# Flux: A Platform for Dynamically Reconfigurable Mobile Crowd-Sensing

NUNO SILVA, EDUARDO R. B. MARQUES, and LUÍS M. B. LOPES, Faculty of Science, University of Porto & CRACS/INESC-TEC

Flux is a platform for dynamically reconfigurable crowd-sensing using mobile devices like smartphones and tablets, programmed under a notion of region-based sensing. Each region is defined by a set of physical constraints that determine the sensing scope, e.g., based on device position or other environmental variables, plus a set of periodic tasks that perform the actual sensing. The resulting behavior is inherently dynamic: as a device's state changes, e.g., moves in space, it enters and/or leaves different regions, thereby changing the set of active tasks; moreover, regions can be added, deleted, and reprogrammed on-the-fly. Flux makes use of a domain-specific language for sensing tasks that is compiled into abstract bytecode, later executed by a low-footprint virtual machine within a device, guaranteeing runtime safety by construction. For region/task dissemination, Flux employs a broker that holds a changeable region configuration plus gateways that mirror the configuration throughout different network access points to which devices connect. Sensing data is streamed by devices to gateways and then back to the broker. Live or archived data streams are in turn fed by the broker to data-processing clients, which interface with the broker using a publish/subscribe API. We conducted two case-study experiments illustrating Flux: a single-region deployment to monitor WiFi signal quality, and a multi-region deployment to monitor noise, temperature, and places-of-interest based on device movement.

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

Additional Key Words and Phrases: Mobile crowd-sensing, software architecture, domain-specific language, virtual machine, android

**ACM Reference format:**
Nuno Silva, Eduardo R. B. Marques, and Luís M. B. Lopes. 2018. Flux: A Platform for Dynamically Reconfigurable Mobile Crowd-Sensing. *ACM Trans. Sen. Netw.* 14, 3–4, Article 20 (November 2018), 25 pages.
https://doi.org/10.1145/3200202

This work was supported by projects TEC4Growth - RL SMILES - Smart, Mobile, Intelligent and Large scale Sensing and analytics (NORTE-01-0145-FEDER-000020, NORTE 2020) and HYRAX (CMUP-ERI/FIA/0048/2013, FCT). An earlier version of the work presented in this article was published in the ACM International Conference on Systems for Energy-Efficient Built Environments (BuildSys 2017), Delft, The Netherlands, November 2017.
Authors' addresses: N. Silva, E. R. B. Marques, and L. M. B. Lopes, Faculty of Science, University of Porto & CRACS/INESC-TEC, Rua do Campo Alegre, 1021, Porto, 4169-007, Portugal; emails: nmsilva@fc.up.pt, edrdo@dcc.fc.up.pt, lblopes@dcc.fc.up.pt.

1550-4859/2018/11-ART20 $15.00
https://doi.org/10.1145/3200202

## 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 specialised sensors (Akyildiz et al. 2002). 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 (Fok et al. 2009; Gay et al. 2003; Newton and Welsh 2004). Today, however, we have more than 1 billion potential multi-sensor personal devices in our smartphones or wearables that are more flexible and becoming more attractive to use than WSNs. Smartphones have become ubiquitous and typically feature powerful multi-core processors, several gigabytes of storage space, multiple communication interfaces and sensors, and can be programmed using general-purpose programming languages (Piejko 2017; Society 2015).

Given the ubiquity of mobile devices, their density in some locations and the increasing sophistication and quantity of sensors, we witnessed the emergence of Mobile Crowd-Sensing (MCS) (Guo et al. 2015; Lane et al. 2010). MCS applications may concern passive monitoring in a spatially aware manner, e.g., environment variables or user mobility patterns, but also explicitly engage users in sensing actions, by triggering sensing actions that require their intervention (e.g., photo acquisition) or using the human as "sensor" for event reporting (e.g., accidents in road traffic).

In this context, most proposals focus on the infrastructure required to move the data from the devices into the cloud infrastructure, for storage or processing, or on specific applications for the context of mobile networks. Comparatively little work has been done to address the need of dynamically re-programming these systems and the varying nature of sensing tasks attending to environmental conditions/parameterisation, e.g., changing location due to device mobility. Current systems are often inflexible and have a low degree of automation, mostly relying on application installations/updates and user intervention to control the sensing process.

To address these concerns, we consider dynamically reprogrammable crowd-sensing using mobile devices, with the overall approach illustrated in Figure 1. We consider that devices automatically retrieve sensing configurations from the network, expressed as a set of *regions*. Each region is defined by a set of physical constraints that determine the sensing scope, e.g., by encoding a geographical zone of interest for sensing and/or predicates over other environmental variables, plus a set of tasks that define the actual sensing. When a device enters a region (verifies the region's constraints), it activates the tasks mapped to that region and starts streaming back the corresponding sensing data to the network; eventually (tasks for) one or more regions may be active at a given time, as enabling constraints for different regions may be verified simultaneously, e.g., regions may overlap in spatial coordinates. When a device leaves a region (the region's constraints no longer hold), the tasks are deactivated. Thus, different regions (tasks) may be active over time, for instance, as the location of a device changes. An additional level of dynamic behavior may be obtained if we let the set of regions be changed over time, implying their dynamic installation/removal along with associated tasks, something that should happen seamlessly, without the intervention of the mobile device users.

We believe this approach is in line with the wide scope of crowd-sensing in the real-world. Public or commercial buildings may have several zones of interest and distinct sensing requirements for each, e.g., shopping malls, where there are different spaces (shops, walking areas, etc.). Regions can also be useful abstractions in different realms, for instance, in citizen science apps to direct sensing tasks/participant users to under-sampled areas, or in transportation systems to differentiate between different routes. Naturally, user participation incentives need to be accounted for. In each of the previous realms, different ones can be considered, e.g., a sweepstakes system giving a

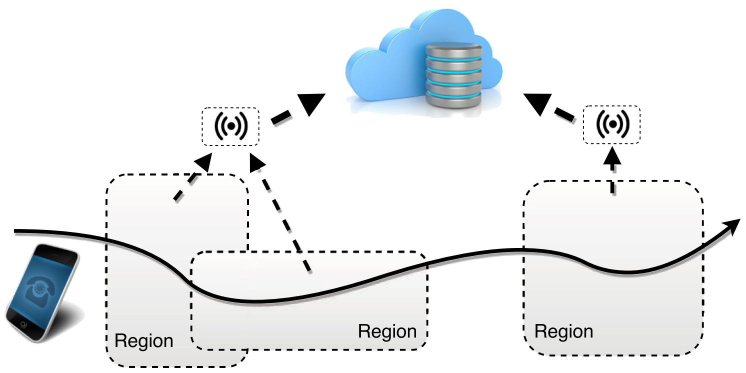


Fig. 1. Dynamic region-based sensing tasks for mobile devices.

bonus for payments in certain shops or movie tickets in shopping malls, co-authorship of a paper in citizen science, or the possibility of obtaining a global view of information in transport systems shaped from data collected by all participants.

In this article, we present Flux, a MCS platform using region-based programming. Flux takes form through an Android service that is installed in devices for data collection and a cloud-based infrastructure for global region-based programming and dissemination. The Android service maintains a set of regions that may be (re)configured through the network for the host device and executes sensing tasks for the regions that become active over time. Tasks are programmed using a domain-specific language and received by devices in a compiled bytecode format amenable to safe execution by a low-footprint virtual machine within the Android service. For region dissemination, Flux employs a cloud broker that holds a changeable set of region configurations, plus gateways that propagate these configurations through different network access points to which devices may connect. The idea is that, if necessary, gateways can typically be deployed near the edge of the network (e.g., as special-purpose access points or cloudlets), alleviating the burden and/or functioning as proxy for the centralised broker. Sensing data is streamed by devices to gateways and then back to the broker. Data-processing clients may in turn interface with the broker using a publish/subscribe API to access live or archived data streams.

To illustrate the applicability of Flux, we asked volunteer students to install the Flux service in their smartphones and conducted two case-study experiments in the physical space of our department building and surrounding gardens. In the first case-study, we defined a single region where geo-referenced WiFi coverage data was sampled. The streams generated by the personal devices 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. In the second case-study, we defined a three-region setting where we took geo-referenced measures of (1) audio noise in the department building, (2) places-of-interest based on (absence of) device movement in surrounding gardens, (3) and temperature in the entire survey area.

