Streaming Sensor Data from Dynamically Reprogrammable Tasks Running on Mobile Devices

Nuno Silva, Eduardo R. B. Marques, Luís M. B. Lopes
Faculty of Science, University of Porto & CRACS/INESC-TEC
{up200700262,ebmarques,lmlopes}@fc.up.pt

ABSTRACT

We describe Flux, a platform for dynamically reconfigurable data sensing using mobile devices, like smartphones or tablets. Periodic sensing tasks are programmed using the Flux Task Language and compiled onto abstract byte-code that is executed by a low-footprint virtual machine, guaranteeing by construction important runtime safety properties. For task dissemination, a Flux gateway performs on-the-fly injection of tasks on devices present in a geographical region, and sensing data is streamed back to the gateway that forwards it to a publish/subscribe broker. Live or archived streams are in turn fed by the broker to data processing clients. We implemented a prototype of Flux and used it to conduct a case-study experiment where the intensity of Wifi signal in our department is measured over a certain survey area, using smartphones and tablets carried by volunteers as they walked over the survey area.

CCS CONCEPTS

• **Information systems** → **Sensor networks**; • **Networks** → **Mobile networks**; • **Computer systems organization** → **Sensor networks**; • **Software and its engineering** → **Virtual machines**; **Domain specific languages**;

KEYWORDS

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

ACM Reference Format:
Nuno Silva, Eduardo R. B. Marques, Luís M. B. Lopes. 2017. Streaming Sensor Data from Dynamically Reprogrammable Tasks Running on Mobile Devices. In *Proceedings of 4th ACM Conference on Systems for Energy-Efficient Built Environments (BuildSys'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3137133.3137139

1 INTRODUCTION

The use of sensors to monitor physical or environmental phenomena has manifold applications. Traditionally, such tasks were performed using Wireless Sensor Networks (WSN), networks of low cost, low power, devices typically composed of a radio transceiver, a microcontroller, a power source, and multiple specialized sensors [9]. The nodes communicate using energy efficient protocols like ZigBee, and report sensor readings to one or more network gateways, also known as base-stations, and are programmed using domain-specific languages [14, 16, 22].

Today, however, we have more that 1 billion potential multi-sensor personal devices in our smartphones or wearables. Smartphones, for example, have become ubiquitous and typically feature powerful multi-core processors, several gigabytes of storage space, multiple communication interfaces and a multitude of sensors, e.g., gyroscope, accelerometer, GPS, temperature, light, camera, among others [23].

Thus, a new paradigm emerged in which, in some scenarios, the sensing tasks are performed by such mobile devices, continuously and in the background. Applications are typically interested in monitoring tasks, e.g., weather variables and mobility. This Mobile Data Sensing paradigm is particularly interesting given the ubiquity of such devices, their density in some locations and the increasing sophistication and quantity of sensors included [20]. From a developer's point of view, the fact that these devices can be programmed in high-level programming languages such as Objective-C and Java, with rich APIs to access the hardware, makes them more attractive and flexible than WSN [28]. Another paradigm that uses data collected by mobile devices is Mobile Crowd Sensing in which tasks are injected into the devices to be executed with the explicit intervention of the users, e.g., surveys and processing data sets [11].

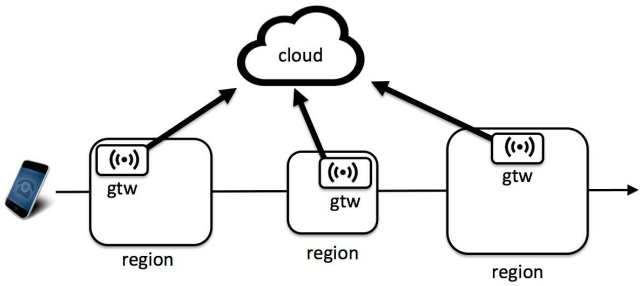


Figure 1: Dynamically reconfigurable tasks for mobile sensing.

While there are quite a few proposals of systems for Mobile Data Sensing, most focus on the infrastructure required to move the data from the devices into the cloud infrastructure, for storage or processing, and on specific applications that take advantage of the flexibility of mobile networks. We are not aware of any system that is capable of dynamic reconfiguration, in the sense that the sensing tasks they perform can be changed according to the geographical context of the device as depicted in Figure 1. In current systems, if a new type of sensing activity is desired, a new application with the appropriate code must be uploaded to the devices and executed. We feel that this is a major obstacle in the management of such

BuildSys'17, November 8–9, 2017, Delft, The Netherlands

ACM ISBN 978-1-4503-5544-5/17/11…$15.00
https://doi.org/10.1145/3137133.3137139

deployments as uploading and installing a new application is highly intrusive and resource consuming.

We envision a dynamically reconfigurable system where tasks are injected and removed seamlessly from devices. In this view, there are regions controlled by one or more gateways connected to a cloud infrastructure. Each gateway has a pool of tasks that it injects into any device roaming within its region. While the device his within the domain of such a gateway, it will execute those specific tasks and provide the corresponding data streams to the gateway, which forwards them to the cloud infrastructure. When a device leaves a region, the task pool is automatically deleted. A typical device can traverse many such regions (e.g., shops, buildings, campuses, roads) and perform manifold sensing tasks.

We believe there is a wide scope for these ideas in real-world use. Consider smart buildings, say shopping malls, where tasks can keep track of the location of the potential clients and send directed publicity or simply gather data that can be used to produce analytics important for the management of the commercial enterprise (e.g., is the interior layout of the building adequate?) and store owners (e.g., is our store attractive?); engaging users may rely on a sweepstakes system giving bonus for payments in certain shops, movie tickets, etc. Other applications can be considered, for instance in the realm of citizen science, where mobile crowd-sensing has been quite successful in engaging volunteers willing to contribute to science projects, or vehicular networks where individual data sensing shapes a global view of information (e.g., traffic data) that is useful for all participants.

