RapIoT Toolkit: Rapid Prototyping of Collaborative Internet of Things Applications

Francesco Gianni, Simone Mora, Monica Divitini

Norwegian University of Science and Technology
Department of Computer Science, Sem Sælandsvei 9
Trondheim, Norway

Abstract

The Internet of Things holds huge promise in enhancing collaboration in multiple application domains. Bringing internet connectivity to everyday objects and environments promotes ubiquitous access to information and integration with third-party systems. Further, connected "things" can be used as physical interfaces to enable users to cooperate, leveraging multiple devices via parallel and distributed actions. Yet creating prototypes of IoT systems is a complex task for developers non-expert in IoT, as it requires dealing with multi-layered hardware and software infrastructures. We introduce RapIoT, a software toolkit that facilitates the prototyping of IoT systems by providing an integrated set of technologies. Our solution abstracts low-level details and communication protocols, allowing developers non-expert in IoT to focus on application logic, facilitating rapid prototyping. RapIoT supports the development of collaborative applications by enabling the definition of high-level data type primitives and allowing interactions spread among multiple smart objects. RapIoT primitives act as a loosely coupled interface between generic IoT devices and applications, simplifying the development of systems that make use of an ecology of devices distributed to multiple users and environments. We illustrate the potential of our toolkit by presenting the development process of an IoT application ideated during a workshop with non-expert developers and addressing real-world

Email addresses: `francesco.gianni@ntnu.no` (Francesco Gianni), `simone.mora@ntnu.no` (Simone Mora), `monica.divitini@ntnu.no` (Monica Divitini)

challenges affecting smart cities. We conclude by discussing the strength and limitations of our platform, highlighting further possible uses for collaborative applications.

Keywords: Internet of Things, IoT, Ubiquitous Computing, Development, Toolkit, Smart Cities

1. Introduction

The Internet of Things (IoT) holds huge promise in enhancing computer-supported collaboration in several application domains. By enabling seamless interconnection of people, computers, everyday objects, and environments, it promotes collaboration off the screen in our everyday routines. Additionally, increasing the amount and quality of information captured by connected objects might ultimately improve collaboration among people using those objects [1]. The technological matrix of Computer Supported Cooperative Work (CSCW) is evolving to facilitate context-aware computing, mobile communication, and interaction. Support for this paradigm shift also comes from the already established collaborative potentials and implications of the IoT [2].

Research has shown how IoT systems can leverage connected objects in collaborative applications, for example, to support patient/physician dialogue in chronic disease treatments [3], to foster social communication among friends and relatives [4], to enhance collaboration in crisis management [5], and to support citizens' participation in public administration [6].

Since the term "Internet of Things" was coined in 1999 by technologist Kevin Ashton [7], research has mainly focused on developing machine-centric infrastructures to enable connected things to exchange information over the internet. Wireless sensor networks (WSN), machine-to-machine (M2M) communication, and technologies connected to design, deployment, and operation of WSN are some of the most common topics of interest.

Few works [8, 1] have investigated how the IoT can enable collaboration and how HCI theory could drive the development of IoT collaborative applications.

Likewise, only a few works have investigated collaborative IoT application authoring [9] and how to involve users in design activities [10, 11].

We define a collaborative IoT application as a technological application where users are engaged in a joint effort, having the ability and flexibility to align their interactions through the support provided by ecologies of interconnected things.

The CSCW agenda has become relevant for these activities, having expanded out of the boundaries of the work environment, and diversified into new areas of human activity. The social organization of these activities, as well as the intertwining of social science with computer science to explore, inform, and propel technological research, is of crucial importance to the continued development of CSCW [12]. Further, connected and interactive objects have been employed in CSCW applications for long time; for example as awareness and coordination devices [13].

With RapIoT, we concentrate on supporting interaction in the physical world in connection with the digital domain. Starting from a list of design goals, grouped in infrastructure, support for developers, and support for collaborative applications, we propose an architecture oriented to support IoT applications that make use of tangible interaction through smart objects in the physical world. We foresee "things" and smart objects as enabling artifacts for shared, collective, and collaborative activities [14]. This concept has its roots in Greenberg's conceptualization of *physical collaborative interfaces* [15] as devices situated in the physical world and designed for collaborative use. These devices may retain the appearance of everyday objects but are able to collect data, visualize information, provide feedback, and sense user interaction and manipulation.

This is a substantial and novel paradigm shift for IoT applications, since lack of mobility is a typical limitation of common IoT devices [16].

The target end user of a RapIoT application needs only to be able to physically interact with the augmented object, which does not require any particular skill, since the object retains its original affordances.

We define RapIoT developers as "non-experts" in IoT. They differ from professionals in the fact that they do not have any skill in electronics, networking protocols, or the assembling and configuration of IoT devices. They do have some programming proficiency and are able to code using standard constructs of high-level programming languages such as JavaScript, Python, etc.

Makers, designers, and students are examples of non-expert developers who can be part of a participatory design-oriented strategy to allow a wide population to take advantage of the potential that IoT technologies offer for collaborative applications.

We foresee that their involvement in design and programming for the IoT will result from a lowering of the threshold of skills required to build prototypes. Although a number of tools are available to support IoT development, those tools often (i) do not offer integrated support to multiple architectural layers, (ii) require pre-existing knowledge in hardware development or embedded programming, and are thus not suitable for non-experts in IoT, and (iii) are often bound to specific hardware and vendor-locked technologies. This results in a steep learning curve for the tools and large time for integration, obstructing the ability and speed with which developers may explore design choices by iterating on the implementation of functional prototypes.

Rapid prototyping is an important development process when creating innovative IoT applications. Ideas can be quickly tested and refined, keeping costs low. Through rapid prototyping, we aim to encourage and engage non-expert developers in exploring the vast solution space offered by IoT technologies. However, prototyping IoT systems is challenging because doing so requires dealing with a heterogeneous mix of hardware and software components arranged in a multi-layer architecture. Lowering the entry barriers and facilitating adoption are steps needed to achieve participation in brainstorming and other collective activities [17]. Toolkits for IoT address these issues: They provide an integrated set of technologies and practices to simplify and scaffold prototyping.

A popular architectural pattern for IoT toolkits consists of three layers [18]:

- an *embedded layer*, implemented as a physical object augmented with sensors, actuators, and short-range wireless connectivity to provide sensing and user interface capabilities;

- a *gateway layer*, implemented as a device such as a smartphone or WiFi router, to provide connectivity to the embedded layer, enabling ubiquitous access to information;
- a *server layer*, implemented as a cloud service, which enables data storage and integration with third-party services.

RapIoT provides support for all three layers of this type of architecture. RapIoT does not explicitly support a specific application domain, acting as an enabling technology allowing non-experts to develop collaborative applications. From this perspective, RapIoT enables the definition, implementation, and manipulation of high-level *data type primitives*. RapIoT primitives allow developers to abstract out low-level implementation details and provide a loosely coupled interface between different architectural layers, allowing IoT devices to serve different applications without the need for firmware reprogramming and thus offering a platform as a service. They introduce a simple shared construct that traverses the three layers of the architecture, providing continuity and facilitating rapid prototyping and deployment of IoT applications.

We envision the emerging domain of collaborative IoT applications for smart cities as a possible application context, where the city is not merely a group of persons but a vibrant ecosystem of communities. Through collaborative practices involving multiple artifacts, citizens build awareness and enable lifelong learning [19]. We concentrate on this domain, which is particularly timely and fitting, given the stagnation of technological advancements in applications for smart cities [20] and the potential impact on society, supporting improvements of all key factors contributing to regional competitiveness: mobility, environment, people, quality of life, and governance [21].