The remainder of the article is structured as follows. Section 2 provides an overview of Flux in terms of underlying requirements, architecture, and component sub-systems. Section 3 describes how the region-based programming works, in terms of region specification, the domain-specific language used to define tasks, and the virtual machine used to execute them. Section 4 describes the main traits of the implementation in technical terms, along with a performance evaluation for the Android service. Section 5 describes our two case-study experiments. Section 6 discusses related work, and Section 7 concludes the article with a discussion of ongoing work and future research topics.

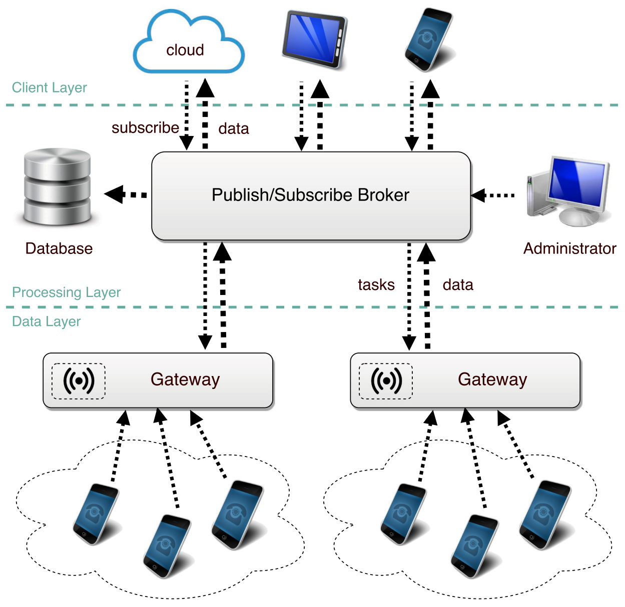


Fig. 2. The Flux architecture.

## 2 FLUX OVERVIEW

### 2.1 Requirements

The use of mobile devices for sensing purposes is subject to a number of constraints when compared to static sensing infrastructures such as WSN. First, there is no guarantee that a mobile device always has an active data connection. Unlike in WSN this may not be due to the faulty nature of wireless connections but rather because the device's owner may impose radio silence. Users can also, without warning, kill the sensing software running on the device. More often, however, this software must share with many other applications the resources of the mobile device, namely processor time and storage. The issue of safety, in the sense that the execution of the sensing software will not disrupt the normal functioning of the device, is crucial as otherwise users will not participate in sensing activities. Finally, the definition of regions and the distribution of the corresponding sensing tasks must be totally transparent from the device's owner perspective, to minimize the impact on the usability of the device.

With these concerns in mind, we considered the following five general requirements for the Flux framework: (a) it must be able to disseminate sensing tasks to be performed by mobile devices and aggregate the captured data to be forward to interested clients; (b) it must allow the specification of regions and their associated task pools and to automatically download and activate the tasks once the mobile devices enter the regions; (c) it must operate in the background without requiring user intervention; (d) it must be able to run tasks in any type of mobile device as long as the sensor requirements are met, and; (e) it must have a low execution footprint in terms of resource consumption, e.g., processor time, storage, and battery.

### 2.2 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

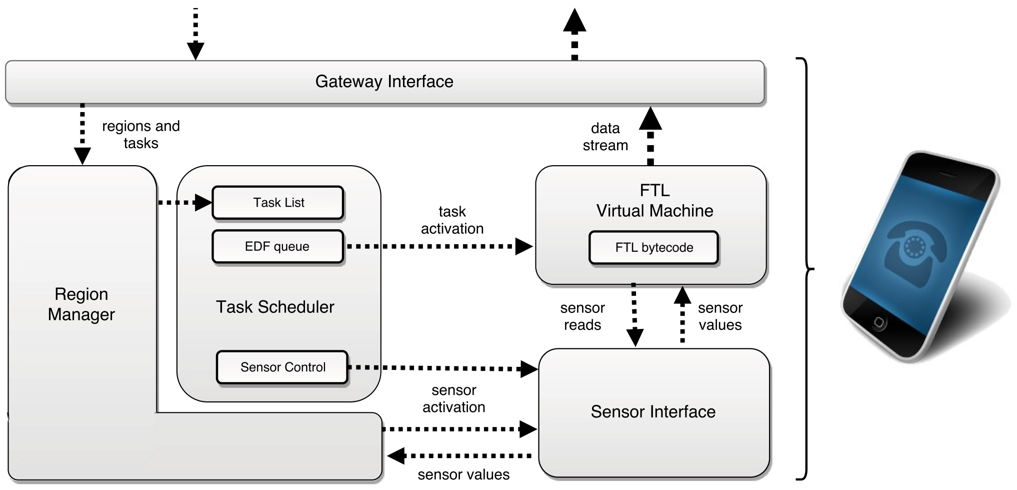


Fig. 3. The Flux Android Service.

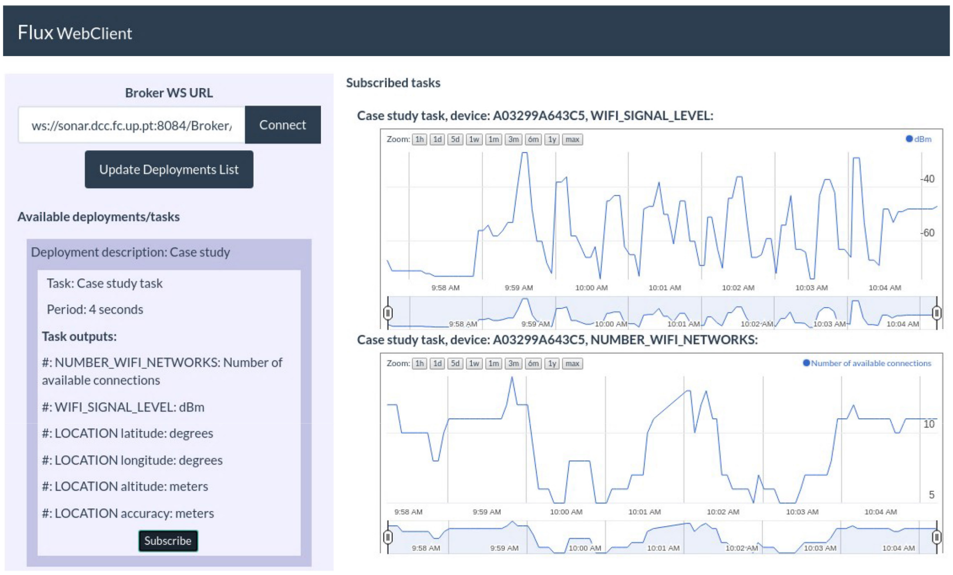
sent from mobile devices in a region to gateway devices that act as proxies of the broker. Figure 2 shows a high-level representation of these three layers and their relations.

2.2.1 *Data Layer.* A Flux gateway is a proxy for the publish/subscribe broker that forwards data streams from the devices to the broker and region and task operations (e.g., install, remove, updates) from the broker to the devices.

Sensing tasks are executed by the Android Flux Service running on the mobile devices, as illustrated in Figure 3. Through its gateway interface, the service discovers gateways and registers itself. Over time, it receives commands from the gateways to be performed locally, e.g., install a region and its task pool. The region manager periodically checks whether the device is within one of the stored regions and, if so, adds the corresponding tasks to an Earliest-Deadline First (EDF) queue for execution. The tasks are executed according to their periodicity in a virtual machine that makes use of the sensor interface to get sensor readings. The data is sent back to the gateways via the gateway interface.

Regions are specified using a domain-specific programming language, the Flux Task Language (FTL), described in Section 3. FTL extends a language originally designed for WSN devices (Ferro et al. 2015), by allowing regions to be specified in addition to corresponding sensing tasks. Thus, a subset of the language concerns region specification, and a different subset is used to program the actual sensing tasks. FTL is a statically typed language that is parametric in the set of sensors available in each device, abstracting away from hardware and low-level operating system details, whilst also providing a number of runtime safety properties. FTL source code for tasks is compiled to machine-independent bytecode that can in turn be scheduled for periodic execution using the EDF scheme mentioned above. The bytecode is executed by the FTL virtual machine with very low memory and processor time footprint, as illustrated by the evaluation later given in Section 4.

