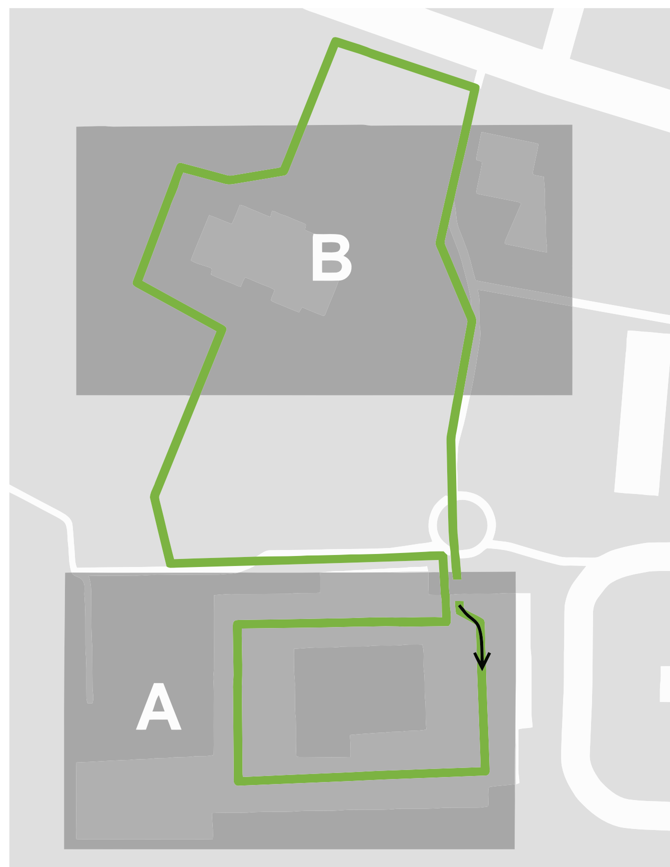
From the plots, we can make a few direct observations. Regarding eduroam's Wifi signal strength, clearly it is significantly weaker in the outside area. An immediate decrease in Wifi signal is observable just a few meters outside the building, and the signal only tended to go up as users move north and get near the two other university buildings. In contrast, the quality of geo-referencing is less reliable inside the building (as would be expectable), given that HDOP measures are clearly better (lower) outside (as also highlighted by the HDOP threshold filtering discussed above).

5.3 Gateway roaming case-study

The second case-study concerns the roaming of devices between gateways.

We deployed two Flux gateways, each responsible for a different geographical region. As in the Wifi survey case-study, the overall area inside and surrounding the department building in our university was considered, but split in two, each under the control of a different gateway.

More specifically, we defined one deployment inside the building (A in the Figure 5.3), where the nodes collected data about the maximum amplitude of recorded sound in samples of one second, by executing the task defined in Listing 5.2. The second deployment was defined in a portion of the garden (B in the Figure 5.3), where we measured the atmospheric pressure by resorting to the task shown in Listing 5.3. All the obtained values were associated with a GPS location information as in the Wifi survey. The audio sampling task ran with a period of 1 second, and the atmospheric pressure sensing task ran with a period of 2 seconds.



Area A: deployment collecting audio amplitude data;
Area B: deployment collecting atmospheric pressure data.

Figure 5.3: Survey area for the roaming case-study.

Listing 5.2: DSL task for Geo-referenced audio amplitude survey.

```
sensors {
  (...)
  AUDIO_AMPLITUDE: void -> int,
  LOCATION: int -> float
}

init {
  int audio_amplitude = 0;
  float lat = 0.0;
  float long = 0.0;
  float acc = 0.0;
}

[ int @ "AUDIO_AMPLITUDE: audio max amplitude",
  float @ "LOCATION latitude: degrees",
  float @ "LOCATION longitude: degrees",
  float @ "LOCATION accuracy: meters"]

loop {
  audio_amplitude = AUDIO_AMPLITUDE();
  lat = LOCATION(0);
  long = LOCATION(1);
  acc = LOCATION(3);
  radio[audio_amplitude, lat, long, acc];
}
```