In Section 2, we provide an analysis of existing IoT frameworks and toolkits. In Section 3, we summarize the characteristics of an IoT toolkit that can support non-experts in developing collaborative applications. We then present *RapIoT*, an integrated set of tools to support rapid prototyping of IoT applications, previously introduced in [22]. The RapIoT approach is then described in detail, in relation to IoT applications developed in the smart city domain. We discuss the strengths and weaknesses of our approach, and we conclude the paper by highlighting future works.

2. Related Work

Several works have provided tools to facilitate the development of IoT systems. Aside from relying on standard protocols and APIs that allow mutual integration, each tool often focuses on supporting a specific architectural layer. In the remainder of this section, we survey development toolkits that can be used for IoT prototyping.

2.1. Embedded Layer

Modkit [23] extends the Arduino [24] platform providing a block-based visual programming language based on the Scratch project [25], further expanding Arduino target users to non-professional developers such as kids, designers, and artists. Focused on developing interfaces based on simple input/output feedbacks, Bloctopus [26] provides a platform based on modules with sensor-actuator couplings and a hybrid visual and textual programming language. Micro:bit [27] is a small electronic board equipped with a microcontroller, a low-fidelity display, and a few sensors. Micro:bit can be programmed with high-level programming languages and has been used extensively in schools.

2.2. Gateway Layer

Developing or deploying gateways to provide internet connectivity to resource-constrained embedded devices is particularly limiting for non-experts, as it requires pre-existing knowledge of low-level technologies such as transport protocols and wireless networks.

Fabryq [28] simplifies the development and deployment of internet gateways for Bluetooth low-energy (BLE) devices by abstracting the complexity of dealing with multiple languages and networking aspects. Rather than invoking BLE commands on each local device, the platform provides a proxy to access multiple devices via a centralized API.

Zhu et al. [29] have addressed the development of a gateway for ZigBee[1] wireless devices. IoT devices can be controlled and accessed remotely, and the gateway handles conversion between different data protocols.

Commercial IoT gateways such as Libelium's Meshlium[2] and Multitech's MultiConnect Conduit[3] are standalone fixed devices which provide a bridge between WSNs and the cloud. Conduit offers BLE connectivity for IoT devices, while Meshlium relies only on the ZigBee protocol to communicate with the sensors. On the cloud side, they both offer WiFi and 3G/4G connection; MQTT protocol[4] is also supported. Conduit's onboard software can be developed, depending on the specific model, using either the Node-RED visual programming language or the mLinux development environment, which allows coding in C++, C#, Python, and Java, among others. Meshlium allows only local data storage or the transfer of sensor readings to a list of supported cloud services such as IBM Bluemix, Microsoft Azure IoT Hub, and Amazon IoT.

2.3. Server Layer

The server layer is the core element that manages IoT devices connected via multiple gateways and interacts with third-party web services such as data providers or social networks.

The framework *PatRICIA* [30] leverages a programming model and a cloud-based execution environment to reduce complexity and support scalable development of IoT applications. Similarly, the framework developed by Khodadadi et al. [31] focuses on connecting data sources by managing querying and filtering of data and facilitating sharing with third-party platforms. Their work takes

[1]http://www.zigbee.org
[2]http://www.libelium.com/products/meshlium/wsn
[3]https://www.multitech.com/brands/multiconnect-conduit
[4]http://mqtt.org

into account data gathering from multiple sources such as sensor networks and other web applications (blogs, social media, databases). Users are provided with an API to configure data sources and to trigger actions within stand alone applications. Kovatsch et al. [32] describe a similar higher-level architecture. They address the need for an API for connected devices for pushing and retrieving data.

IFTTT is an online platform to connect event conditions, called “triggers”, generated by a device or online service, to “actions” associated with other devices or services. IFTTT is not exclusively oriented to the IoT but supports a number of physical smart devices that can be used to trigger events or to perform actions in response to a triggered event.

SpaceBrew[5] is a software toolkit to connect interactive things to each other, which can be defined as publishers or subscribers. The data is exchanged as Boolean, numeric, or string values. Data processing is handled by the interactive things themselves in addition to sensing or actuation.

Paraimpu [33] allows developers to connect sensors and actuators through a centralized RESTful service. Data is exchanged in several formats, among which are numeric, text, JSON, and XML. A simple programmable layer between sensor and actuator allows developers to specify filters and conditional logic. The programming language to code the logic is dependent on the data type used by the sensor, for example, RegEx, JavaScript, or XPath. Internet services like Twitter can act as a sensor and be connected to actuators. Arduino can be used as an actuator, but a specific sketch should be generated and downloaded to embed the logic and to handle the output pins.

Node-RED is a visual data flow programming language (VDFPL) which also targets IoT scenarios. It allows developers to create flows connecting self-contained blocks which are treated as black boxes, following the principles driving the VDFPL paradigm.

Shiftr.io is an online MQTT broker as a service. It allows interconnection

[5]http://docs.spacebrew.cc/about

of MQTT clients and online message flow visualization. Data and MQTT connected services sharing is publicly encouraged by the platform's design. Shiftr.io uses the same MQTT messaging protocol adopted in RapIoT. The service targets the IoT facilitating the interconnection of MQTT clients, which can run on different types of hardware devices.

Amazon AWS offers two IoT oriented services: AWS Greengrass and AWS IoT Platform. Both provide secure messaging among devices, connection to the AWS cloud and an SDK to code the application. Greengrass targets the gateway layer, while IoT Platform addresses the cloud.

WoTkit [9] is a toolkit for IoT *mashups*: web applications that blend data and services available on the web with physical data sources such as IoT devices. Data is combined and visualized through a dashboard, accessible using a browser. WotKit focuses less on the integration of applications, rather providing basic built-in visuals and processing components [9].