2.2.2 *Processing Layer.* The processing layer is composed of a publish/subscribe broker that keeps track of a set of gateways, contributing with data streams, and a set 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). The broker also maintains a database where data streams are logged for a (configurable) time window, making it possible for clients to access streams that were captured in the recent past rather than live-streamed. Finally, the broker provides an


Flux WebClient

Broker WS URL

ws://sonar.dcc.fc.up.pt:8084/Broker/

Available deployments/tasks

Deployment description: Case study

Task: Case study task

Period: 4 seconds

Task outputs:

#: NUMBER_WIFI_NETWORKS: Number of available connections

#: WIFI_SIGNAL_LEVEL: dBm

#: LOCATION latitude: degrees

#: LOCATION longitude: degrees

#: LOCATION altitude: meters

#: LOCATION accuracy: meters

Subscribed tasks

Case study task, device: A03299A643C5, WIFI_SIGNAL_LEVEL:

Case study task, device: A03299A643C5, NUMBER_WIFI_NETWORKS:

Fig. 4. A Flux client showing live data streaming on a web browser.

administration interface through which programmers can provide new region specifications and associated task pools, which it then injects in the gateways.

*2.2.3 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 4, where live or logged data streams can be visualised. A command-line tool is also available that can readily be composed in Unix style (e.g., using pipes) with arbitrary data-processing scripts, e.g., to preprocess and forward the data to a cloud service.

## 3 REGION PROGRAMMING

As mentioned earlier, Flux uses FTL, a domain specific programming language, to specify regions and their associated task pools. The FTL compiler takes a region specification and produces a binary representation that is suitable for installation in devices. The representation includes the region's constraints and the bytecode for all associated tasks. After a region is setup in a device, the Flux Android service takes care of monitoring its activation, and (when active) executing the bytecode of the region's tasks in safe manner.

### 3.1 FTL Regions

A region is defined as a subset of the $k$-dimensional attribute space associated with $n$ sensors required to define it. Notice that $k \geq n$ as some sensors, e.g., the accelerometer or the GPS, have multiple outputs. The subset is defined as a conjunction of constraints on the outputs of the sensors. If all the constraints hold, then the device is said to be *in the region*. Thus, to define a region, we need to specify a set of required sensors that must be available in the devices participating in the crowd-sensing activity, and we also need to specify the set of constraints that define the region boundaries. Last but not least, we must define the set of tasks associated with the region. The execution of these tasks is triggered once a device enters the region, and disabled when the device

```
region FC6_Building {
  sensors {
    NUMBER_WIFI_NETWORKS: void -> int,
    WIFI_SIGNAL_LEVEL: void -> int,
    LOCATION: int -> float,
    BATTERY_LEVEL: void -> int
  }
  constraints {
    LOCATION(0) in [-8.640235302792531, -8.641549110412598],
    LOCATION(1) in [+41.151954650878906, +41.15246725673063],
    BATTERY_LEVEL > 25
  }
  gateways {
    gw1.flux.dcc.fc.up.pt,
    gw2.flux.dcc.fc.up.pt
  }
  tasks {
    WiFiCoverage {
      source = "WiFiCover.ftl",
      period = 4000,
      description = "WiFi Coverage Sensing"
    }
    EnvNoiseLevel {
      source = "EnvNoiseLevel.ftl",
      period = 1000,
      description = "Environmental Noise Sensing"
    }
  }
}
```

Fig. 5. A specification for a region.

leaves it. A task in a region is defined by the location of the corresponding source code file as well as other pertinent attributes such as its period in milliseconds and a short textual description.

Figure 5 shows an example of a specification for a region in our faculty department (FC6_Building). The **sensors** block discriminates the sensors required by the devices to participate in the crowd-sensing activity and the **constraints** block defines the region boundaries. In the example, all devices within a geographical rectangle defined by the coordinates shown and with a battery level higher than 25%, will run the tasks. The hostnames of 2 FLUX gateways that will disseminate the region specification are identified next in the **gateways** block; after receiving the region's specification, the FLUX broker will push it to the identified gateways. Then, in the **tasks** block, two tasks are specified that periodically measure: (1) the available WiFi coverage every 4 seconds, i.e., the number of available WiFi networks at each device location and the maximum signal strength, and; (2) the noise level in the region with a period of 1s.

## 3.2 FTL Tasks

Tasks associated with a given region are programmed using another subset of FTL. The code for FTL tasks code is compiled into very compact bytecode sequences run by the virtual machine embedded in the FLUX Android Service, at each device. Both FTL and the virtual machine were designed to minimize the processing and memory footprint as well as to provide assurances of runtime safety.

```
stream {
  int   : "NUMBER_WIFI_NETWORKS: #nets",
  int   : "WIFI_SIGNAL_LEVEL: dBm",
  float : "LOCATION latitude: degrees",
  float : "LOCATION longitude: degrees",
  float : "LOCATION altitude: meters",
  float : "LOCATION accuracy: meters"
}

requires {
  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;
}


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];
  }
}
```

Fig. 6. FTL source code the WiFi coverage task.

*3.2.1 Source Code.* Figure 6 shows the FTL source code for the WiFiCoverage task specified for the FC6_Building region in Figure 5. The code starts with the type of the stream generated by the task (only one type of stream per task is allowed), the **stream** block. Each item in the stream is described in terms of component fields, their type, label, and some textual information. The **requires** section contains a description of the sensors required 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 every time the task is activated; recall that the activation period itself is not defined by the task's code, but instead configured in the region specification, as discussed earlier. The **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; the accuracy value corresponds to the horizontal dilution of precision (HDOP) reported for each GPS reading. Beyond the constructs shown in the example, FTL also provides support for other common (e.g., arithmetic, relational, logical) operators over 32-bit integer and floating-point scalar values.

*3.2.2 The Virtual Machine.* The listing shown in Figure 7 depicts the bytecode for the FTL task given in Figure 6 in human-readable form; the actual binary representation is just 137 bytes. The execution of bytecode follows the familiar scheme of a stack-based VM, i.e., each operation pops operands from a stack and/or pushes its results onto to the stack.

As shown in the figure, the bytecode 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

```
.total   137
.offset  19

.data
.0   0
.4   0.0
.8   1
.12  2
.16  3
.20  10
.24  number_wifi
.28  wifi_level
.32  latitude
.36  longitude
.40  altitude
.44  accuracy

.stack
.48  0
.52  0
.56  0
.60  0
.64  0
.68  0
.72  0
```

```
.text
.0   rd  NUMBER_WIFI_NETWORKS  0
.3   st  number_wifi
.5   rd  WIFI_SIGNAL_LEVEL  0
.8   st  wifi_level
.10  ld  #0
.12  rd  LOCATION  1
.15  st  latitude
.17  ld  #1
.19  rd  LOCATION  1
.22  st  longitude
.24  ld  #2
.26  rd  LOCATION  1
.29  st  altitude
.31  ld  #3
.33  rd  LOCATION  1
.36  st  accuracy
.38  ld  accuracy
.40  ld  #10
.42  f2i
.43  fle
.44  bf  60
.46  ld  number_wifi
.48  ld  wifi_level
.50  ld  latitude
.52  ld  longitude
.54  ld  altitude
.56  ld  accuracy
.58  rad  6
.60  ret
```

Fig. 7. Bytecode for the WiFi coverage task.

flow control logic. For instance, the bf 60 instruction at address 44 is a conditional branch to the instruction at address 60, where ret (the "return" instruction) 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.

#### 3.2.3 *Runtime Safety Properties.*

By construction, FTL is quite constrained 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, 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 stack (discussed previously).

The use of the FTL language and VM also defines a secure sand-box, as opposed to a scheme where arbitrary binary code may be downloaded and used for data sensing tasks, raising well-known security issues that are complex to detect and mitigate; e.g., see Arzt et al. (2014), Das et al. (2010), and Enck et al. (2011).

## 4 IMPLEMENTATION

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

### 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,[1] the Flux service was programmed on Android Studio[2] to run on Android 4.4 or higher, and the FTL compiler was implemented using the ANTLR compiler infrastructure.[3] All components of the Flux architecture communicate using a common message format, specified using Google's Protocol Buffers.[4] Plain TCP/IP sockets were used for gateway-device communications and Web-sockets for interactions of the broker with gateways, administration module, and clients. For logging data streams at the broker, we used an SQLite database.[5]

### 4.2 The Flux Broker

*4.2.1 Region Administration and Dissemination.* The Flux broker maintains a list of all the region specifications uploaded to the broker by authenticated users using the administrative interface. Users upload a region specification plus the source code of FTL tasks used by it. The source code of tasks are compiled by the broker into Flux virtual machine bytecode, and then packaged together with the region specification for delivery to the gateways specified by it (cf. the **gateways** block, Section 3.1).

For each sensing task in a region, the broker also considers the meta-data information for the task's data stream (cf. the **stream** block, Section 3.2) to create a corresponding SQLite database table. The table will store values received for the data stream, hence it will have the same fields as defined for the stream, along with extra information identifying data collection timestamps plus the originating devices and gateways.