In this paper we present an architecture and a prototype implementation for Flux, a service that is installed in smartphones once and receives asynchronous uploads of byte-code files corresponding to sensing tasks, compiled from a hardware independent domain specific language. The service runs the byte-code files in an internal, low footprint, virtual machine. As the tasks run they collect sensor data from the device, may do some processing and eventually send the final data to a gateway web service. Other administrative actions can be performed on the tasks per device, such as termination or code updates. Once the data reaches the gateway, it is forwarded to a publish/subscribe broker, another web-service, that in turn forwards the streams to interested clients. The clients receive the data in raw format with additional metadata describing the contents of the stream as they subscribe it, so that they can unpack it and pipe it to the next level of the processing stack.

As a case study, we asked volunteer students to install the Flux service in their smartphones and used the system to inject tasks that measured the number of Wifi networks detected by the device as well as the access point it is currently connected to. The streams generated by the smartphones as the students moved through the survey area were gathered and post-processed to provide a real time map of the Wifi coverage at the facilities.

The remainder of the paper is structured as follows. Section 2 describes the architecture and an implementation of Flux. Section 3 describes the domain specific language used to implement tasks, and Section 4 provides a description of the prototype implementation, along with a performance evaluation for the task service running on Android devices. Section 5 describes our case-study experiment. Section 6 describes the state-of-the-art in Mobile Data Sensing and Mobile Crowd Sensing, and compares it to the approach taken in Flux. Section 7 concludes the paper with a discussion of ongoing work on some additional features, and future research directions.

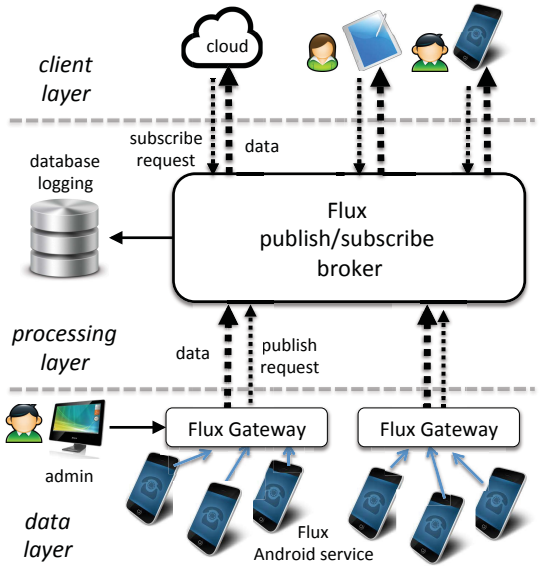


Figure 2: The Flux architecture.

2 ARCHITECTURE

Flux has a typical three-layer architecture where a set of clients, connected to the Internet, access data streams generated by mobile devices through a publish/subscribe broker. The streams are sent from mobile devices in a region to a gateway device. Figure 2 presents a high-level representation of these three layers and their relations.

2.1 Data Layer

A Flux gateway is the interface between the devices generating data streams and a P/S broker. A gateway keeps track of the mobile devices, connected through Wifi or 3G/4G, in a geographical region that have the Flux service installed. Such devices eventually register with the gateway once they enter the region. The gateway manages a dynamic task pool through operations that are mirrored in registered devices: the installation of new tasks, the removal of pre-existing ones, plus on-the-fly reconfiguration of active tasks in terms of periodicity and/or code updates. In interface with the broker, the gateway publishes the data streams associated with tasks injected in the devices and henceforth it forwards the corresponding data to the broker.

Task are executed by the Android Flux service running on the mobile devices, illustrated on Figure 3. The service senses the existence of gateways within network reach and registers itself. Over time, it then receives task setup actions (installations/removals/reconfigurations) from the gateway, and yields back data generated by running tasks.

Tasks are programmed using a domain-specific language called FTL (Flux Task Language), originally designed for WSN devices [13]. FTL is a statically-typed language for periodic sensing tasks that is

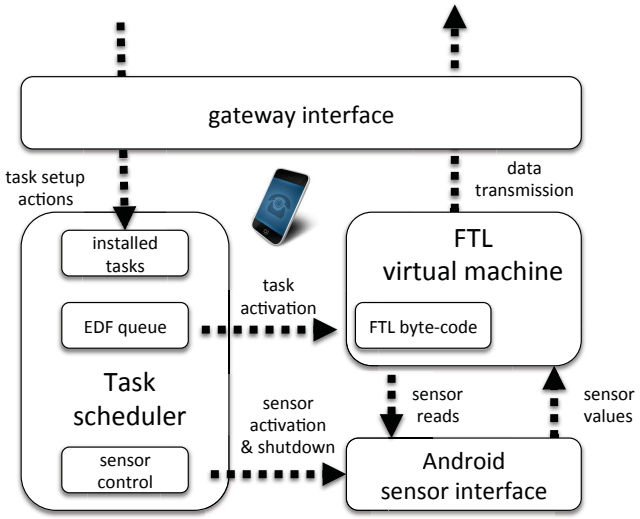


Figure 3: The Flux Android service.

parametric in the set of sensors available in each device, and is compiled to machine-independent byte-code, abstracting away from hardware and low level operating system details, whilst also providing a number of runtime safety properties. A very compact virtual machine (VM), ported to Android in the context of Flux, runs the byte-code with very low memory and CPU footprint, and using earliest-deadline first (EDF) scheduling for task activation. Tasks are programmed and compiled offline and injected in gateways using an administration module.

Figure 4 lists the code for the FTL task used in the case-study experiment described in Section 5. The purpose of the task is to collect data for GPS-referenced Wifi signal coverage. As mobile devices (users) change their location, the task measures the signal of the Wifi internet connection, the total number of Wifi networks currently detected, and tags these values with an estimate of the mobile device position using GPS. The details of the FTL language, VM, and the underlying runtime safety properties are discussed below in Section 3.