2.4. The RapIoT Position

Our system takes advantage of the Arduino platform but differs from the approach used by Modkit, Micro:bit, and Bloctopus since we do not try to include any application logic in the embedded layer. In RapIoT, the implementation of the embedded layer simply provides a domain-specific language (DSL) as an Arduino library whose only purpose is to facilitate the definition and coding of input/output primitives. The approach used in Fabryq [28] requires pre-existing knowledge about the BLE protocol, so the toolkit is not suitable for the skill level of non-expert developers. The gateway solution by Zhu et al. [29] implies that only the parent node is connected to the network; child nodes are not directly accessible from the application environment, hindering multi-object dynamics in the application logic. *Conduit* gateway is a powerful device, but its limitations mainly consist of the price being around ten times the price of a smartphone with the same connectivity, processing power and of the device not being designed to be power efficient or portable. Both *Conduit* and *Meshlium* are devices clearly oriented to WSNs deployed in a fixed environment. They do not fit in a scenario where deployment flexibility is a requirement: To support a new sensor topology, the application on the gateway needs to be updated and redeployed. This process requires an external computer and programming skills in the languages supported. The proposed approach and implementation of *RapMobile* is suited to solve many of the limitations conventional IoT gateways present. Zachariah et al. [18] describe in detail the shortcomings of current IoT gateways. Their envisioned solution presents many points in common with *RapMobile*, including (i) disentangling of network connectivity, in-network processing, and user interface functions; (ii) leveraging of BLE connectivity for IoT devices and a mobile application as a gateway; (iii) providing application-agnostic connectivity; (iv) using a single mobile application to connect heterogeneous BLE devices in an opportunistic way; and (v) moving the power burden of WiFi/3G/4G network protocols from the IoT devices to the gateway. The framework *PatRICIA* [30] focuses on providing sensor management in a cloud environment and storing data received from connected devices, neglecting interaction with other third-party solutions. They also neglect the management of connected devices through an API and rather focus on reading and combining data from different sources. Each device needs to be directly connected to the cloud through the MQTT protocol, which prevents the inclusion of mobile and low-powered IoT devices. The solution proposed by Kovatsch et al. [32] enables devices to publish data to third-party servers but doesn't support bi-directional exchange of events in real-time. The restrictions of IFTTT lie in its supported services, devices, triggers, and actions, which are limited to those offered by the platform and not extensible by end users. Users are then constrained to mixing and matching triggers and events already implemented. SpaceBrew and Paraimpu do not facilitate deployment of applications; the user has to find a way to connect the things, sensors, and actuators to the web. They also lack a unified structure for the messaging protocol and data format. In Paraimpu, the programming logic is also dependent on the data format, requiring programming skills in different languages even for simple scenarios. Both platforms allow only very simple logic constructs in the applications, and SpaceBrew is also constrained by the data type used: For example, it is not possible to connect a publisher of Boolean values to a subscriber that handles strings. In SpaceBrew, computation does not happen in the cloud but on the devices, which by definition are usually quite limited in processing power and are often running on batteries. The Node-RED platform is based on an interesting programming approach which can possibly complement the RapIoT platform. Node-RED supports the programming phase but is not a full-stack toolkit intended to handle and scaffold deployment, hardware device programming, and data flow from BLE-enabled sensors and actuators. Although Shiftr.io provides some advanced MQTT functionalities such as graph-based live visualization of messages and data sharing, the core function remains to serve as an online MQTT broker. Limitations are imposed on the supported Arduino hardware, which should have onboard WiFi/Ethernet connectivity, restricting the choice to few boards such as Arduino Yún or others that require cable-based connectivity[6]. RapIoT already integrates an online MQTT broker supporting the message flow from the hardware devices to the programming environment. Shiftr.io does not offer an integrated full-stack platform, presenting similar limitations as the ones discussed for Node-RED. The Amazon AWS services for IoT allow developers to connect IoT devices to the Amazon cloud. However, no low-power BLE hardware is supported, and the devices are required to have support for WiFi/Ethernet connectivity and run a Linux OS or a software stack such as Python, Node.js, or Java, which is not usually supported by low-power microcontrollers. No support is provided for a gateway able to bridge BLE connections from IoT devices to the cloud. Compared to WoTkit [9], RapIoT focuses more on interaction in the physical world. No widget or computer-based visualizations are supported since the architecture of RapIoT is oriented to tangible interaction through smart objects in the physical world.

[6]https://github.com/256dpi/arduino-mqtt

2.5. Summary of Differences with Related Works

We reviewed toolkits that can be used to support the development of the embedded, gateway, and server layers of an IoT infrastructure and summarized the results in Table 1. Current solutions often support only a subset of the architectural layers of a common IoT toolkit. The knowledge required to use each tool also varies according to the level of abstraction it provides and the complexity of the applications that can be achieved. With RapIoT, we chose to relax some of the constraints typically found in single-layer toolkits — namely device-specific optimizations or fine-tuned trade-offs among energy efficiency, accuracy, and latency — to gain a better support for non-expert developers and rapid prototyping.

Table 1: Architecture layers covered and non-expert developers support of related works and RapIoT.

	Modkit	Bloctopus	Fabryq	Zhu et al.[29]	Meshlium	Conduit	PatRICIA	Khodadadi et al.[31]	Kovatsch et al.[32]	IFTTT	SpaceBrew	Paraimpu	Node-RED	Shiftr.io	WoTkit	AWS	RapIoT
Embedded	●	●	●								●						●
Gateway			●	●	●	●										●	●
Cloud			●				●	●	●	●	●	●	●	●	●	●	●
Non-experts	●	●								●			●	●		●	●

3. RapIoT Fundamentals

3.1. RapIoT Architecture: Design Goals

RapIoT aims at providing holistic support to non-experts developing collaborative IoT applications. The following architectural design goals constitute the foundation of our platform; we grouped them into (i) hardware and infrastructure guidelines (Table 2), (ii) directions to support non-expert developers (Table 3), and (iii) characteristics supporting collaborative applications (Table 4).

Table 2: Hardware and infrastructure.

A1	**Support both novice and expert developers** – provide basic, simple-to-use functionalities without hindering expert users in building complex systems.
A2	**Decouple infrastructure from application** – provide the IoT infrastructure as a service to applications. In this way the infrastructure (IoT devices, gateways, and servers) is completely decoupled and can be reused across different applications with no or little changes.
A3	**Hide hardware complexities** – provide high-level representations of low-level embedded hardware complexities.
A4	**Hide networking details** – spare developers from implementing connection and data transfer protocols.
A5	**Support for generic embedded devices** – enable the development of applications that make use of a wide range of IoT devices, no matter of manufacturer.
A6	**Support for multiple embedded devices** – enable the development of IoT systems that make use of multiple devices which collaborate as a structured ecology.

Table 3: Support for non-expert developers.

B1	**Employ technology close to the user** – use of mainstream solutions and technologies that are easily accessible and widespread.
B2	**Provide an efficient workflow** – minimize the time needed to deploy a first prototype of an IoT application; this includes using programming languages which allow for quick evaluation of the application code.
B3	**Have a low cost** – adopt low-cost hardware and software solutions which are free to use and preferably open source.
B4	**Have a generic architecture** – provide a structure that is adaptable and extensible to different problem domains.
B5	**Empower community support** – facilitate cooperative work and reuse of knowledge. All the points above allow for community-based sharing of knowledge and support.

With these goals, we promote hands-on collaboration based on the shared physical experience of a small community. At the same time, we also support asynchronous coordination and information sharing via connected services.

3.2. Input/Output Primitives

One of the crucial features of RapIoT is the concept and implementation of high-level input/output primitives. We envisioned a developer-friendly construct that could be easily grasped by non-experts while supporting data exchange in collaborative multi-object IoT applications. Making primitives human readable facilitates development and debugging as opposed to dealing with raw

Table 4: Support for collaborative applications.

C1	**Support for coordination of interdependent activities across space** – which is one of the problems faced by actors engaged in cooperative work *in the wild* [34].
C2	**Integration with third-party services** – provide hooks for web standards and cloud computing, which are base technologies for IoT systems [35].
C3	**Support for tangible interaction, physical user interfaces, and smart objects** – use of physical affordances to interact with computer systems, which have been proved effective in supporting collaboration [36, p. 97]. The IoT can leverage physical and embodied interaction approaches to interact with the "things".
C3	**Interaction spread among multiple things** – support a user experience distributed on an ecology of devices, providing more opportunities for collaboration via distributed actions performed by users on multiple interfaces.

sensor data. A RapIoT *input primitive* is discrete information sensed by an IoT device, for example, a data-point captured by a sensor or a manipulation performed via a user interface. An *output primitive* is an action that can be performed by the IoT device via output components such as actuators or displays, for example, a motor spinning or an LED (light-emitting diode) blinking (Figure 1). Primitives act as a loosely coupled interface between embedded devices and the application logic. Each primitive encapsulates a data type plus up to two optional parameters as payload. An example of an input primitive is "AirQuality *(primitive name)*, city center *(parameter 1)*, low *(parameter 2)*" in case of an air quality sensor device or "Knocked, twice" in case of a smart home equipped with an accelerometer device on the front door. Otherwise "Vibrate, long" represents an output primitive that issues a vibrate command to a necklace equipped with a haptic motor device.