*4.2.2 Client Interface.* Clients interface with the broker through a publish/subscribe service. Clients specify the data streams they are interested in from a list of available data streams provided by the broker. Subscriptions may optionally be bounded by a time window, possibly in the past. In a live stream setting, data received from gateways is published (forwarded) immediately to clients who subscribed to it. Archived data streams in the broker's database are also supplied to clients if the data for the specified time window is available.

### 4.3 Flux Gateways

*4.3.1 Gateway Discovery and Region Dissemination.* Each gateway manages a set of regions, previously uploaded by the broker, that will be injected or updated in mobile devices that connect to it. The gateways advertise themselves over wireless networks, so the Flux Android Service does not need any prior knowledge of the location or IP address of any gateway, i.e., it just probes the network for possible gateway advertisements. When a mobile device connects to the gateway, the

[1] http://tomcat.apache.org.
[2] https://developer.android.com/studio/.
[3] http://www.antlr.org.
[4] https://developers.google.com/protocol-buffers.
[5] http://sqlite.org.

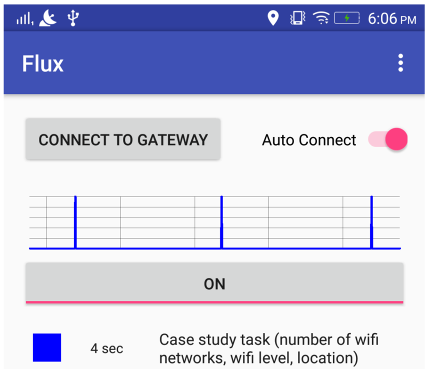

6:06 PM
Flux
CONNECT TO GATEWAY
Auto Connect
ON
4 sec
Case study task (number of wifi networks, wifi level, location)

Fig. 8. The Flux service Android application.

latter compares the registered pool of regions to the ones running on the device and sends the updates to the device. This synchronization also takes into account whether a given device meets all the requirements for running a region's tasks, i.e., if it has all the necessary sensors (cf. the **sensors** block in region specification described in Section 3.1). If the sensor requirements are not fully met by a device, then it does not download the region.

*4.3.2 Data Stream Handling.* When a task is activated in a device, i.e., the device enters the task's governing region, the corresponding data stream is uploaded to the current gateway the device is engaged with. Data stream chunks, possibly aggregated from various tasks, may be uploaded with a configurable period or continuously, as described later in the text. If the connection is temporarily lost, then data can be received later by the gateway when the connection is resumed. It is possible that connection is resumed with a different gateway, but any data from buffered data streams is forwarded in any case (even if the region is not currently programmed or has been removed in the meanwhile) so it still eventually reaches the broker.

## 4.4 Flux Android Service

The implementation of the Android service follows the organisation earlier depicted in Figure 3. It is composed of five main modules: the gateway interface, the region manager, the task scheduler, the FTL virtual machine, and a sensor interface. As a service, it runs in the background without need for user interaction. A user-interface application, shown in Figure 8, 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.

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

*4.4.1 Region Manager.* The region manager monitors the constraints associated to the regions presently configured on the device. When the constraints associated with a region are satisfied, the corresponding tasks are passed on to the scheduler for execution. The evaluation of such constraints uses values provided by the sensor interface, which is notified of which sensors are required for evaluating the constraints when a new region is downloaded to the device. This control of the active regions is done periodically, according to a global configuration parameter with a default value of 1min. A region is kept in the device so long as it does not stay inactive for too long, currently 24h. Likewise, it can be removed when a gateway command is received to disable it.

*4.4.2 Sensor Interface.* The sensor interface is responsible for obtaining sensor readings, interacting with assorted Android OS APIs for that purpose. In addition to the requests from the region manager, the task scheduler also uses the sensor interface to enable sensors on/off as tasks are scheduled for execution, so the virtual machine can obtain the sensor readings at runtime. The scheduler implements an adaptive activation/de-activation strategy for sensors. Active sensors may consume significant battery power (e.g., GPS), whilst their repeated initialisation/shutdown may cause unnecessary latency. In particular, initialisation may imply high latency until valid readings are obtained (e.g., again 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 empirical) balance between initialisation latency and battery consumption (e.g., the current value is set to 2 minutes for GPS).

*4.4.3 Gateway Interface.* Regarding the gateway interface, the Android service 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.

## 4.5 Performance Evaluation

We conducted an evaluation of the Android service in terms of resource consumption and virtual machine performance during bytecode execution. For the evaluation, we used a Google Nexus tablet running Android 6.0 with 2GB of RAM and a dual-core 2.3GHz CPU, plus a gateway installed on a 4-core machine with 12GB 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 experiments 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 1Hz, (2) an atmospheric pressure sensing task at 2Hz, (3) a gyroscope sensing task at 4Hz, and (4) an accelerometer sensing task at 8Hz.

For each configuration, we then conducted five monitoring sessions of a 2min run of the service using the Android Debug Shell (adb). In terms of resource consumption, we sampled the CPU utilisation and RAM usage in 1s intervals, plus the total of the TCP/IP data transmitted by the service in each 2min interval. We also measured the power consumption by sampling the remaining energy on the device battery (watt-hours) before and after each configuration, through the use of

Table 1. Resource Consumption

| Tasks | CPU (%) | RAM (KB) | Net. (bytes/s) | Battery (J/min) |
|---|---|---|---|---|
| None (∅) | 0.17 ± 0.04 | 9731 ± 2.8 | 6.2 ± 1.1 | 12.8 ± 0.6 |
| WS | 0.25 ± 0.06 | 9851 ± 3.3 | 86.3 ± 15.8 | 31.7 ± 2.8 |
| WS + AP | 0.32 ± 0.06 | 9864 ± 3.3 | 139.0 ± 25.0 | 34.2 ± 3.3 |
| WS + AP + GS | 0.58 ± 0.08 | 9877 ± 3.8 | 288.9 ± 52.4 | 35.9 ± 2.9 |
| WS + AP + GS + AS | 1.39 ± 0.10 | 9915 ± 5.6 | 576.6 ± 104.0 | 36.5 ± 3.6 |

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

the Android `BatteryManager` API, converted to the scales of Joules. To withdraw the total energy consumed by the device, we calculated the difference between the sampled values. The evaluation was performed with the device screen turned off, seeking to minimize the power consumption unrelated to the Flux Android service, but with WiFi turned on to measure the consumed network bandwidth. 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, the average network bandwidth used to send the sensed data and the power consumption measured in Joules per minute.

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 10MB, and the consumed network bandwidth is less than 1KB/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.

Regarding battery consumption, we can observe that the service consumes little battery. When no tasks are running, the average consumption observed was 12.7J/min. This value is significantly increased to 31.7J/min for the second configuration: the WiFi survey task activates the GPS sensor, that is the most power-hungry of all the sensors used on this evaluation. In the remaining configurations, even though we introduced tasks with higher frequencies, the increment in the power consumption was small, namely, because the remaining sensors are more constraint in energy consumption. Overall, the peak value of battery consumption is close to 40 J/min. To put this in perspective, given that the devices at stake have a nominal battery capacity of 96, 480J, assuming an average voltage of 4V, 40J per minute represent a battery consumption of roughly 2.5% of the battery capacity per hour. This is not to say that, in proportion, the Android service should withstand about 40h of operation non-stop. Power consumption levels do not generally evolve linearly, depend on several factors, and are reasonably complex to reason on (Tarkoma et al. 2014). In addition, note that our evaluation considers the device in stand-by mode with the screen turned off, mitigating interference from apps and services that are active (or more active) during normal use of a device.

In addition to resource consumption, we also measured the performance of bytecode 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 5ms on average to run. All other tasks run in less than 0.5ms, on average.

Table 2. Bytecode 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 |

## 5 CASE STUDIES

In this section, we present two case-studies that aim to demonstrate: (a) the dynamic injection and activation of tasks in mobile devices moving around in a given region, with the acquisition of the corresponding data streams, and; (b) the activation/de-activation of tasks as devices roam between regions. For each experiment, we also present a synthetic analysis of the data streams gathered during the experiments.

### 5.1 WiFi Coverage Case-Study

*5.1.1 Outline.* For evaluating the base functionality of Flux, we conducted a controlled real-world experiment where WiFi service quality was surveyed over a certain area. For this, we programmed Flux using a single region constrained to the geographical zone of interest, and a single sensing task. Volunteer users carried Android devices and walked through prescribed paths along the survey area, while the Flux Android service executed the sensing task to collect GPS-referenced WiFi signal data and streamed that data to a Flux gateway. The experiment was also useful to validate the Android service across a heterogeneous set of devices in terms of device types (smartphones or tablets), manufacturers, and Android versions.