2.2 Processing Layer

The processing layer is composed of a publish/subscribe broker that keeps track of a collection of gateways, contributing with data streams, and a collection of clients, that subscribe to the data streams. The data is live-streamed from gateways to clients according to subscription parameters (e.g., split per task). For offline data analysis, the broker also maintains a database where data streams are logged for a (parameterizable) time window, making it possible for clients to access streams that were captured in the recent past rather than live-streamed.

2.3 Client Layer

Flux clients connect to the broker by first requesting a list of available streams and then selecting which to receive through subscription commands. An example client takes the form of a web browser app, depicted in Figure 5, where live streams can be visualised in real-time or logged data streams can also be plotted. A command-line tool is also available that can readily be composed (e.g., using pipes) with arbitrary data processing scripts for, e.g., data mining and interface with cloud services.

3 THE FTL LANGUAGE AND VM

We now cover the main traits the FTL language, associated guarantees of runtime safety, plus the FTL virtual machine, crucial aspects of Android service and the overall Flux proposal.

3.1 FTL code

The FTL code for a task is structured in three block sections, as illustrated in the example of Figure 4. The **sensors** section contains a description of the sensors used by the task, where each sensor is defined by a type signature. The **init** block declares and initialises task variables, that persist in memory across task invocations. Finally, the **loop** block contains the actual instructions that execute

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

init {
   int number_wifi = 0;
   int wifi_level  = 0;
   float latitude  = 0.0;
   float longitude = 0.0;
   float altitude  = 0.0;
   float accuracy  = 0.0;
}

[ int   @ "NUMBER_WIFI_NETWORKS: # networks",
  int   @ "WIFI_SIGNAL_LEVEL: dBm",
  float @ "LOCATION latitude: degrees",
  float @ "LOCATION longitude: degrees",
  float @ "LOCATION altitude: meters",
  float @ "LOCATION accuracy: meters" ]

loop {
   number_wifi = NUMBER_WIFI_NETWORKS();
   wifi_level  = WIFI_SIGNAL_LEVEL();
   latitude    = LOCATION(0);
   longitude   = LOCATION(1);
   altitude    = LOCATION(2);
   accuracy    = LOCATION(3);

   # send only when position is accurate
   if (accuracy <= 10) {
      radio[number_wifi, wifi_level, latitude,
            longitude, altitude, accuracy];
   }
}
```

Figure 4: FTL task for geo-referenced Wifi data collection.

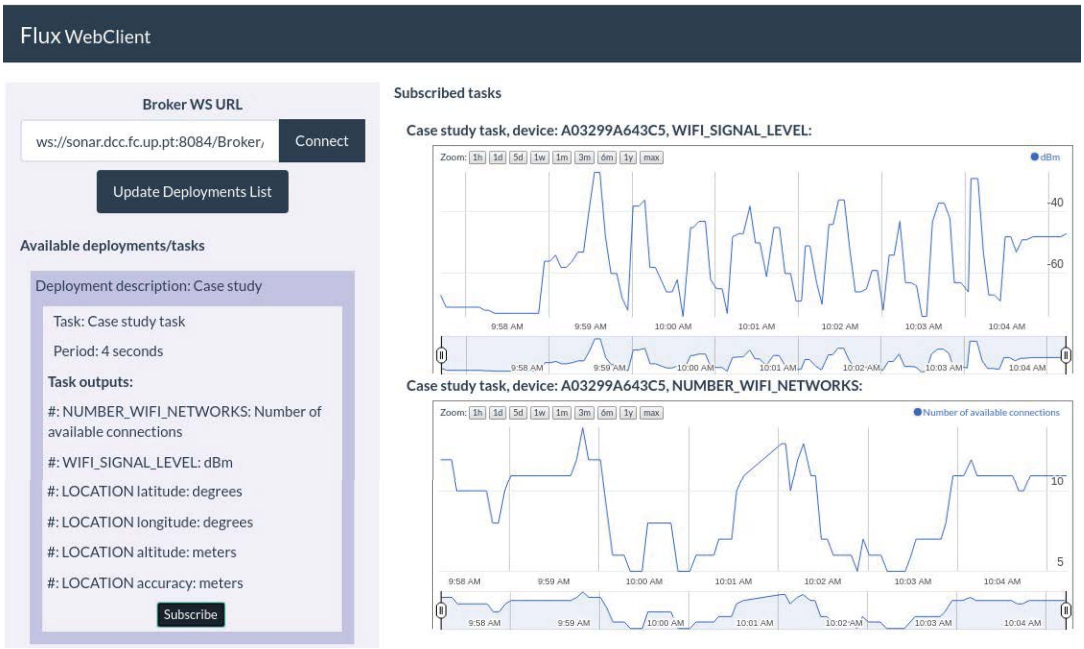


Figure 5: Live data streaming on a web browser

every time the task is activated; the activation period itself is not defined by the task's code, but instead configured by the gateway upon installation. The **loop** block is annotated with a preceding definition for the type for messages that can be sent in its body (only one type of message can be sent), in terms of component fields, their type, label, and textual description of the units used.

The example **loop** block illustrates the FTL support for sensor reading, variable assignments, conditional branching (the **if** construct), and data transmission (the **radio** instruction). The code proceeds by first reading the Wifi and GPS measurements onto task variables, and then transmitting if the GPS accuracy does not exceed a threshold of 10 meters. Beyond the instructions shown in the example, FTL also provides support for basic arithmetic over scalar values and logical/relational operators over boolean values.

3.2 Runtime safety

By construction, FTL is quite constrained in order to provide guarantees of safe execution and predictable memory footprint; we discuss possible extensions of the language as future work in Section 7.

In terms of control flow, branching is strictly limited to plain if-else blocks as in the example task. Thus, FTL provides no support for iteration, function calls, or recursion. This guarantees proper termination of each task activation, while, in our view, still providing reasonable expressiveness for plain data sensing.

To guarantee memory access safety, FTL provides support for scalar types only (integer, floating-point, and boolean), excluding composite types like arrays or lists for instance, which could make it extremely complex to guarantee memory access safety at compile-time. The type constraints also imply that a precise memory footprint is inferred by the compiler for a task. This footprint is bounded by the task variables' memory plus the maximum possible size of the FTL VM stack (discussed below).

The use of the FTL language and VM also defines a secure sandbox, as opposed to a scheme where arbitrary Android code is (downloaded and) used for data sensing tasks, raising well-known, complex to detect and mitigate, security issues [10, 12].

3.3 FTL virtual machine

The listing shown in Figure 6 depicts the bytecode for the FTL task given in Figure 4 in text form; the actual binary representation has a size of just 137 bytes. The FTL VM is 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.

As shown in the figure, the byte-code has three sections: **data**, **stack** and **text**. The **data** section contains all the space for the program variables, corresponding initial values, and other program constants. The **stack**, whose size is precomputed statically, is used for data manipulation (e.g., arithmetic, argument passing and storage of return values) and, finally, the **text** section contains the actual instructions to be executed. As should be reasonably intuitive from the listing, instructions in the **text** block include loads (`ld`) onto the stack, stores (`st`) from the stack onto memory, sensor reads (`rd`), and radio transmission (`rad`). Other assorted instructions relate to arithmetic and flow control logic. For instance, the `bf 60` is a conditional branch to the instruction with program counter `60`, where `ret` ("return") ends the execution for a task activation; the flow corresponds to bypassing radio transmission, when the `accuracy` value exceeds 10, as in the original FTL code.

4 IMPLEMENTATION

In this section we describe some details of a prototype implementation of the Flux architecture, and present a basic performance and resource footprint evaluation for the Flux Android Service.