The role of primitives is twofold. They allow an event-driven approach to programming, providing at the same time simple constructs to describe the data exchanged between embedded devices and applications. Furthermore, they allow non-expert developers to think in terms of high-level abstractions without dealing with hardware complexities, e.g., "shake, clockwise rotation, free fall" for physical manipulations detected by accelerometer data. The definition, implementation, and registration of primitives is performed by programming the firmware of an Arduino-compatible device, and the primitives are then available

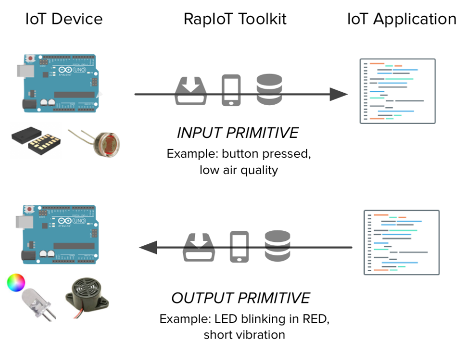

IoT Device
RapIoT Toolkit
IoT Application
INPUT PRIMITIVE
Example: button pressed, low air quality
OUTPUT PRIMITIVE
Example: LED blinking in RED, short vibration

Figure 1: Structure of input and output primitives.

to the toolkit. When coding the firmware, it is possible to deal with low-level hardware details, for example, accelerometer or GPS sensors as well as motor or display actuators. Primitives not only support simple input/output operations, they can also encapsulate more complex behaviors to support the development of physical interfaces, as illustrated in [11]. An example of HCI primitive introduced in [11] is the *"proximity"* input primitive. The primitive does not encapsulate any sensor data from the surrounding environment but is triggered when one or more IoT devices are moved close to one another. It is available to be used by devices that have the on-board hardware to support the functionality (i.e., RFID antennas and tags to sense one another). Primitives completely rely on the operations supported by the hardware, both in terms of input and output capabilities. They are bound to the hardware device and its sensing and actuation means. The gateway and server layer do not embed any specific list of primitives; rather, these two architectural layers transparently allow the programmer to work with any primitive supported by the hardware when coding the application.

3.3. Architecture

We now present the architecture of RapIoT, describing the requirements for the hardware, supported devices, software features and developer interaction. RapIoT follows the three-layer architecture described in Section 1 and represented in Figure 2. The implementation of each software stack is discussed in Section 6.

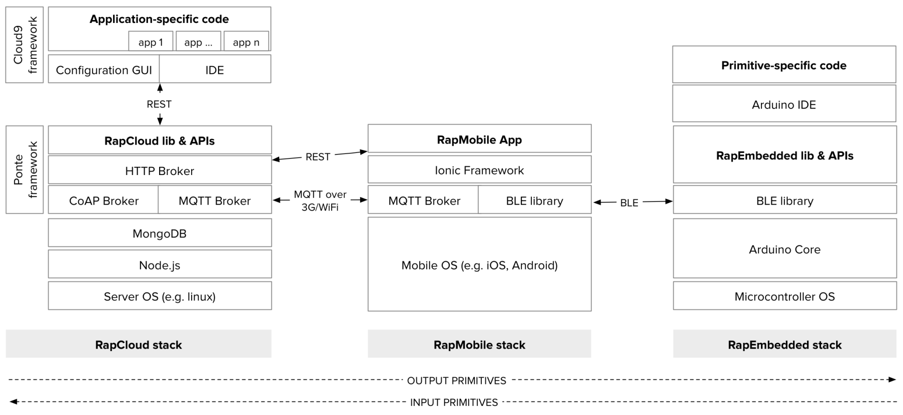


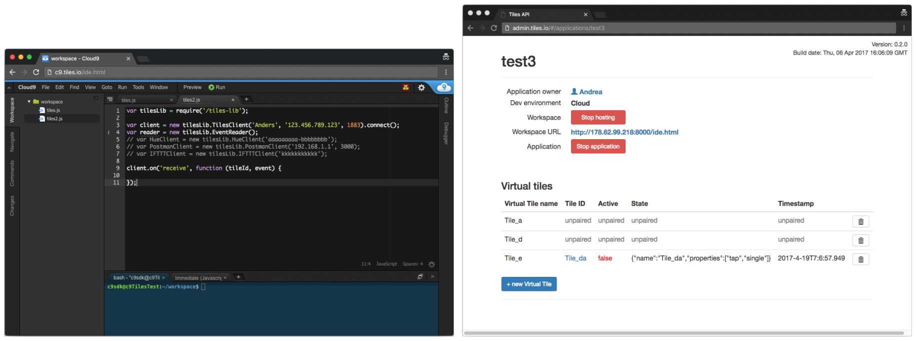
Figure 2: RapCloud infrastructure implementation.

Server Layer

Rapcloud: This consists of an online IDE[7], a JavaScript library, and a web-based configuration utility, as seen in Figure 3. The online IDE is based on the Cloud9[8] platform, which combines a code editor and a back-end server workspace based on Ubuntu[9] Linux. The whole platform is provided as SaaS (software as a service) through the browser; developers do not need to install any software on their own computers, as the application code developed is automatically saved and ready to be executed directly on the server. To get started with the application coding, as a first step, the developer gets access to the web-based configuration utility and signs up, choosing a username and password. She can then create the skeleton of her first IoT application, picking a name and adding as many *Virtual Devices* as needed. *Virtual Devices* are placeholders used as IoT device handlers. They are available in the form of JavaScript objects when writing the application code. As an example, for a "smart shower" application,

[7]Integrated Development Environment
[8]https://c9.io
[9]https://www.ubuntu.com

two *Virtual Devices* can be named "shower handle" and "shower light". To add a *Virtual Device*, the developer needs only to specify its name; no other information is needed. The developer can then launch the Cloud9 IDE, where a precompiled JavaScript source file is made available to be extended with custom application code. The precompiled source file includes the JavaScript objects of the *Virtual Devices* previously defined, ready to be employed by the developer when coding the application logic and handling the primitives.



```
var tilesLib = require('/tiles-lib');

var client = new tilesLib.TilesClient('Anders', '123.456.789.123', 1883).connect();
var reader = new tilesLib.EventReader();
// var HueClient = new tilesLib.HueClient('aaaaaaaaa-bbbbbbbb');
// var PostmanClient = new tilesLib.PostmanClient('192.168.1.1', 3000);
// var IFTTTClient = new tilesLib.IFTTTClient('kkkkkkkkkkk');

client.on('receive', function (tileId, event) {

});
```

```
c9sdk@c9TilesTest:~/workspace$
```

Version: 0.2.0
Build date: Thu, 06 Apr 2017 16:06:09 GMT

test3

Application owner	Andrea
Dev environment	Cloud
Workspace	Stop hosting
Workspace URL	http://178.62.99.218:8000/ide.html
Application	Stop application

Virtual tiles

Virtual Tile name	Tile ID	Active	State	Timestamp
Tile_a	unpaired	unpaired	unpaired	unpaired
Tile_d	unpaired	unpaired	unpaired	unpaired
Tile_e	Tile_da	false	{"name":"Tile_da","properties":["tap","single"]}	2017-4-19T7:6:57.949

Figure 3: RapCloud infrastructure: the Cloud9 online IDE on the left and the web-based configuration utility on the right.