*5.1.2 Setup.* The survey area, depicted in Figure 9, has a dimension of roughly 100 × 150m, and comprises the Computer Science department 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.[6] 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[7] instalment at our university, the most commonly used campus network by students and staff. For sensing, we defined a single region geographically constrained to the zone of interest and with one sensing task associated to it. The task is the same as described earlier in Section 3, with the single difference that no accuracy filter is set when transmitting to the gateway (i.e., the if guard condition in Figure 6 is omitted), and was configured to run with a periodicity of 4s. Thus, it is programmed to collected two items of information over time and space: the WiFi signal strength, and the number of nearby networks also

[6]The satellite and map imagery used in this article 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).
[7]http://eduroam.org.

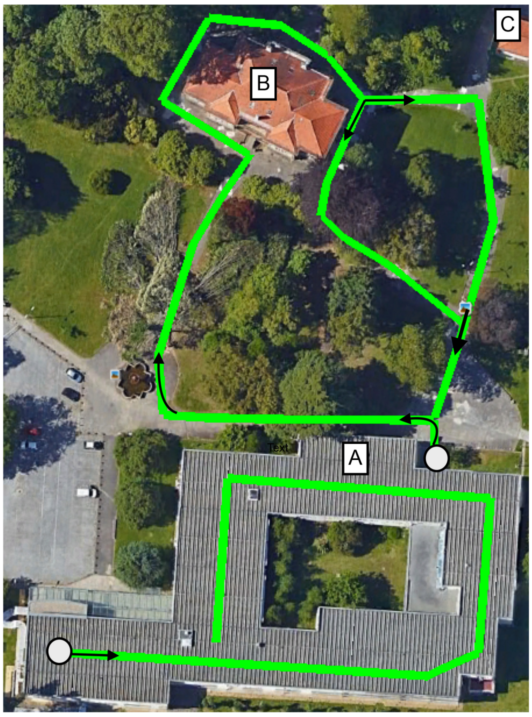


Fig. 9. Survey area for WiFi coverage analysis.

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.

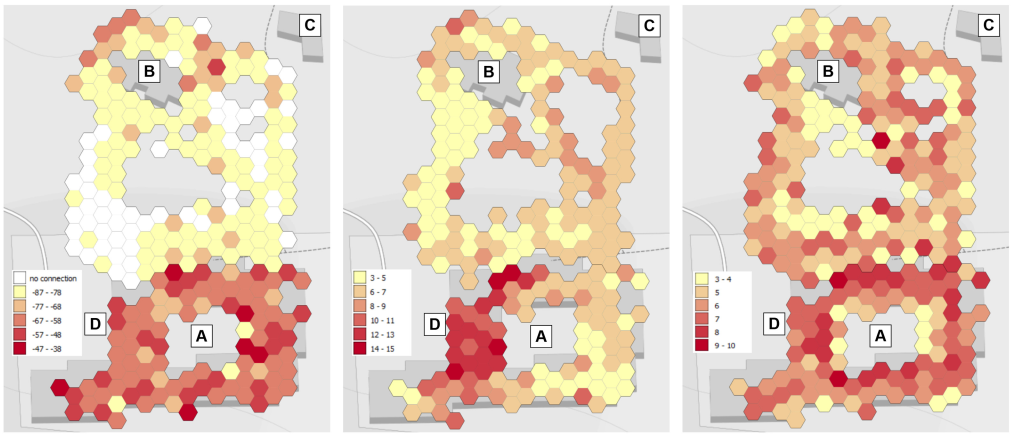
In terms of the cloud setup, we used a CentOS Linux virtual machine (CentOS VM) with two cores and 1,837MB of RAM, hosted on a OpenStack cloud infrastructure. An Apache Tomcat application server instance runs on the VM, hosting a Flux gateway and a Flux P/S broker. The CentOS VM is accessible over the 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 article, multiple gateways running on different hosts can be used, interacting with a broker on another, possibly distinct, host.

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.

*5.1.3 Results.* After setup, the volunteers conducted 33 trips along the prescribed survey paths, resulting in the collection of 2,726 data sample measurements, 1,193 inside the department building and 1,533 outside. For data analysis, we filtered out measurements for which the GPS accuracy exceed 10 meters, reducing our data set to 1,922 samples (69% of the original) inside the building and to 1,212 samples (79% of the original) outside. Figure 10 depicts the filtered data set as

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) |



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

Fig. 10. WiFi coverage survey—data plots.

geo-referenced "heat maps", in terms of the eduroam WiFi signal strength (Figure 10(a)), the number of detected WiFi networks (Figure 10(b)), and the GPS HDOP (Figure 10(c)). In the plots, rendered using QGis,[8] 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 9).

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).

Regarding interference between eduroam and other networks, we can observe areas inside the building 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 10(b). 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 11 relating the WiFi signal and the number of networks; no correlation pattern emerges, as illustrated by the

[8]http://www.qgis.org.

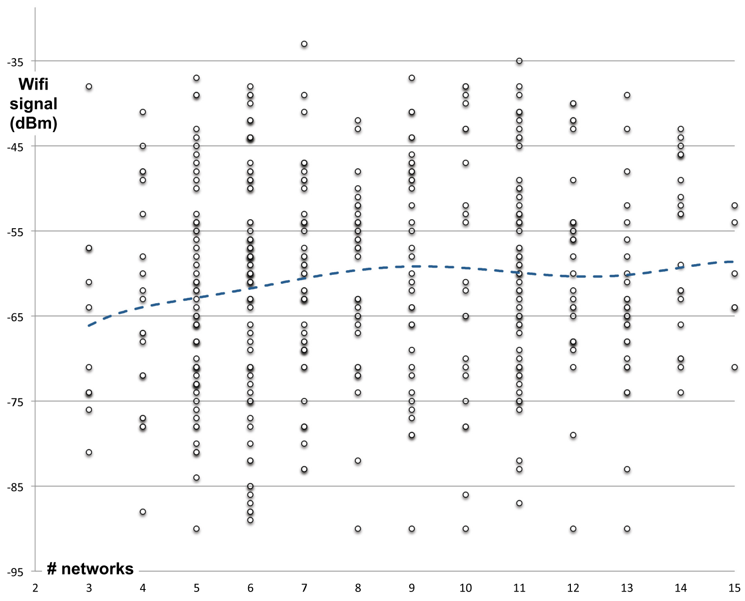


Fig. 11. WiFi signal vs. number of WiFi networks inside the department building.

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.

## 5.2 Region Roaming Case-study

*5.2.1 Outline.* For evaluating the activation and deactivation of regions, we considered roughly the same area as for the WiFi coverage case-study as a target for sensing, but defined three FLUX regions over it, and distinct sensing tasks for each region: (1) in an indoor region, we measured audio noise levels, (2) stop-points for users outdoors, and (3) temperature in a region that enclosed the first two. As before, volunteer users carried Android devices and walked through prescribed paths along the survey area, triggering FLUX to activate different regions/tasks over time.

*5.2.2 Setup.* The survey area and FLUX regions are depicted in Figure 12, along with an outline of the paths followed by volunteer users. As shown, the survey area again comprises our department building and part of the surrounding gardens. The three regions, marked A to C, were defined with different geographical constraints and geo-referenced sensing tasks:

- Region A encompasses the entire survey area. For this region we programmed a task measuring internal battery temperature with a period of 2s. The devices we had at hand for testing (like most Android devices currently in the market) did not have a built-in ambient temperature sensor, which would be a better choice for environmental monitoring. However, internal temperature correlates and normally flows in line with variations in ambient temperature, which makes the former an interesting proxy measure for the latter (Overeem et al. 2013).
- Region B covers part of the garden area. For this region, we programmed a movement detection task, using the step detector "meta-sensor" provided by the Android API, with a