```
.total   137          .text
.offset  19           .0   rd NUMBER_WIFI_NETWORKS 0
                      .3   st number_wifi
.data                 .5   rd WIFI_SIGNAL_LEVEL 0
.0   0                .8   st wifi_level
.4   0.0              .10  ld #0
.8   1                .12  rd LOCATION 1
.12  2                .15  st latitude
.16  3                .17  ld #1
.20  10               .19  rd LOCATION 1
.24  number_wifi      .22  st longitude
.28  wifi_level       .24  ld #2
.32  latitude         .26  rd LOCATION 1
.36  longitude        .29  st altitude
.40  altitude         .31  ld #3
.44  accuracy         .33  rd LOCATION 1
                      .36  st accuracy
.stack                .38  ld accuracy
.48  0                .40  ld #10
.52  0                .42  f2i
.56  0                .43  fle
.60  0                .44  bf 60
.64  0                .46  ld number_wifi
.68  0                .48  ld wifi_level
.72  0                .50  ld latitude
                      .52  ld longitude
                      .54  ld altitude
                      .56  ld accuracy
                      .58  rad 6
                      .60  ret
```

Figure 6: Bytecode for the example FTL task

4.1 Programming framework

We used Java as the development language for all components, except for the Web browser client that was implemented in Javascript. The gateway and P/S broker were implemented as Apache Tomcat web-services [4], the Flux service was programmed on Android Studio [3] to run on Android 4.4 or higher, and the FTL compiler was implemented using the ANTLR compiler infrastructure [29]. All components of the Flux architecture communicate using a common message format, specified using Google's Protocol Buffers [5]. Plain TCP/IP sockets were used for gateway-device communications and Web-sockets for all other interactions (gateway-broker, client-broker, and admin-gateway). For logging the data streams at the broker we used an SQLite database [8].

4.2 The P/S broker and the gateway

The broker keeps track of all the region gateways and their task pools. This information is updated every time an authenticated user accesses a gateway to manage the corresponding task pool. Any modifications introduced by the user are communicated to the broker by the gateway. This information allows the clients to get more details about the registered tasks and is also relevant to perform other actions such as storing the streamed data in a more flexible way in the broker. Once the broker gets the first items from a data stream associated with a task, it creates a table on a SQLite database with the specific fields to match the task output data and metadata (e.g., name, units). The broker also keeps track of the active clients so it can forward the data or notify possible changes in the data stream.

Each gateway stores a list of registered tasks, previously uploaded by an administrative user, that can be injected or updated on the mobile devices. When a new device connects to the gateway, the latter compares the registered pool of tasks to the tasks running on the device and sends missing tasks or updates to the device. This synchronization of the running tasks also takes into account whether a given device meets all the requirements for running the tasks, e.g. if it has all necessary sensors. If a task's requirements are not fully met, it is ignored for that particular device. When a device detects that a gateway is no longer connected, the task associated with it are killed and the memory reclaimed.

4.3 The Android service

The implementation of the Android service follows the organization depicted in Figure 3. It is composed of four main modules: the gateway interface, the task scheduler, the FTL virtual machine, and a sensor control interface. As a service, it runs in the background without need for user interaction. A user-interface application, shown in Figure 7, can in any case be used to turn the gateway connection on or off, or the entire service on or off, besides providing basic information regarding the state of running tasks.

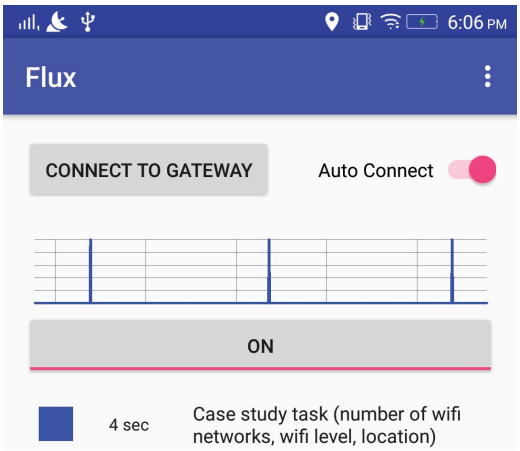


Figure 7: The Flux service Android application.

The basic rationale and functionality of the scheduler and the virtual machine modules were discussed earlier in the paper (Sections 2 and 3), hence we merely provide some complementary details regarding the implementation of the sensor control and gateway interfaces.

The sensor control interface is responsible for obtaining sensor readings, interacting with assorted Android OS APIs for that purpose. The task scheduler directs it to enable sensors on/off as tasks are installed/removed, and the virtual machine obtains sensor readings from it. The module implements an adaptive activation/deactivation strategy for sensors. Active sensors may consume significant battery power (e.g., as is the case of GPS), whilst on the other

hand their repeated initialisation/shutdown may cause unnecessary latency. In particular, initialisation may imply high latency until valid readings are obtained (again as in the case of GPS).

The sensor activation strategy takes into account the periodicity of tasks with respect to the sensors they read. For a task with 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 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. The period threshold is configured per each type of sensor, attending to a (for now relatively heuristic) balance between initialisation latency and battery consumption, for instance the threshold is currently set to 2 minutes for GPS.