Gateway Layer

RapMobile: This is a cross-platform mobile app for Android and iOS devices that acts as an internet gateway and allows developers discover and configure IoT devices. The app at first connects to the *RapCloud* server, and a username and password must be entered to identify the user. The end user is then presented with the list of applications that the developer previously created in *RapCloud*, fetched directly from the server. Tapping on the application that needs to be launched, the user is faced with the list of *Virtual Devices* for that application, previously defined in the *RapCloud* environment. Following on our "smart shower" example, the "shower handle" and "shower light" entries will be visualized. Tapping on each *Virtual Device*, the user can associate it to a BLE IoT device available in proximity to the smartphone. Since there might be several BLE devices in the vicinity, the user is presented with a list to choose from, containing the BLE advertised names. Tapping on one of the advertised names associates the IoT device to the *Virtual Device*. The association is stored locally on the *RapMobile* app and is remembered until the user deletes it. Once all the *Virtual Devices* have been associated to physical devices, the IoT application (i.e., the “smart shower” application) can be started directly from the mobile app. Whenever the application is running, the phone can be set in standby mode, but it should remain within a 10 meter reach of the hardware devices to ensure reliable data transfer. We call this process of association between virtual and physical devices *application appropriation*. The GUI supporting appropriation and execution of the application is shown in Figure 4.

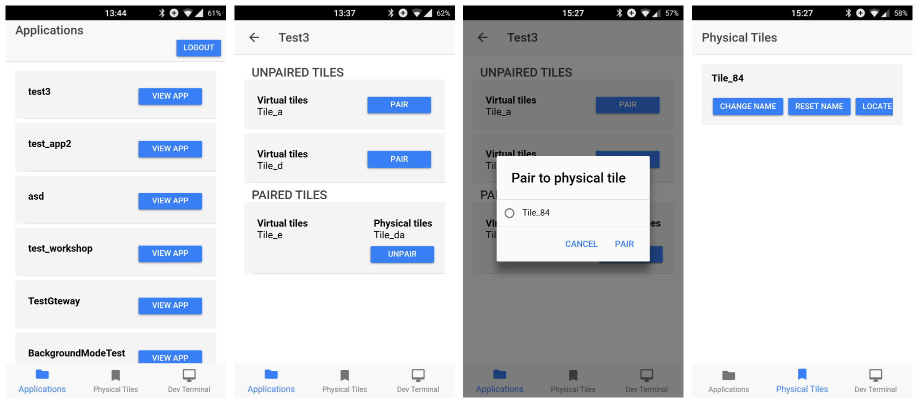


Figure 4: RapMobile application.

The *RapMobile* app transparently bridges the BLE and MQTT protocols. When a primitive is received from a BLE device, the internal components of the app re-route the complete primitive packet through the MQTT connection with *RapCloud*. The *RapMobile* gateway includes all the needed information to correctly route the primitives: For every input primitive received, the sender IoT device is known, and the associated *Virtual Device* belongs to an application running on *RapCloud*, to which the MQTT packet is forwarded. The same packet routing steps apply in inverse order when an output primitive is forwarded from *RapCloud* to the IoT device.

Embedded Layer

RapEmbedded: This consists of an Arduino library to support the definition and implementation of input and output primitives on electronic Arduino boards and microcontrollers. Arduino is a popular prototyping platform which includes both a microcontroller board to which sensors and actuators can be wired and a software library created to simplify writing code without limiting flexibility [24]. The Arduino library spares developers from learning microcontroller-specific instructions or electronic principles. Device-specific optimizations are handled by the deployment toolchain, which compiles the Arduino code into a microcontroller-optimized binary.

3.4. Hardware Requirements

The server-side software runs completely on Linux. A low-end Linux server or a small Linux virtual machine will suffice to cover the requirements. The requirements for the device running the mobile application, typically a smartphone, are limited to providing BLE and WiFi or cellular connectivity. The BLE interface is used to connect one or more Arduino-enabled boards, while a WiFi or cellular network provides TCP/IP connectivity to the cloud. The lowest architectural layer comprises BLE-enabled, Arduino-compatible boards and microcontrollers such as RFDUINO[10] and Simblee[11] boards. RFDUINO is able to run the *RapEmbedded* firmware without introducing any bottleneck in terms of CPU speed, flash, or RAM size. The microcontroller is based on a 16 MHz ARM Cortex-M0 CPU, 128kB of flash memory and 8kB of RAM. There are no other technical requirements for the embedded layer; the board can be equipped with additional I/O ports, extra connectivity, or sensors/actuators. Thanks to the open-source nature of Arduino, several independent producers were able to add Arduino compatibility to their products, allowing non-expert developers to choose from several royalty-free, affordable BLE-enabled microcontrollers and

[10]http://rfduino.com
[11]http://simblee.com

boards while at the same time taking advantage of the support provided by the growing Arduino community. Expert developers can still choose to build their own hardware, assembling the electronic to create a smart augmented object, while non-experts can simply buy a pre-assembled Arduino board, already equipped with sensors and actuators. Arduino boards are typically low power, small sized and can run on small batteries. RapIoT builds on Arduino's strengths and extends a similar approach to the IoT world. Developers interested in building applications are offered a set of primitives tailored and specific to the affordances of the IoT hardware in use, but at the same time they share a common semantic structure and are used in the same way when coding the application logic. Another point in common is the abstraction of vendor-specific programming mechanisms: Like the Arduino user, who is not required to know the type and producer of the microcontroller, RapIoT developers are not required to know any hardware- or software-related details of the IoT device. The non-expert developer need only to be aware of the set of primitives defined and available to be used for application development. The *RapEmbedded* software layer might be completely transparent to non-expert developers, who can buy a pre-programmed Arduino board embedded with the library and a firmware to handle input/output primitives. More skilled developers can decide to upgrade the firmware with a community-developed version, which can, for example, implement more or different primitives for the hardware in use. While no programming is needed to upgrade the firmware, expert developers are free to implement new primitives using the *RapEmbedded* Arduino library and flashing the new firmware on the board afterward. The *RapEmbedded* library provides functions to (i) register primitive definitions according to the name of the primitive, type (input or output), and name of (up to two) optional parameters and (ii) code conditions under which primitives are triggered, in case of input primitives, or consumed, as for output primitives.

4. Creating RapIoT Applications

In this section we illustrate how RapIoT can be employed to support the prototyping phase of an IoT application. The list of required steps and their relation with RapIoT components is reported in Figure 5. We use an IoT application for smart cities as a running example.

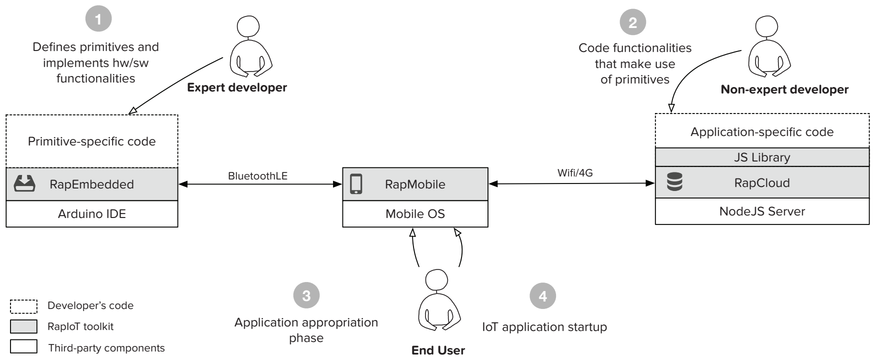


Figure 5: The RapIoT toolkit, development and deployment process.

RapIoT-supported application development and deployment is a four-step process. The first and second steps entail application development by developers, while the last two involve application appropriation by end users.

Step 1: *Device development* – This involves, in order of complexity for non-experts, either of these options: (i) building a hardware prototype of an IoT device using electronic components on a BLE-enabled, Arduino-compatible board using the *RapEmbedded* Arduino library to implement and register input/output primitives and flashing the firmware on the Arduino board; (ii) purchasing a complete IoT device or smart object based on an Arduino BLE microcontroller, coding the primitives as explained in the previous option or flashing a pre-made firmware embedding a set of primitives for the specific device; and (iii) purchasing a ready-to-use, RapIoT-compatible IoT device or smart object that mounts out of the box a firmware with coded primitives.