Listing 5.3: DSL task for Geo-referenced atmospheric pressure survey.

```
sensors {
  (...)
  TYPE_PRESSURE: void -> float,
  LOCATION: int -> float
}

init {
  int pressure = 0;
  float lat = 0.0;
  float long = 0.0;
  float acc = 0.0;
}

[ float @ "Atmospheric pressure: hPa (millibar)"",
  float @ "LOCATION latitude: degrees",
  float @ "LOCATION longitude: degrees",
  float @ "LOCATION accuracy: meters"]

loop {
  pressure = TYPE_PRESSURE();
  lat = LOCATION(0);
  long = LOCATION(1);
  acc = LOCATION(3);
  radio[pressure, lat, long, acc];
}
```

When a mobile device was outside of both the regions defined on the deployments, the Android service entered in an idle mode, since there were no more gateways registered that included the current area.

During the experiment, the path followed by mobile devices starts on the first floor, inside the department building (A in the figure), and goes through the corridors on that floor. It then continues to the outside, following walkways, reaching the outer limit of the campus area. The path ends with a way back to the department building, by a walkway on the opposite side.

5.3.1 Setup

The devices used were two Google Nexus tablets running Android 6.0. They used a 3G data connection to interact with the broker and the gateways in order to prevent data connection loss. The objective of this experiment is to assess the switching capabilities between gateways, where data connection problems could alter the results. So we exclude the use of a Wifi connection since it has a limited coverage on the exterior. The installation of the Android service was performed on the mobile devices at the beginning of the experiment.

The broker and the gateways were hosted on the same CentOS Linux virtual machine mentioned in the Wifi survey, running on the same hardware. A Client was defined to subscribe the data from both the deployments for subsequent analysis.

5.3.2 Results

From the experiment, we obtain a total of 2285 samples for both tasks: 1843 for the deployment on region A, 442 for the deployment on region B. As in the previous case study, we exclude samples that were gathered with a HDOP that exceeded 10 meters, resulting in 406 samples for the deployment on region A (22% of the original) and 259 for the deployment on region B (59% of the original). This data resulted from a total of ten trips, five with each device.

The intention was to evaluate the correct execution of the tasks in function of the

location of the device. We compile the data on the Figure 5.4 based on the location of each sample. Blue markers represent the samples from region A, and red markers represent the samples from region B.

Figure 5.4: Location of the samples.

From the obtained results, we can conclude that the Android service was able to detect the regions under the control of the two gateways. Two details are worth remarking, though. First, on the results from region A, we can notice a few markers on the boundary that resulted from walking on the outside but close to the building. From this, it is obvious that when defining the geographic regions for the deployments, it is important to acknowledge that the readings from the GPS sensor on the devices are not always reliable. Second, regarding the results from region B, we can notice that the Android Service can have a small delay when detecting the limit of the boundaries. Once again, we need to define the regions in a way that can tolerate some degree of imprecision.

Regarding the values of the collected data, for the deployment on region B, we obtain an average of 1016.9 millibars for the atmospheric pressure. For a region so small the variation of the actual value for atmospheric pressure should prove negligible, so we

only present the average value for all the samples recorded in that region.

For the deployment on region A, we calculated the average value from all the samples selected by corridor, as shown in Figure 5.5. The values retrieved from the task refers to the maximum absolute amplitude measured on recordings of one second (defined by the task period), where they represent the absolute reading from the microphone sensor (range from 0 to 32767, since it records with a resolution of 16 bit). We chose to give the bare reading from the sensor because, given that the microphone is not calibrated in most mobile devices.

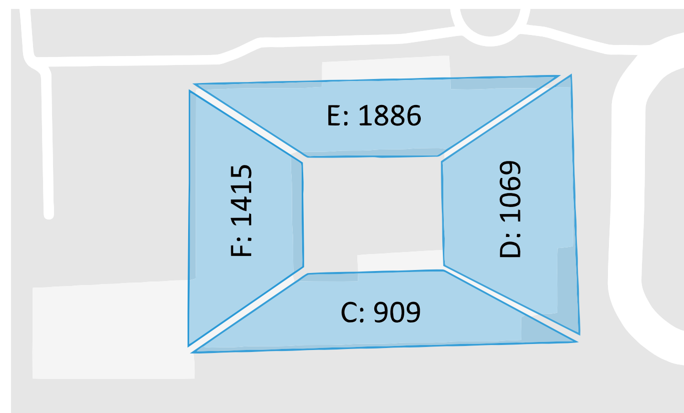


Figure 5.5: Average of microphone readings by corridor (absolute reading).