Regarding the gateway interface, it employs some built-in data buffering mechanisms for network resilience and for reducing battery/bandwidth consumption. Data produced by tasks is buffered when a connection to the gateway is lost, making the service robust to network outages. Moreover, time and buffer size limits may be set and fine-tuned if desired, so that transmission to the gateway occurs periodically using buffered data, rather than continuously using live data.

4.4 Performance evaluation

We conducted an evaluation of the Android service in terms of resource consumption and virtual machine performance during byte-code execution. For the evaluation we used a Google Nexus tablet running Android 6.0 with 2 GB of RAM and a dual-core 2.3 GHz CPU, plus a gateway 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. Note also that a much more lightweight configuration can be used for hosting a gateway (and/or a broker), like the one for the case-study experiment discussed in Section 5.

The service 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 four other 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: (1) the example Wifi survey task running at 1 Hz, (2) an atmospheric pressure sensing task at 2 Hz, (3) a gyroscope sensing task at 4 Hz, and (4) 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 utilisation and RAM usage in 1-second intervals, plus the total of the TCP/IP data transmitted by the service in each 2-minute interval. The results for the resource consumption (with the corresponding 95% confidence intervals) are shown in Table 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.

Table 1: Resource consumption.

Tasks	CPU (%)	RAM (KB)	Net. (bytes/s)
None (∅)	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 2: Byte-code size and execution time.

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

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 2, that lists the byte-code size and average execution time in milliseconds per each of tasks. 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 CASE STUDY

For evaluating Flux, we conducted a controlled real-world experiment 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 Flux Android service executed an FTL task to collect GPS-referenced Wifi signal data and streamed that data to a Flux gateway. The following sections describe this experiment in detail.

5.1 Outline

The survey area, depicted in Figure 8, has a dimension of roughly 100×150 meters, and comprises the Computer Science department

Figure 8: Survey area.

building that is part of the Faculty of Science of our university (A in the figure), plus walkways in a garden north of the same building[1]. The figure also depicts an outline of the paths followed by volunteer users carrying mobile devices, covering corridors within the department building plus walkways outside. The walkways pass through the outside of two other university buildings (B and C in the figure).

Open-air GPS precision was better than within the building (as expectable), but anyway judged to be fair enough in both vicinities (as discussed below). The department building has two floors, but data was sampled only for the second floor, since most of the ground floor has reserved access (there is only a small portion of corridors).

The Wifi network subject to monitoring is the eduroam[2] instalment at our university, the most commonly used campus network by students and staff. The FTL task running on devices collected two items of information over time and space: the Wifi signal strength, and the number of nearby networks also detected by Android. The aim was to analyse a suspected inverse correlation between eduroam's Wifi signal strength and interference from other active networks, in addition to physical location.

[1]The satellite and map imagery used in this paper was obtained from Google Earth, in compliance with Google's terms (https://www.google.com/permissions/geoguidelines.html), and Open Street Maps, in compliance with the ODbL license (http://www.openstreetmap.org/copyright).
[2]http://eduroam.org

5.2 Setup

For the experiment, we used a CentOS Linux virtual machine (CentOS VM) with 2 cores and 1837 MB of RAM, hosted on a 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 Flux Android service to install tasks (and relay data) from (to) the gateway, and external clients to access the P/S broker. This is a relatively simple setup, but one that served the purpose of the experiment; note that, as mentioned earlier in the paper, multiple gateways running on different hosts can be used, interacting with a broker on another, possibly distinct, host.

Table 3: Android device characteristics

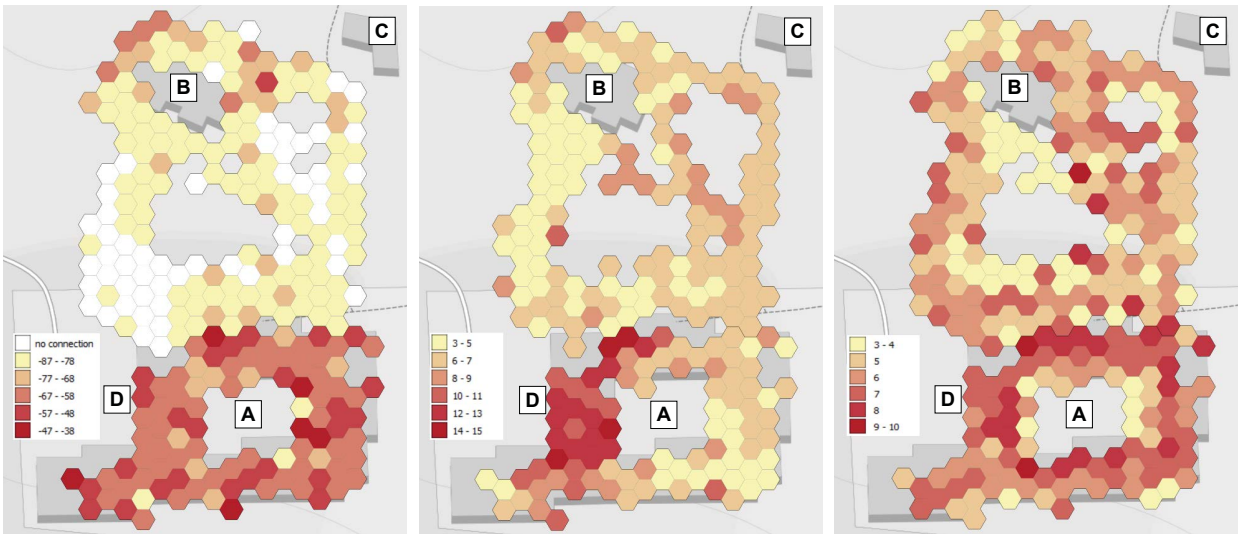
Type	Version	Vendor
Tablet	6.0	Google (9)
Smartphone	7.0	Samsung (1)
	6.0	Asus (1), Huawei (1), Lenovo (1), LG (1), OnePlus (2), Vodafone (1), Wiko (2)
	5.1	Xiaomi (1)
	4.4	Alcatel (1)

For measurements we used a total of 23 devices, divided in 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 3 summarises the basic characteristics of these devices. The Android service was installed in each of the devices, followed by an automatic download and installation of the FTL task for the survey by the service itself, as soon as it got a connection to the gateway. The FTL task is the same as described earlier in Section 2, with the difference that no HDOP filter is set when transmitting to the gateway (i.e., the **if** guard condition in Figure 4 is omitted), and was configured to run with a periodicity of 4 seconds.

5.3 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 1922 samples (69% of the original) inside the building and to 1212 samples (79% of the original) outside. Figure 9 depicts the filtered data set as geo-referenced "heat maps", in terms of the eduroam Wifi signal strength (9a), the number of detected Wifi networks (9b), and the GPS HDOP (9c). In the plots, rendered using QGis [7], the colors depicts the average measure for data points within each hexagon that forms the heat map (buildings are marked A to C as in Figure 8).