Step 2: *Application development* – This entails creating an application through the web-based configuration utility and coding the application logic by using the online IDE provided by *RapCloud*. Input and output primitives are here employed as programming constructs.

Step 3: *Application appropriation* – This involves selecting and starting from the *RapMobile* app an application previously developed on *RapCloud* and performing the wireless discovery of the BLE-enabled devices built in Step 1, as previously described in Section 3.3.

Step 4: *Application startup* – This entails tapping the "start application" button on the *RapMobile* app. The IoT application source code hosted on *RapCloud* is then executed.

These steps do not require advanced skills in hardware, electronics, or network protocols, only a general knowledge in coding using high-level programming languages. This matches our definition of non-expert developers reported in Section 1.

To describe the development process of RapIoT applications, we introduce as a running example the development of an application idea generated using the Tiles Ideation Toolkit for IoT [37]. The idea was produced during a workshop which involved computer science university students and consists of a *Smart Shower*, an augmented shower to promote learning of sustainable behaviors in children. The application targets children and parents, requiring a collaborative approach at a family level. The *Smart Shower* makes use of several connected IoT devices that provide feedback to the user, connect with online services, and sense the user interaction aimed at controlling the water flow. The system makes use of an IoT device that connects to the shower water handle. When the handle is operated by the child, the tilt is sensed by the IoT device, which can then infer water temperature and flow. This information is sent to the *RapCloud* server, which computes the energy consumption based on the temperature and quantity of the water used. The application logic is then configured to trigger different types of feedback based on water and energy thresholds reached. A second

IoT device, in the form of an LED array governed by a BLE-enabled Arduino microcontroller, is used to provide shared ambient feedback that are visible to individual users or to the family, for example signalling that a weekly family consumption threshold has been reached, a common goal has been achieved, or visualizing triggers to collaborative reflection based on individual consumption compared to the rest of the household. When the *RapCloud* application computes the reach of a threshold, it triggers an output primitive addressed to the LED array, which changes color from green to yellow or from yellow to red when too much water is being used. The *RapCloud* application collects data about water and energy usage, which is then shared using an online spreadsheet or charting service. Based on the data, parents can reward their children for sustainable behaviors. Families can set group goals for weekly or monthly consumption and check their progress day by day, increasing community awareness and improving collaboration to reach the shared objective. In the following, we describe the applications development and deployment process.

4.1. Device Development

Device development includes hardware and firmware development. Hardware development involves plugging together electronic components using an Arduino-compatible BLE board (Figure 6). Firmware development requires writing Arduino code that controls the hardware and handles input and output of the primitives. According to our example, the electronic for the Smart Shower uses of a Tiles Square module (an Arduino-compatible BLE board) [11] and an LED array (Figure 6). The Tiles Square module embeds in a single compact package an RFDUINO microcontroller, an accelerometer, an RGB LED light, and haptic feedback vibration.

After having installed the *RapEmbedded* library in the Arduino IDE, the developer defines the input primitive *Orientation* and the output primitive *Light-Color*. The *Orientation* primitive models the 3D position of the shower handle; it is triggered by sensor readings continuously provided by the accelerometer on the Tiles Square and has a *Position* parameter that reports the angular

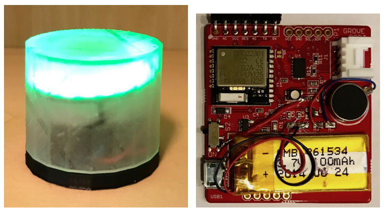


Figure 6: LED array light and Tiles Square module hardware devices. The Tiles Square measures approximately 4x4 cm.

orientation on X,Y,Z axes. The *LightColor* output primitive defines the *color* parameter which can assume “green”, “yellow”, and “red” states and causes the LED array to light up in different colors.

```
RIOTe.regPrimitive(in,"Orientation","Position");
RIOTe.regPrimitive(out,"LightColor","Color");
```

Finally, the developer codes the loop of conditions under which the input primitives are triggered according to readings from the accelerometer and implements how to consume the output primitives by issuing commands to cause the LED array to light up in different colors.

```
### RapEmbedded: Tiles Square code
if(Accelerometer.movement_detected())
  RIOTe.trigger("Orientation", Accelerometer.3D_position());

### RapEmbedded: LED Array code
RIOTe.when("LightColor", "green", LED.set_color("green"));
RIOTe.when("LightColor", "red",   LED.set_color("red"));
```

4.2. Application Development and Deployment

After the firmware is developed and deployed, each hardware device is autonomous and ready to establish a connection with *RapCloud* to send and receive primitives via the *RapMobile* app, which acts as a gateway. Primitives are now available while coding the application logic using the *RapCloud* online IDE. Back to our example, the developer first registers the application name and the *Virtual Devices* required in the web-based configuration utility. Then she proceeds with coding the application logic. When the first *Orientation* primitive is received, the application logic starts keeping track of the shower time. Water and energy consumption are inferred based on the orientation of the handle. When the consumption of either energy or water exceeds the first configured threshold, the *Light Color* output primitive is triggered to switch the LED array to yellow. If the second threshold is also reached, another *Light Color* primitive is triggered to change the LED array color to red. When the child finishes showering, data about time, energy, and water consumption is uploaded to an online spreadsheet or charting service.

The application code is written directly in the browser using the *RapCloud* online IDE (Figure 3). When the code is executed, the application is immediately available to end users.

4.3. Application Appropriation

Using the *RapMobile* app, the end user performs the *application appropriation* process as described in detail in Section 3.3. She can then start the application directly from the *RapMobile* app.

5. Initial Evaluation

Two preliminary evaluations of the system were performed: a pilot test with five computer science university students and a workshop with 14 high school students, aged 15 and 16. The users were asked to develop and deploy an IoT application in less than 60 minutes. We decided to focus the assessment on the support provided by RapIoT during the implementation of an IoT application, starting from a provided idea. For this reason, and to avoid a complicated application logic, the example application does not include any particular collaborative aspect. All the participants had some experience in programming, although none of them declared to be an expert. Their knowledge of IoT was rather generic and limited, if any; they fit into our definition of non-expert developers in IoT provided in Section 1. Data was collected in the form of observations of the process, answers to a five-point Likert scale questionnaire, and, for the pilot test only, analysis of the source code produced.

The pilot test users were divided into two groups, referenced as A and B. Both groups were able to rapidly get started with application coding and prototyping with no help from workshop supervisors. A scenario describing the IoT application they were asked to develop was provided: *Have you ever been to a party where your shoes have been separated and you can find only one when leaving? To solve this problem, you can implement an application to find a shoe when you have located the other. Double tapping on one shoe will turn on the LEDs on both shoes and vibrate the other shoe. Tilting any shoe will turn off the LEDs on both shoes.*

The code produced by group A of the pilot test can be seen in Listing 1. The groups utilized two separate IoT devices and employed two input primitives (Double Tap, Tilt) and four output primitives (Haptic Long, Haptic Burst, Led On, Led Off). Using the online IDE, both the groups implemented correctly the functionalities described in the scenario provided. The participants were also able to deploy the IoT devices and connect them to *RapCloud* using the *RapMobile* app and the web-based configuration utility. The official documentation available on the web-based configuration utility provided guidance for application deployment and development.

```
var tilesLib = require('/tiles-lib/api');
var client = new tilesLib.TilesClient('Petter', 'Petter_test', '
    ↪ cloud.rapiot.com', 1883).connect();
var reader = new tilesLib.EventReader();

client.on('receive', function (tileId, event) {
  /* AUTO GENERATED CODE START (do not remove) */
  var shoe_right = reader.getTile('shoe_right', client);
  var shoe_left = reader.getTile('shoe_left', client);
  /* AUTO GENERATED CODE END (do not remove) */
  var tileEvent = reader.readEvent(event, client);

  if (tileEvent.pName == "double tap") {
    shoe_right.trigger("led", "on", "green");
    shoe_left.trigger("led", "on", "green");
    if(tileEvent.name == shoe_left.name) {
      shoe_right.trigger("haptic", "burst");
    } else {
      shoe_left.trigger("haptic", "burst");
    }
  }
  if(tileEvent.pName == "tilt") {
    shoe_right.trigger("led", "off");
    shoe_left.trigger("led", "off");
} });
```

Listing 1: Source code produced by group A during the pilot test.