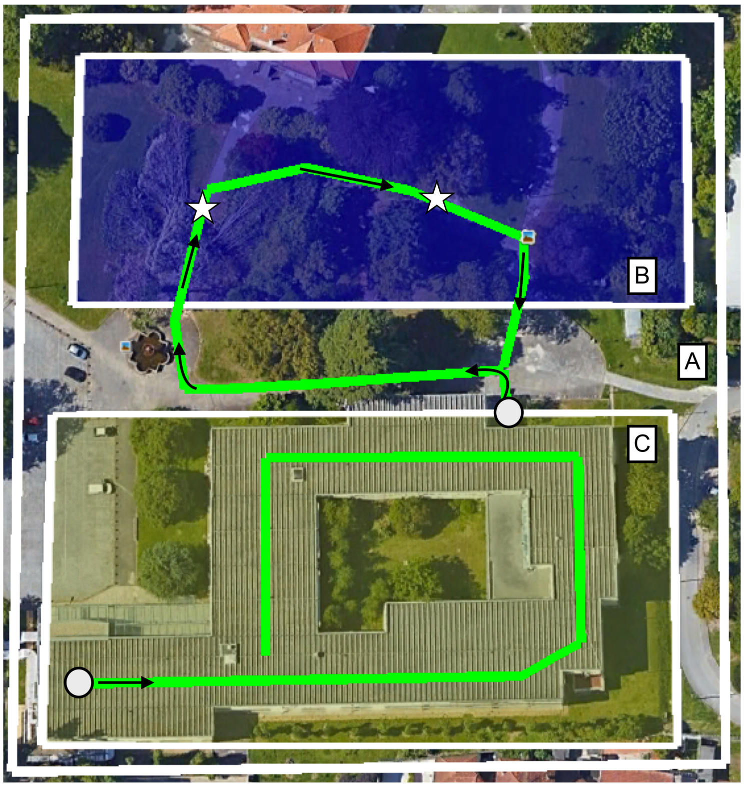


Fig. 12. Multi-region survey area.

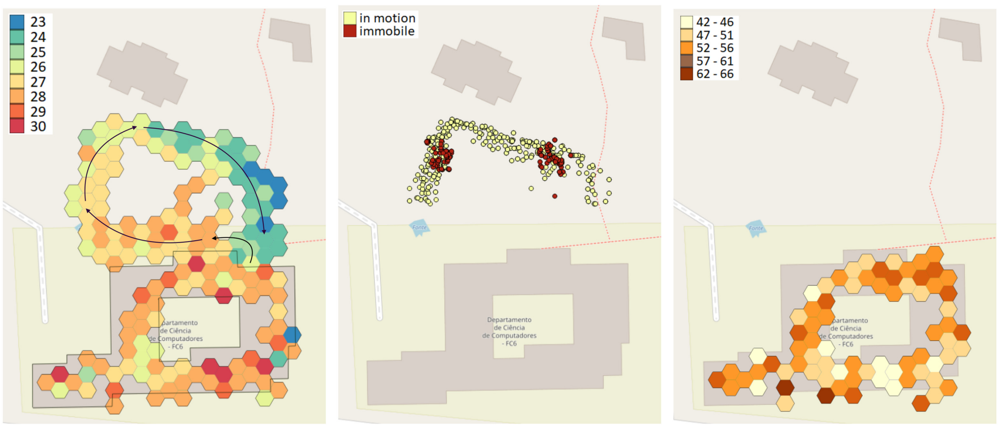
period of 1 second. To impose stopping points, we instructed volunteers to stop near two prescribed points in the garden walkways, marked as stars in Figure 12, for approximately 10s.

- Region C corresponds to the department building. For this region we programmed an audio noise sensing task using built-in device microphones, also with a period of 1s. The aim was to get a sense of possible noise variations in different parts of the building, e.g., possibly higher levels of noise near classrooms, and less so near offices.

The hardware setup for the broker and gateway were the same as for the WiFi coverage case-study. As for mobile devices, we employed six of the Google Nexus tablets also used in the previous experiment. We decided not to use heterogeneous devices, given that internal temperature and audio measurements may vary widely for distinct devices.

The volunteers walked several times along the prescribed survey paths over a period of two hours. As in the WiFi survey experiment, we filtered out measurements for which the GPS accuracy exceeded 10m, obtaining 3,161 data sample measurements, 1,031 of which inside the department building and the remaining 2,130 outside. Figure 13 depicts this data set as geo-referenced "heat maps" for temperature measurements (Figure 13(a)) and audio noise levels (Figure 13(c)) representing average values in regions A and C, and as a scatter plot for stop-point/movement data (Figure 13(b)) in region B.

*5.2.3 Results.* Regarding the temperature data in Figure 13(a), the plots show an overall variation of internal battery temperature between 20 and 30 Celsius degrees. Higher temperatures are



(a) Battery temperature (celsius). (b) Stop-point/movement. (c) Audio noise levels (dB).

Fig. 13. Multi-region survey—data plots.

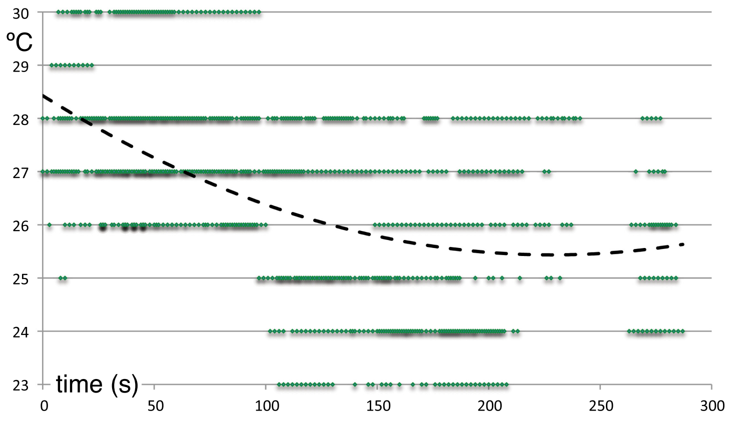


Fig. 14. Temperature measurements outdoors over time.

more frequently observed inside the building, which has air conditioning, and colder temperatures outdoors (ambient temperature was about 10 degrees). Observe that for outdoors data, if we account for the direction of survey paths (sketched in the figure, and originally depicted in Figure 12), the overall trend is that battery temperature decreases with progressive exposure to open air. This is best illustrated in the scatter plot of Figure 14, where we depict the temperature outside as a function of walking time, and a trend line that clearly indicates that temperature decreases over time.

Regarding the stop-points/movement analysis in Figure 13(b), the scatter plot clearly indicates the absence of movement near the two prescribed stop-points, previously identified by star symbols in Figure 12.

Finally, for audio noise level measurements, the data captured by the task was the maximum absolute amplitude measured in recordings of 1s (the task's period), as indicated by the absolute reading (an unsigned 16-bit value) from the uncalibrated microphone sensor the devices we used. The dB values are estimates from these amplitude values, assuming a 30dB value for an empty office room with air conditioning turned on. We can observe in Figure 13(c) that there are roughly two identifiable zones in the building. The south area of the building is more silent, with estimated noise predominantly lower than 51dB, coinciding with the zone where staff offices can be found, while the remaining areas are louder, with estimated noise usually between 52 and 62dB, coinciding with rooms used for classes and by the student union.

## 6 RELATED WORK

### 6.1 MCS Platforms

We now survey some state-of-the-art MCS platforms and then make an overall comparison with Flux.

The Medusa (Ra et al. 2012) platform is designed for general purpose crowd-sensing using Android smartphones, in interface with Amazon Mechanical Turk (AMT) to recruit workers for sensing tasks and manage monetary incentives. Each task is defined in MedScript, an XML-based domain-specific language, as a sequence of stages. Stages may define human actions (e.g., taking a photo) or passive sensing, and are supported by binary modules that are downloaded (once) when needed. For security, these binary modules are verified by static Java bytecode analysis that detects the use of disallowed APIs. Medusa is not reconfigurable in the sense of Flux regions, as tasks and stages do not have associated constraints, instead the dynamics of the system are driven by the AMT crowd-sourcing model. Like FTL bytecode, the use of MedScript makes for compact task specifications, which the authors found to several orders of magnitude small to equivalent binary modules in a few examples.

Sensus (Xiong et al. 2016) is an Android and iOS crowd-sensing application oriented toward human studies surveys. Human studies surveys can be scheduled or sensor-triggered, and also integrate sensor data in survey responses, including internal and external smartphone sensors like Bluetooth LE beacons or smart-watch sensors. Sensus is self-contained in the sense that the same mobile application is used both for programming surveys, taking them, and storing the results. The Amazon S3 cloud service can also be used for storage, and an R toolkit library (SensusR) allows for processing, analysis, and visualisation of aggregated data. Tasks, called sensing protocols, are specified using the app stored on a cloud server, and disseminated using email, URLs, or other means. Sensing protocols are expressed in a domain-specific JSON-based language. As in Flux, constraints may be specified for sensing, e.g., geographical locations for triggering user surveys.