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



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

Figure 9: Data plots for collected data

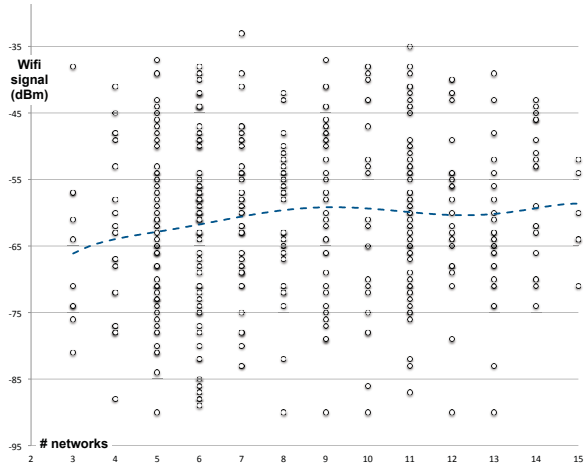


Figure 10: Wifi signal vs. number of Wifi networks inside the department building.

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).

Regarding interference between eduroam and other networks, we can observe areas inside the building a where a significantly higher number of networks are active, on the west side particularly where a considerable number of computer labs are concentrated, the D "hotspot" in the plots of Figure 9b. From the plot it seems apparent that these do not interfere with eduroam's Wifi signal significantly, however. To clarify the analysis, we depict a scatter plot in Figure 10 relating the Wifi signal and the number of networks; no correlation pattern emerges, as illustrated by the relatively uniform distribution of scatter points per network count, and the point is reinforced by the trend line shown for the average signal. We did not pursue an exhaustive analysis of this finding, but conjecture that it relates to the fact that there are several eduroam's access points scattered around the building, and suspect that they should also typically have a stronger signal than more modest special-purpose Wifi access points/routers operating in computer labs.

6 RELATED WORK

Mobile Data Sensing and Mobile Crowd Sensing are emerging paradigms with proven monitoring applications in areas like health, environment urban management, and citizen science. In what follows, we survey several examples of state-of-the-art systems and applications, then make an overall comparative assessment with the Flux proposal.

SmartRoad [18] 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 collects data from the GPS sensor. It uses a client-server architecture in which the smartphone acquires the data and sends it to the server where it is processed. An interesting 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 to improve the results. Users are motivated to participate by providing the collected data as input for navigation systems.

NoizCrowd [32] proposes the use of smartphone sensors to collect noise levels from a region for the purpose of generating accurate noise models. A large data set is required for this, hence the option for a crowdsourcing application. The system consists of four components: (a) an application located in the smartphones that is responsible for recording the noise levels, using the microphone, and the current position, using the GPS sensor; (b) storage to keep

all the data received from the application; (c) a model generator, and; (d) a visualisation and data transfer component. The data collection consists of recording the mean noise levels in decibels, as well as measuring peaks, in intervals of only a few seconds. The data is then sent to the system's data storage via a web service.

Medusa [24] 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. Each task is defined by a XML-based domain-specific language, that provides high-level abstraction for specifying the sequence of steps in the sensing task. The framework consists of a distributed runtime system separated in two main components: the Medusa Cloud Runtime and the Medusa Runtime on the Smartphone. The Cloud Runtime is responsible for receiving and parsing the tasks, keep track of the different generated instances for each task and manage the associated workers. This uses the Amazon Mechanical Turk [1] 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.

Sensus [33] is a system designed for 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. The system is composed of an application that runs on the mobile devices and a cloud storage. Tasks are generated using a mobile app and disseminated to the study participants through the Amazon Mechanical Turk [1], 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 S3 [2] for retrieval and analysis by researchers.

SARANA [17] is a system architecture that supports the development of applications that execute tasks on remote devices based on the services they can provide (e.g. camera, image analysis). It provides a language and a run-time system that allow programmers to express spatial regions of interest as well as resource constraints needed to run the tasks. SARANA makes use of a domain-specific language, a superset of Java, that provides abstractions for device discovery, task distribution under spatial, temporal, and resource constraints, and processing of aggregated data.

Device Analyzer [30] is a mobile application, developed at the University of Cambridge, that collects data from mobile phones and transmits it to a central server where the dataset is kept and analysed for pattern extraction. 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.

PressureNet [21] is the first application for smartphones built on top of atmosphere sensors, in this case barometers. The information collected from the devices is uploaded to the cloud where it can be used as a real-time weather monitoring and forecast network.

OpenSignal [6] is an application that uses crowdsourced information on wireless connections and signal strength to create a connectivity map for users. Using this feedback, the application can give the users instructions, e.g., move in a certain direction, to improve their connectivity. The application also provides speed tests directed to popular service providers like Google and Facebook and upload/download tests using popular CDNs.