The two groups adopted different programming strategies to code the behavior. Group A produced a shorter but less robust program than group B, which included additional controls that might have been helpful when extending the application. During the workshop, we tested the same scenario used in the pilot test, providing the same tools and documentation. The participants were younger and less experienced compared to the pilot test. They were divided into four groups; each group had at its disposal a laptop, an Android smartphone running the *RapMobile* app, and several IoT devices. The students were ultimately able to prototype the IoT scenario with little help from workshop supervisors, although they had some difficulties understanding the JavaScript syntax. This issue is not indicative of a systemic problem: RapIoT can retain the same architecture and paradigm while being updated to support more user-friendly high-level programming languages. The adoption of different abstractions such as visual programming languages is a possible extension.

In Fig. 7, we report the results from six of the questionnaire statements: Q5 - the steps of the prototyping process were easy to follow; Q6 - I faced few challenges with the process description; Q8 - during the prototyping process, it was always clear to me what I was supposed to do; Q9 - following the steps of the prototyping process was fun; Q14 - using the *RapCloud* web IDE was easy; Q15 - I faced few challenges using the *RapCloud* web IDE.

Most of the questionnaire answers reported are in the positive end of the Likert scale. The participants of the workshop found the prototyping process slightly more difficult to follow than the pilot test users did. Based on the

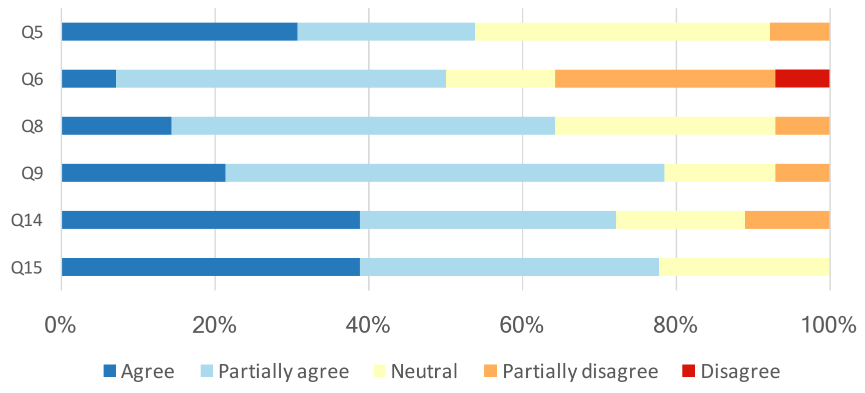


Figure 7: Questionnaire results.

feedback received and the observed behaviors, the problem seemed due to the fact that the students, being familiar with the Python programming language, tried to use its syntax when coding in JavaScript.

During the pilot test, we observed the participants struggling with the *Rap-Mobile* application; however, no issues were recorded during the workshop, where a redesigned and updated version of the application was used.

6. Implementation

Here we describe in more in detail the building blocks and technicalities of the RapIoT framework. RapIoT comprises three different stacks of software modules implementing the functionalities provided by *RapCloud*, *RapMobile*, and *RapEmbedded* (Fig. 2).

This design choice spares the implementation of event routing, since each IoT device can be unequivocally controlled by an application running in the cloud no matter where the application or the hardware is deployed. This architecture enables the reuse of deployed devices for different applications without changing the firmware. The development of RapIoT applications is supported by the Cloud9 platform, which allows non-expert developers to create, run, and debug code in a browser. Each user-made application is stored in a user-assigned workspace. The Cloud9 IDE has been extended with a configuration GUI that allows developers to create workspaces, assign them to users, and generate template code directly in a user's workspace. Both the IDE and the configuration GUI (Fig. 3) interact with lower *RapCloud* tiers via a REST interface defined by *RapCloud* APIs, allowing for easy IDE replacement or for usage of multiple IDEs. When writing the logic of a RapIoT application, non-expert developers need only to handle instances of input primitives received from the IoT devices and send instances of output primitives to those devices without the need to know how the modules implement the actual recognition and actuation of primitives. The primitives available to the user within the Cloud9 IDE can be created via the APIs provided by the *RapEmbedded* library. The library implements both registration of primitives and handling of primitives at run-time. The library is written in C++ and built on top of the Arduino IDE. The library is compatible with most Arduino boards that provide BLE connectivity (see Section 3.4). Once primitives are implemented through the *RapEmbedded* library, they can be exchanged back and forth between applications and IoT devices. Instances of output primitives are generated by an application and propagated to a specific IoT device. Otherwise, instances of input primitives are generated by a device and propagated to an application. RapIoT makes use of MQTT as a transport protocol to exchange primitives between *RapCloud* and *RapMobile* stacks. Primitives are coded in JSON-formatted messages that contain a unique identifier for the IoT device, followed by the identifier of the primitive and two optional parameters. The current implementation of the unique identifier allows the IoT device to use a BLE advertised parameter. This guarantees consistency between different mobile OSs, compared to using the Bluetooth MAC address as an identifier. Problems arise in particular with iOS, which scrambles the Bluetooth MAC address, preventing a robust mapping between IoT devices and application logic. An alternative, more reliable solution is to use a UUID[12], advertised by the IoT device via Bluetooth, before connecting to

[12]https://tools.ietf.org/html/rfc4122

the mobile app gateway. After an application written using the Cloud9 IDE is started, it begins exchanging instances of primitives on an event-driven basis. Multiple applications can run at the same time inside the server, although one device can exchange primitives with one application only. The JavaScript application created by the developer generates or consumes primitives thanks to the *RapCloud* library and API. The library parses input primitives and triggers JavaScript events that are handled by the developer's code. When an output primitive is triggered, the library takes care of building a well-formed JSON packet and forwarding it to the user's gateway over MQTT. The library makes use of Ponte[13], to bridge the REST interfaces exposed towards the Cloud9 IDE and administration GUI. Ponte also bridges *RapCloud* with the MQTT interface exposed towards *RapMobile*. Finally, *RapCloud* employs MongoDB[14] to store associations among users, applications, gateways, virtual devices, and real devices. The *RapCloud* stack runs on top of the Node.js JavaScript runtime environment[15]. *RapMobile* bridges IoT devices with applications; the app has two roles. During application appropriation (Section 3.3), it assigns *Virtual Devices* to physical devices in Bluetooth range and locks them within a specific application belonging to a user. This operation is done by the user through the *RapMobile* app (Fig. 4), which is developed using the Ionic Framework[16]. At run-time, *RapMobile* translates MQTT packages containing JSON-formatted primitives into simple comma-separated values that are forwarded to paired devices over a BLE link. Bluetooth connectivity is guaranteed thanks to the BLE library provided by the Cordova BLE Central plugin[17]. *RapMobile* also takes care of error handling when devices move out of Bluetooth range, disconnect from the network, or turn off because they run out of battery power. Concurrent access to a single IoT device, resulting in an access conflict, is prevented by

[13]http://eclipse.org/ponte/
[14]https://www.mongodb.com/
[15]https://nodejs.org
[16]https://ionicframework.com/
[17]https://github.com/don/cordova-plugin-ble-central

the BLE handshaking mechanism: When a device is connected to a gateway, it stops advertising and accepting further connections. On the gateway, a single IoT application at a time can be started. Furthermore, hardware modules can be discovered, attached to, or removed from the platform while applications are running. Special system-wide events inform connected applications of the availability of new devices in real time.