Considering the obtained values, the difference between the four corridors on the first floor of the department was what we were expecting. The corridors C and D were the less noisy because these are the sections of the building with the less movement of people. Corridor F presented a little higher value since, at the time of the experiment, there were classes in progress. Corridor E was the noisiest since, in this section of the department, there is always a constant flow of people.

5.4 Summary

In this chapter we presented a benchmark evaluation of the Android Flux service, and two case-study scenarios for the entire Flux framework. The benchmark evaluation demonstrates the feasibility of data sensing with very low-footprint on the mobile devices. The case-studies demonstrated the applicability of Flux in a real-world setting. In particular, the Wifi survey illustrates the use of the platform in a relevant scale and using heterogeneous Android devices. As for the gateway roaming scenario, it illustrates how region-based data sensing can be accomplished using Flux.

Chapter 6

Conclusion

In this chapter we present a discussion about the dissertation, also proving some suggestions for future work.

6.1 Discussion

In this dissertation we presented the Flux framework for streaming sensor data from dynamically reprogrammable tasks, injected in mobile devices. We described its architecture and its prototype implementation, demonstrating the dynamic injection of tasks in mobile devices and the acquisition of the corresponding data streams.Flux tasks are programmed in the DSL domain-specific programming language, boasting enough expressiveness for basic sensing tasks, compiled to a compact byte-code that is executed by a low-footprint virtual machine on Android.

The policy used to inject/kill tasks is related to the way region boundaries are detected. In this work, we took the simple view that tasks are injected as a device enters a region, and killed as they leave it. We also took into consideration the initialization and shutdown of the sensors on the device, balancing the time a sensor needs to stay active, so the resource consumption overhead is reduced while minimizing the latency of doing so.

The prototype of Flux has been evaluated in terms of the overhead on mobile devices, demonstrating that low-footprint is a defining trait of Flux. Furthermore, we conducted two case-study experiments, illustrating the capabilities of the entire platform and potential applicability to real-world scenarios.

6.2 Future Work

For future work, we are considering a few key directions. First, we consider the challenge of extending the DSL for increased expressiveness in data processing, for example, adding constructs with support of iteration or array types. Moreover, DSL has no communication constructs that allow neighboring nodes to exchange data for aggregation or pre-processing purposes. In particular, we are interested in mobile edge-cloud environments, where groups of nearby devices form a network to work collaboratively.[24]

Besides extending the expressiveness of DSL, we are also considering several improvements on the framework, for instance, a mobile device being able to participate in multiple deployments simultaneously. Given the processing capabilities of the current mobile devices, this would not prove difficult. Regions could also be defined more broadly, for instance, set boundaries using conditions on attributes of the sensed data.

Another relevant matter is the user privacy. Most sensing tasks are intended to keep the participant identity anonymous, but by the nature of the captured data, it can prove to be a challenge. For example, collecting data using the GPS sensor or the microphone can give clues about the user identity, becoming even more relevant if the user participates in multiple sensing tasks where data can be combined to give stronger indications. On the same topic, the security for the data storage and transmission channels must be considered.

Bibliography

[1] C.-L. Fok, G.-C. Roman, and C. Lu, "Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications," in *Proc. of the 24th International Conference on Distributed Computing Systems*, ser. ICDCS'05. IEEE, 2005, pp. 653–662.

[2] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC Language: A Holistic Approach to Network Embedded Systems," in *Proc. of the 2003 ACM Conference on Programming Language Design and Implementation*, ser. PLDI'03. ACM, 2003, pp. 1–11.