Zooniverse [27] is a quite successful citizen-science platform that allows users of mobile devices to contribute to scientific projects, e.g., by helping with the processing of large datasets. After installing the mobile application the user selects one or more projects to contribute and receives data chunks to process. The results are submitted by the application to the cloud.

Flux is more closely related to systems like Medusa, Sensus, and SARANA, in the sense that it uses a domain-specific language to define sensing tasks and provides the necessary infrastructure to inject them into the mobile devices. By the use of a constrained language and VM, however, Flux guarantees safe and predictable execution and resource usage. Also, unlike Medusa and Sensus, Flux tasks run continuously in the background gathering sensor data, without need for user intervention. And, unlike SARANA, all system layers operate modularly and are configured independently.

In Flux, tasks are associated with geographical regions managed by gateways, allowing for scenarios in which devices can roam between regions and perform different tasks accordingly. SARANA, discussed earlier, also accounts for device location when assigning tasks. The concept of region has also been proposed as a programming abstraction for WSN [31]: a region can be defined based on criteria such as device location, radio connectivity, or values for sensed data. The main purpose is to enable online data pre-processing or aggregation to reduce the bandwidth (and energy) required to send the streams to the base stations. Regiment [22], for example, uses regions in combination with data streams as fundamental programming abstractions. As a functional macro-programming language, it does not support the dynamical reconfiguration of the tasks since the sensor network needs to be reprogrammed as a whole. Agilla [15], on the other hand, implements a mobile-agent programming flavour, where each agent (a task) can proactively migrate across the network, executing code that depends on the conditions sensed at each node. It differs from Flux in that nodes are typically geographically fixed, whereas tasks may move.

7 FINAL REMARKS

In this paper we presented the Flux framework for streaming sensor data from dynamically reprogrammable tasks injected in mobile devices. We described its architecture, prototype implementation, and a case-study application as a proof-of-concept, demonstrating the dynamic injection of tasks in mobile devices moving around in a given region, and the acquisition of the corresponding data streams. Flux tasks are developed in the FTL domain-specific programming language, boasting enough expressiveness for basic sensing tasks while providing compile time guarantees of runtime safety, and compiled to extremely compact byte-code that is executed by a low-footprint virtual machine on Android.

We have a working proof-of-concept of the system that implements the roaming scenario presented in Figure 1, i.e., letting devices engage with gateways dynamically as their physical location changes. As a first approximation we opted to define regions with geographical boundaries and associated with specific gateways that inject tasks in the devices and receive the data produced by the devices. Regions, however, can be defined more broadly, as a set of boundary conditions on attributes other than geographical position, e.g., the region formed by all devices reading temperatures higher

than 30°C and humidity below 50%, the region formed by devices whose owners are interested in sports. Such attribute-based regions have been suggested in the setting of programming languages for WSN [22, 31], mostly with the goal of supporting online data aggregation and reducing power consumption due to communication. In the mobile setting, however, the hardware and energy limitations of WSN are not present and more sophisticated uses of regions are perhaps possible. Such regions would also provide the means to make task submission more fine-grained in the sense that only a subset of all devices in a geographical region, those with certain attributes, are injected with a given task pool.

In association, the policy used to inject/kill the tasks, is, of course, 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. Flux is already resilient to problems such as a temporary failure or disconnection from the device despite being in the region; when it surfaces again, the device synchronises with the gateway again and the tasks are reloaded. There are also failures related to errors in the sensors, e.g., missing a reading or providing a very imprecise, making it difficult to establish whether or not a device is still within a region's boundaries, and errors related to rapidly changing conditions, e.g., devices moving too fast between regions for correct task injection/killing semantics to be enforced. These are problems for future work. We are also looking at event-driven activation for tasks, beyond the current support for strict periodic activation. The motivation is that many sensing activities are not continuous over time but instead triggered by the onset of certain environmental conditions (spatial location, hour of the day, temperature, available light, etc) or by the users themselves in crowd-sensing scenarios.