In PRISM (Das et al. 2010), implemented for Windows Mobile smartphones, sensing tasks are disseminated from a central server using a push model that implements a two-level predicate characterisation for task installation, overall in line with the aims of Flux regions. A high-level predicate is defined by a set of coarse-grained geographical regions, the required devices in each region, and the required sensors for a task. A low-level predicate, evaluated per device, may be expressed accounting for more fine-grained conditions for task activations, such as precise locations and speed or temperature ranges. Tasks take form as executable binaries that may be untrusted, raising concerns of safety and security. To cope with these, tasks run in sandboxed environment where all system calls are relayed through the PRISM daemon through a system call interposition scheme. The daemon uses access control and dynamic taint analysis mechanisms to regulate sensor access and obtained data, and monitors execution to terminate tasks that cause resource depletion (e.g., CPU and memory), or to enforce limit for resource usage in other cases (e.g., sensor and file system access).

AnonySense (Cornelius et al. 2008), a platform for Linux-based PDAs and iPhones, allows applications to submit sensing tasks to mobile devices to obtain sensed data reports in return. A task is programmed using a declarative language, AnonyTL, that specifies acceptance conditions based on device characteristics (e.g., sensor payload), data report statements, and an expiration time for sensing. Report statements express the desired sensor data, along with conditions for reporting them, expressed, for instance, in terms of a geographical area, periodicity, or time of day. Applications inject tasks and receive back data in interface with the AnonySense server, organised as a set of services for device registration, task dissemination, and data streaming. Security, in general, and anonymization, in particular, are core concerns of the system, by employing the use of mix networks, trust models, and peer certification.

USense (Agarwal et al. 2013) is an Android smartphone crowd-sensing platform for environmental sensing. Tasks, called sensing moments, are programmed using an XML-based domain-specific language. They have a periodic or event-based nature to monitor specified sensor values, and can be configured with sensing constraints such as geographical location and time. Additionally, an utility value from 0 to 1 denotes the urgency or importance of the task. A significant feature of the runtime system is the use of adaptive sampling strategies, enabling and disabling sensors on-the-fly, that factor in event detection and task utility values that seek to minimize energy consumption. As discussed in Section 4, the current version of Flux, in spite of having low-footprint, follows very simple heuristics in this regard.

SARANA (Hari et al. 2008) 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, in particular device location may be accounted for when assigning tasks. SARANA makes use of a domain-specific language, a superset of Java, that provides macro-programming abstractions for device discovery, task distribution under spatial, temporal, and resource constraints, and processing of aggregated data.

Summarising the above discussion of MCS frameworks, Flux more closely relates with AnonySense, Medusa, Sensus, SARANA, USense, in the sense of using a domain-specific language for expressing sensing tasks and associated execution environment. We believe these approaches are more principled to promote safety and security, as opposed to relatively more complex handling of these issues in arbitrary binary code as the PRISM platform is obliged to. Like Flux, AnonySense, Medusa, PRISM, Sensus, SARANA, and USense provide the necessary infrastructure to inject sensing tasks into the mobile devices. In the spirit of Flux regions, most platforms allow the specification of constraints to parameterise task dissemination, except Medusa, which relies on Amazon Mechanical Turk's crowd-sourcing engine for that purpose. Unlike Medusa and Sensus, but similar to AnonySense, PRISM, SARANA, and USense, Flux tasks run in the background gathering sensor data, without user participation in sensing.

To finish this survey, we additionally make a brief reference to MCS frameworks where the "human" is the sensor, more often called crowd-sourcing frameworks, since sensor data does not play a pivotal role, apart from typically GPS for geo-tagging purposes. Two examples are Zooniverse (Simpson et al. 2014) and Ushahidi (Okolloh 2009). Zooniverse is a quite successful citizen-science platform that allows users of mobile devices to contribute to scientific projects, typically by helping with the processing of large datasets with classification tasks. Ushahidi, in its cloud and mobile app form, is a general purpose system to aggregate data supplied by humans using disparate means (e.g., SMS, Twitter) and its use has been specially noteworthy for crisis management in disaster scenarios.

### 6.2 WSN Platforms

Flux comes in sequence to previous work on the SONAR (Ferro et al. 2015) and Callas (Lopes and Martins 2016) systems for programming static WSN deployments. As in Flux, SONAR's architecture is organised in terms of client, broker, and data layers. SONAR, however, does not support the notion of regions. FTL is an extension of the SONAR Task Language (STL) with support for the specification of regions, and the virtual machine that runs FTL bytecode for sensing tasks is essentially a Java/Android port of the STL virtual machine originally written in C. Similar to Flux and SONAR, Callas used a domain-specific language and an associated virtual machine environment. Like SONAR, Callas does not support regions. The Callas language is quite different in nature from STL and FTL, as it takes a process-calculi approach to define running modules in a WSN. STL and FTL are simpler, as their expressiveness is limited to simple periodic sensing tasks that run in logical isolation without any explicit concurrency constructs.

The concept of regions in Flux is similar to attribute-based regions proposed for WSN in Welsh and Mainland (2004), where a region can be defined based on several criteria, such as device location, radio connectivity, or values for sensed data. WSN languages like Regiment (Newton and Welsh 2004), for example, use regions in combination with data streams as fundamental programming abstractions, but as a functional macro-programming language, Regiment does not support the dynamical reconfiguration of the tasks, since the sensor network needs to be reprogrammed as a whole. Agilla (Fok et al. 2009), however, implements a mobile-agent programming flavour for WSN, where each agent (a task) can proactively migrate across the network, executing code that depends on the conditions sensed at each node. Agilla differs from Flux in that nodes are typically geographically fixed, whereas tasks may migrate across devices.

The driving goals for the use of regions in the WSN setting are to enable online data preprocessing or aggregation to reduce the bandwidth and energy required to send the streams to the base stations. In the mobile setting, however, bandwidth and energy limitations are much less strict, but device mobility and churn are important factors to account for, unlike in WSN that are typically composed of a fixed set of static nodes.

## 6.3 MCS Applications

There are several crowd-sensing applications powered by the use of personal devices. We highlight a few key areas of interest and related works.

For environmental sensing, crowd-sensing smartphone applications have been successfully used to collect data that serves monitoring, modelling or forecasting purposes. For instance, noise levels from a region for the purpose of generating accurate noise models as in NoizCrowd (Wisniewski et al. 2013) smartphones, sensor readings for barometric pressure can be used for weather monitoring and forecasting as in PressureNet (Mass and Madaus 2014) and WeatherSignal (Price and Shachaf 2017), custom sensors can be connected to smartphones to monitor air pollution as in GasMobile (Hasenfratz et al. 2012), and (as illustrated in our second case-study experiment) internal battery temperature can be used as a proxy value for ambient temperature (Overeem et al. 2013).

Another common application is monitoring urban infrastructure such as networks or roads. Large-scale coverage maps of cellular networks have been derived using OpenSignal (OpenSignal 2010) and Epitiro (Wakefield 2011), and other applications like Palz (Radu et al. 2013) can be used to monitor WiFi networks. Urban traffic can also through sensing or road incident reports from users, as in the works for Crotis (Roopa et al. 2013), SignalGuru (Koukoumidis et al. 2011), or SmartRoad (Hu et al. 2015), besides well-known commercial apps that engage millions of users like Waze or Google Traffic.

We finish by reporting on applications that monitor mobile device usage, as enablers for data analytics. The DeviceAnalyzer project (Wagner et al. 2013, 2014) is particularly noteworthy. The project's application has been used over the years to collect data for mobile phone characteristics and usage, building up a publicly available data set containing billions of records collected from thousands of devices. This data set used by the research community at large for several studies, e.g., as in Hintze et al. (2017) and Thomas et al. (2015). Similar initiatives to DeviceAnalyzer are Orange's D4D challenge (Blondel et al. 2012) and the Lausanne data collection campaign (Kiukkonen et al. 2010).

## 7 CONCLUSIONS

In this article, we presented the Flux framework for streaming sensor data from dynamically reprogrammable tasks injected into mobile devices. The fundamental concept is that of a region, a subset of the multi-dimensional "sensor space" (the cartesian product of the sets of sensor ranges). Regions are thus defined through constraints on the output of a set of sensors plus a set of associated sensing tasks. As devices move through the sensor space they alternatively enter and leave regions, activating and de-activating the corresponding task pools, respectively. Regions are specified using a domain-specific programming language called FTL. The subset used to implement tasks is sufficiently expressive for basic sensing operations and provides compile time guarantees of runtime safety. FTL source code is compiled into a compact bytecode that is in turn executed with a low-footprint virtual machine in the devices. We implemented a complete prototype of the framework and two case-studies that demonstrate: (a) the dynamic injection and activation of tasks in mobile devices moving around in a given region, with the acquisition of the corresponding data streams, and; (b) the activation/de-activation of tasks as devices enter/leave regions.