[3] R. Newton and M. Welsh, "Region Streams: Functional Macroprogramming for Sensor Networks," in *Proc. of the 1st International Workshop on Data Management for Sensor Networks*, ser. DMSN'04. ACM, 2004, pp. 78–87.

[4] G. Ferro, R. Silva, and L. Lopes, "Towards out-of-the-box programming of wireless sensor-actuator networks," in *Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on.* IEEE, 2015, pp. 110–119.

[5] R. Müller, G. Alonso, and D. Kossmann, "SwissQM: Next Generation Data Processing in Sensor Networks," in *CIDR 2007, Online Proceedings.* CIDR, 2007, pp. 1–9.

[6] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, *TinyOS: An Operating System for Sensor Networks.* Springer Berlin Heidelberg, 2005, pp. 115–148.

[7] P. Levis and D. Culler, "Mate: A Tiny Virtual Machine for Sensor Networks," in *Proceedings of ASPLOS X.* ACM, 2002, pp. 85–95.

[8] C.-L. Fok, G.-C. Roman, and C. Lu, "Agilla: A mobile agent middleware for self-adaptive wireless sensor networks," *ACM Trans. Auton. Adapt. Syst.*, pp. 16:1–16:26, 2009.

[9] P. Stanley-Marbell and L. Iftode, "Scylla: a smart virtual machine for mobile embedded systems," in *Proceedings Third IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, 2000, pp. 41–50.

[10] S. Hu, L. Su, H. Liu, H. Wang, and T. F. Abdelzaher, "Smartroad: Smartphone-based crowd sensing for traffic regulator detection and identification," *ACM Trans. Sen. Netw.*, pp. 55:1–55:27, 2015.

[11] M. Wisniewski, G. Demartini, A. Malatras, and P. Cudré-Mauroux, "Noizcrowd: A crowd-based data gathering and management system for noise level data," in *Proceedings of the 10th International Conference on Mobile Web Information Systems*. Springer-Verlag New York, Inc., 2013, pp. 172–186.

[12] M.-R. Ra, B. Liu, T. F. La Porta, and R. Govindan, "Medusa: A programming framework for crowd-sensing applications," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. ACM, 2012, pp. 337–350.

[13] "Amazon mechanical turk," https://www.mturk.com, Amazon, 2017, Accessed in 3 January, 2017.

[14] H. Xiong, Y. Huang, L. E. Barnes, and M. S. Gerber, "Sensus: A cross-platform, general-purpose system for mobile crowdsensing in human-subject studies," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2016, pp. 415–426.

[15] "Amazon Web Services Simple Storage Service (S3)," https://aws.amazon.com/pt/s3/, Amazon, 2017, Accessed in 3 January, 2017.

[16] D. T. Wagner, A. Rice, and A. R. Beresford, "Device Analyzer: Understanding smartphone usage," in *Proc. of the 10th International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, ser. MobiQuitous'13. Springer, 2013, pp. 195–208.

[17] "Apache Tomcat," http://tomcat.apache.org/, Apache Software Foundation, Accessed in 2 September, 2017.

[18] "Google Protocol Buffers," https://developers.google.com/protocol-buffers/, Google, Accessed in 2 September, 2017.

[19] "SQLite," https://www.sqlite.org/, Accessed in 2 September, 2017.

[20] T. Parr, "ANTLR (ANother Tool for Language Recognition)," http://www.antlr.org/, Accessed in 2 September, 2017.

[21] K. Maeda, "Performance evaluation of object serialization libraries in xml, json and binary formats." in *DICTAP*. IEEE, 2012, pp. 177–182.

[22] Google, "Google charts," https://developers.google.com/chart/, Accessed in 2 September, 2017.

[23] "QGis," http://www.qgis.org/, Accessed in 2 September, 2017.

[24] P. M. P. Silva, J. Rodrigues, J. Silva, R. Martins, L. Lopes, and F. Silva, "Using Edge-Clouds to Reduce Load on Traditional WiFi Infrastructure and Improve Quality of Experience," in *Proc. ICFEC*. IEEE Computer Society, 2017, pp. 61–67.