Extending FTL for more expressive online data processing, whilst preserving runtime-time safety guarantees, is another topic worthy of future research. Currently the language is quite simple, with scalar types, sensor/actuator I/O, basic arithmetic and control flow. Adding constructs, e.g., in support of iteration or array types, can in principle be built-in into the FTL compiler leveraging technologies like SMT solvers [19] to preserve runtime safety. Furthermore, FTL currently has no communication constructs that allow neighbouring nodes to exchange data for aggregation or pre-processing purposes. This is particularly desirable given the rich networking capabilities of mobile devices, in particular in the context of mobile edge-clouds, where nearby devices form a network to work collaboratively, a topic we are also currently working on [25, 26].

ACKNOWLEDGMENTS

This work was funded by projects HYRAX (CMUP-ERI/FIA/0048/2013, FCT) and SMILES (NORTE-01-0145-FEDER-000020, NORTE 2020).

REFERENCES

[1] [n. d.]. Amazon Mechanical Turk. https://www.mturk.com/. ([n. d.]).
[2] [n. d.]. Amazon Web Services Simple Storage Service (S3). https://aws.amazon.com/s3/. ([n. d.]).
[3] [n. d.]. Android Studio. https://developer.android.com/studio/. ([n. d.]).
[4] [n. d.]. Apache Tomcat. http://tomcat.apache.org/. ([n. d.]).
[5] [n. d.]. Google Protocol Buffers. https://developers.google.com/protocol-buffers/. ([n. d.]).
[6] [n. d.]. OpenSignal. https://opensignal.com/. ([n. d.]).
[7] [n. d.]. QGis. http://www.qgis.org/. ([n. d.]).
[8] [n. d.]. SQLite. https://www.sqlite.org/. ([n. d.]).
[9] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. 2002. A Survey on Sensor Networks. *IEEE Communications Magazine* 40, 8 (2002), 102–114.
[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. PLDI*. ACM, 259–269.
[11] B. Guo and Z. Wang and Z. Yu and Y. Wang and N. Y. Yen and R. Huang and X. Zhou. 2015. Mobile Crowd Sensing and Computing: The Review of an Emerging Human-Powered Sensing Paradigm. *ACM Computing Surveys* 48, 1 (2015), 1–31.
[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. 2011. A Study of Android Application Security. In *Proc. SEC*. USENIX, 21–21.
[13] G. Ferro, R. Silva, and L. Lopes. 2015. Towards Out-of-the-Box Programming of Wireless Sensor-Actuator Networks. In *Proc. CSE*. IEEE, 110–119. https://doi.org/10.1109/CSE.2015.20
[14] C.-L. Fok, G.-C. Roman, and C. Lu. 2005. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In *Proc. ICDCS*. IEEE, 653–662.
[15] C. L. Fok, G. C. Roman, and C. Lu. 2009. Agilla: A Mobile Agent Middleware for Self-adaptive Wireless Sensor Networks. *ACM Trans. Auton. Adapt. Syst.* (2009), 16:1–16:26. https://doi.org/10.1145/1552297.1552299
[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. 2003. The nesC Language: A Holistic Approach to Network Embedded Systems. In *Proc. PLDI*. ACM, 1–11.
[17] P. Hari, K. Ko, E. Koukoumidis, U. Kremer, M. Martonosi, D. Ottoni, L.-S. Peh, and P. Zhang. 2008. SARANA: Language, compiler and run-time system support for spatially aware and resource-aware mobile computing. *Philosophical Transactions of the Royal Society A* 366 (2008), 3699–3708.
[18] S. Hu, L. Su, H. Liu, H. Wang, and T. F. Abdelzaher. 2015. SmartRoad: Smartphone-Based Crowd Sensing for Traffic Regulator Detection and Identification. *ACM Transactions on Sensor Networks* (2015), 55:1–55:27. https://doi.org/10.1145/2770876
[19] S. Lahiri and S. Qadeer. 2008. Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. In *Proc. POPL*. ACM, 171–182.
[20] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell. 2010. A Survey of Mobile Phone Sensing. *IEEE Communications Magazine* 48, 9 (2010), 140–150.
[21] C. F. Mass and L. E. Madaus. 2014. Surface pressure observations from smartphones: A potential revolution for high-resolution weather prediction? *Bulletin of the American Meteorological Society* 95, 9 (2014), 1343–1349.
[22] R. Newton and M. Welsh. 2004. Region Streams: Functional Macroprogramming for Sensor Networks. In *Proc. DMSN*. ACM, 78–87.
[23] P. Piejko. [n. d.]. Global Mobile Statistics 2017. https://mobiforge.com/research-analysis/13-statistics-on-mobile-web-performance-in-2017. ([n. d.]).
[24] M-R. Ra, B. Liu, T. F. La Porta, and R. Govindan. 2012. Medusa: A Programming Framework for Crowd-sensing Applications. In *Proc. MobiSys*. ACM, 337–350. https://doi.org/10.1145/2307636.2307668
[25] J. Rodrigues, J. Silva, R. Martins, L. Lopes, U. Drolia, P. Narasimhan, and F. Silva. 2016. Benchmarking Wireless Protocols for Feasibility in Supporting Crowd-sourced Mobile Computing. In *Proc. DAIS*. Springer, 96–108.
[26] P. M. P. Silva, J. Rodrigues, J. Silva, R. Martins, L. Lopes, and F. Silva. 2017. Using Edge-Clouds to Reduce Load on Traditional WiFi Infrastructure and Improve Quality of Experience. In *Proc. ICFEC*. IEEE, 61–67.
[27] R. Simpson, K. R. Page, and D. De Roure. 2014. Zooniverse: Observing the World's Largest Citizen Science Platform. In *Proc. WWW*. ACM, 1049–1054.
[28] The Internet Society. 2015. Internet Society Global Report 2015 - Mobile Evolution and Development of the Internet. https://www.internetsociety.org/globalinternetreport/2015/assets/download/IS_web.pdf. (2015).
[29] T. Parr. [n. d.]. ANTLR (ANother Tool for Language Recognition). http://www.antlr.org/. ([n. d.]).
[30] D. T. Wagner, A. Rice, and A. R. Beresford. 2013. Device Analyzer: Understanding Smartphone Usage. In *Proc. MobiQuitous*. Springer, 195–208.
[31] M. Welsh and G. Mainland. 2004. Programming Sensor Networks Using Abstract Regions. In *Proc. NSDI*. USENIX Association.
[32] M. Wisniewski, G. Demartini, A. Malatras, and P. Cudré-Mauroux. 2013. NoizCrowd: A Crowd-Based Data Gathering and Management System for Noise Level Data. In *Proc. MobiWIS*. Springer, 172–186. https://doi.org/10.1007/978-3-642-40276-0_14
[33] H. Xiong, Y. Huang, L. E. Barnes, and M. S. Gerber. 2016. Sensus: A Cross-platform, General-purpose System for Mobile Crowdsensing in Human-subject Studies. In *Proc. UbiComp*. ACM, 415–426. https://doi.org/10.1145/2971648.2971711