The key issues in the implementation of regions are, of course, related to the way their boundaries are detected and their impact on activation/de-activation semantics. There are issues related to sensor readings we wish to address as future work. For instance, missing a reading or getting an imprecise value may make it difficult to establish whether or not a device is still within a region's boundaries. Moreover, errors related to rapidly changing conditions, e.g., devices moving too fast between regions, may make it hard to enforce region activation constraints accurately. 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 are instead triggered explicitly by users or by the onset of certain environmental conditions in complement to (possibly too coarse-grained) region constraints.

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 minimalistic with scalar types, simple sensor and network interfaces, and 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 (Lahiri and Qadeer 2008) 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 (Rodrigues et al. 2017, 2018; Silva et al. 2017).

Finally, a few technical implementation aspects are worth future work, such as: extending the platform to accommodate for sensing actions with user intervention, e.g., as is normally required for crafted image acquisition or guided sensing steps; support for interface with external sensors, e.g., Bluetooth LE beacons, sensors embedded in smart-watches, or generic IoT sensors; and a more principled approach toward battery management and its relation with sensor activation/deactivation or data uploading through the network.

## REFERENCES

V. Agarwal, N. Banerjee, D. Chakraborty, and S. Mittal. 2013. USense—A smartphone middleware for community sensing. In *Proceedings of the MDM*, Vol. 1. IEEE, 56–65.

I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. 2002. A survey on sensor networks. *IEEE Commun. Mag.* 40, 8 (2002), 102–114.

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 *Proceedings of the PLDI*. ACM, 259–269.

V. D. Blondel, M. Esch, C. Chan, F. Clérot, P. Deville, E. Huens, F. Morlot, Z. Smoreda, and C. Ziemlicki. 2012. Data for development: The D4D challenge on mobile phone data. *arXiv preprint arXiv:1210.0137* (2012).

C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, and N. Triandopoulos. 2008. Anonysense: Privacy-aware people-centric sensing. In *Proceedings of the MobiSys.* ACM, 211–224.

T. Das, P. Mohan, V. N. Padmanabhan, R. Ramjee, and A. Sharma. 2010. PRISM: Platform for remote sensing using smartphones. In *Proceedings of the MobiSys.* ACM, 63–76.

W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. 2011. A study of Android application security. In *Proceedings of the SEC.* USENIX, 21–21.

G. Ferro, R. Silva, and L. Lopes. 2015. Toward out-of-the-box programming of wireless sensor-actuator networks. In *Proceedings of the CSE.* IEEE, 110–119.

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.

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 *Proceedings of the PLDI.* ACM, 1–11.

B. Guo, Z. Wang, Z. Yu, Y. Wang, N. Y. Yen, R. Huang, and X. Zhou. 2015. Mobile crowd sensing and computing: The review of an emerging human-powered sensing paradigm. *ACM Computi. Surveys* 48, 1 (2015), 1–31.

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. *Philos. Trans. Roy. Soc. A* 366 (2008), 3699–3708.

D. Hasenfratz, O. Saukh, S. Sturzenegger, and L. Thiele. 2012. Participatory air pollution monitoring using smartphones. In *Proceedings of the MobiSys.* ACM, 1–5.

D. Hintze, P. Hintze, R. D. Findling, and R. Mayrhofer. 2017. A large-scale, long-term analysis of mobile device usage characteristics. *Proceedings of the IMWUT* 1, 2 (2017), 13.

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 Trans. Sensor Netw.* (2015), 55:1–55:27.

N. Kiukkonen, J. Blom, O. Dousse, D. Gatica-Perez, and J. Laurila. 2010. Toward rich mobile phone datasets: Lausanne data collection campaign. *Proceedings of the ICPS.*

E. Koukoumidis, L. S. Peh, and M. R. Martonosi. 2011. SignalGuru: Leveraging mobile phones for collaborative traffic signal schedule advisory. In *Proceedings of the MobiSys.* ACM, 127–140.

S. Lahiri and S. Qadeer. 2008. Back to the future: Revisiting precise program verification using SMT solvers. In *Proceedings of the POPL.* ACM, 171–182.

N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell. 2010. A survey of mobile phone sensing. *IEEE Commun. Mag.* 48, 9 (2010), 140–150.

L. Lopes and F. Martins. 2016. A safe-by-design programming language for wireless sensor networks. *J. Syst. Arch.* 63 (2016), 16–32.

C. F. Mass and L. E. Madaus. 2014. Surface pressure observations from smartphones: A potential revolution for high-resolution weather prediction?*Bull. Amer. Meteorol. Soc.* 95, 9 (2014), 1343–1349.

R. Newton and M. Welsh. 2004. Region streams: Functional macroprogramming for sensor networks. In *Proceedings of the DMSN.* ACM, 78–87.

O. Okolloh. 2009. Ushahidi, or 'testimony': Web 2.0 tools for crowdsourcing crisis information. *Participat. Learn. Action* 59, 1 (2009), 65–70.

OpenSignal. 2010. OpenSignal. Retrieved from https://opensignal.com/.

A. Overeem, J. C. R. Robinson, H. Leijnse, G. H. Steeneveld, B. K. P. Horn, and R. Uijlenhoet. 2013. Crowdsourcing urban air temperatures from smartphone battery temperatures. *Geophys. Res. Lett.* 40, 15 (2013), 4081–4085.

P. Piejko. 2017. Global Mobile Statistics 2017. Retrieved from https://mobiforge.com/research-analysis/13-statistics-on-mobile-web-performance-in-2017.

C. Price and H. Shachaf. 2017. Using smartphone data for studying natural hazards. In *EGU General Assembly Conference Abstracts (EGU General Assembly Conference Abstracts)*, Vol. 19. EGU, 2659.

M.-R. Ra, B. Liu, T. F. La Porta, and R. Govindan. 2012. Medusa: A programming framework for crowd-sensing applications. In *Proceedings of the MobiSys.* ACM, 337–350.

V. Radu, L. Kriara, and M. K. Marina. 2013. Pazl: A mobile crowdsensing based indoor WiFi monitoring system. In *Proceedings of the CNSM.* IEEE, 75–83.

J. Rodrigues, E. R. B. Marques, L. Lopes, and F. Silva. 2017. Toward a middleware for mobile edge-cloud applications. In *Proceedings of the MECC.* ACM, Article 1, 1:1–1:6 pages.

J. Rodrigues, E. R. B. Marques, J. Silva, L. Lopes, and F. Silva. 2018. Video dissemination in untethered edge-clouds: A case study. In *Proceedings of the DAIS (to appear).* Springer.

T. Roopa, A. N. Iyer, and S. Rangaswamy. 2013. Crotis – crowdsourcing based traffic information system. In *Proceedings of the BDC*. IEEE, 271–277.

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 *Proceedings of the ICFEC*. IEEE, 61–67.

R. Simpson, K. R. Page, and D. De Roure. 2014. Zooniverse: Observing the world's largest citizen science platform. In *Proceedings of the WWW*. ACM, 1049–1054.

The Internet Society. 2015. Internet Society Global Report 2015—Mobile Evolution and Development of the Internet. Retrieved from https://www.internetsociety.org/globalinternetreport/2015/assets/download/IS_web.pdf.

S. Tarkoma, M. Siekkinen, E. Lagerspetz, and Y. Xiao. 2014. *Smartphone Energy Consumption: Modeling and Optimization.* Cambridge University Press.

D. R. Thomas, A. R. Beresford, and A. Rice. 2015. Security metrics for the Android ecosystem. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 87–98.

D. T. Wagner, A. Rice, and A. R. Beresford. 2013. Device analyzer: Understanding smartphone usage. In *Proceedings of the MobiQuitous*. Springer, 195–208.

D. T. Wagner, A. Rice, and A. R. Beresford. 2014. Device analyzer: Large-scale mobile data collection. *ACM SIGMETRICS Perform. Eval. Rev.* 41, 4 (2014), 53–56.

J. Wakefield. 2011. 3G mobile data network crowd-sourcing map by BBC News. Retrieved from http://www.bbc.com/news/business-14574816.

M. Welsh and G. Mainland. 2004. Programming sensor networks using abstract regions. In *Proceedings of the NSDI*. USENIX Association.

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 *Proceedings of the MobiWIS*. Springer, 172–186.

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 *Proceedings of the UbiComp*. ACM, 415–426.

Received January 2018; accepted March 2018