7. Discussion

In this section, we discuss strengths and limitations of RapIoT in relation to the design goals described in Section 3.1. We then elaborate on how collaboration is supported and why the approach of the toolkit is interesting for smart cities applications.

7.1. Meeting the Design Goals

Using qualitative assessments and the initial evaluation of the software toolkit presented in Section 5, we now connect the requirements listed in Section 3 with the components of RapIoT. Primitives provide high-level abstraction to encapsulate input and output data packets. Development of plug-and-play software and hardware prototyping platforms based on such high-level object abstractions could mitigate the challenges related to heterogeneity and complexity of Internet of Things network nodes as well as the diversity of modes of communication [38]. In line with the plug-and-play philosophy, RapIoT hides hardware and network complexities (A3, A4), allowing non-expert developers to concentrate the technical effort into the cloud-supported programming phase (B2). On the other hand, more expert developers can extend the primitives supported by the IoT devices through the *RapEmbedded* Arduino library (A1). Primitives permit developers to decouple the application logic, which resides in the cloud, from the rest of the infrastructure tasked with generating, consuming and routing the application-independent primitives across the embedded, gateway, and cloud layers (A2). Thanks to the neutral and multi-purpose nature of the primitives, the architecture is not tied to any specific application domain (B4). Bluetooth-equipped networked sensor nodes can achieve good interoperability with consumer devices, have lower power consumption than WiFi, and have a lower cost (B3) [39]. Bluetooth is also by far the most widespread technology supported by existing consumer devices (B1) [39]. BLE connectivity and the *RapMobile* implementation allow for several IoT devices to be connected to the application layer at the same time (A6). Constraints on the supported IoT hardware are dictated only by Arduino compatibility, which is currently provided by many hardware manufacturers and devices, while new ones are constantly added (A5). Being as both RapIoT and Arduino are open source, community support is a viable medium to share and reuse knowledge (B5). Our approach to IoT system development embeds mechanisms that facilitate the authoring of collaborative applications. Primitives are flexible constructs that allow developers to break down interaction routines and data flows into simpler blocks that can be combined when writing the application logic. The RapIoT toolkit presents four fundamental features that help in the development of collaborative applications:

- *Support for multiple devices* – RapIoT supports applications that make use of several IoT devices connected to the same gateway (C1). This allows multiple users to interact with various devices placed in the same environment, which are then ruled by a centralized application logic running on the *RapCloud* server.

- *HCI primitives for physical interaction* – Some of the primitives rely on composite human actions and events (C3), which involve more than one physical device. It is possible to design and implement applications that support time coordination, sequential actions, awareness, proximity, and other forms of cooperative practices that characterize coordinated ecologies of devices (C4). Under this perspective, the application matches with the definition of a CSCW product: a technology-driven application supporting coordination of collaborative activities [40] and with the notion of *physical collaborative interface* [15].

- *Distributed gateways and devices* – Applications developed with RapIoT can use several gateways physically located in different places, each of which can control a group of devices. This opens these devices up to more flexible scenarios of use: (i) groups of users can move from site to site where different groups of IoT devices are located and perform collaborative tasks that involve IoT devices at the site, e.g., a collaborative treasure hunt game and (ii) users can carry one or more IoT devices connected to their smartphones and perform some tasks or collect data in the environment, remotely cooperating with other users who are following the same workflow but at a different site.

- *Integration with online services* – Connection with third-party collaborative services and online data sets is supported through specific primitives (C2). Asynchronous collaboration is facilitated, allowing users to reflect and cooperate on shared resources connected to the IoT application logic.

To make collaboration happen, users should be engaged in a joint effort, with the ability and flexibility to align their goals and resources with others in real time. All parties should be brought into alignment around what's needed [41]. The technical infrastructure of RapIoT supports this collaborative effort, allowing the creation of collaborative IoT applications addressing a shared goal or challenging and engaging users at different levels.

7.2. RapIoT in a Smart City Context

Starting from ideas generated by a set of possible end users, the RapIoT architecture has been used to design the technical infrastructure of IoT applications aimed at solving real problems affecting modern smart cities. The simplicity of the approach used, resulting from the architectural choices at the base of RapIoT, can allow developers to prototype and program an initial demo of the wished behavior in a few hours. Future scenarios can involve physical data visualization devices in the city [42] and development of customized primitives tailored to handle smart city sensor data. RapIoT is not limited to any particular domain but is a promising approach able to address the problem of technical stagnation in smart city applications [20].

7.3. Limitations

The RapIoT architecture does not comprehend any coded application logic embedded into IoT devices. Since the primitives have to follow a complete round trip from the embedded layer to the application layer, network latency can be a significant factor affecting performance and application responsiveness. Network quality and availability is crucial for the entire period when the application is in use. This limitation can be particularly amplified when the application layer deals with batches of primitives in rapid sequence. In these cases, most of the execution time is spent waiting for the network, which can hinder the user experience. Another possible limitation is connected to the concept of primitives: for some applications, the behavior to encapsulate in a primitive can be too complex to be exposed with a simple interface like the one provided by input/output primitives. This restriction could be partially mitigated by splitting the logic into two or more primitives, with the drawback of delegating more work to the network.

8. Conclusions

In this paper, we presented the RapIoT toolkit for rapid prototyping of IoT applications. The development of a RapIoT application has been presented by describing the prototyping process of a solution addressing real challenges of smart cities. RapIoT leverages the concept of data primitives as communication blocks and interfaces between generic IoT devices and the application layer. Further, we have highlighted how RapIoT primitives can support the development of collaborative applications via multiple embedded devices, physical interfaces, and distributed gateways. RapIoT takes advantage of and builds on the most recent technological evolutions in the field, such as the Arduino platform, cloud computing, BLE radios, and mobile applications, reducing complexity and entry barriers for non-experts. Compared to the state of the art, with RapIoT we are lowering the adoption threshold by shielding developers from some of the complexity connected to prototyping IoT applications. This process is also facilitated by the holistic nature of the architecture encompassing all three layers of a typical IoT system. In this paper, we have focused on providing an overview of RapIoT and have illustrated through a preliminary evaluation how the development process is supported and facilitated by the toolkit. We have conducted workshops with different users to test the system: Non-expert developers were asked to code and prototype a simple IoT scenario previously presented to them. The groups were able to program the desired behavior and test the final IoT application by physically interacting with the IoT devices employed. As part of our future work, we aim to conduct more systematic studies with users with different levels of programming skill to evaluate what scaffolding mechanisms can be embedded in the toolkit so that it can be used directly by end users with no programming knowledge.

Acknowledgements

We would like to thank all the students that participated in the development of RapIoT, contributing with their work to the publication of this article.

References

[1] O. Eris, J. Drury, D. Ercolini, A collaboration-focused taxonomy of the Internet of Things, 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT) (2015) 29–34.

[2] K. Schmidt, L. Bannon, Constructing CSCW: The first quarter century, Computer Supported Cooperative Work (CSCW) 22 (4-6) (2013) 345–372.

[3] A. J. Jara, M. A. Zamora, A. F. G. Skarmeta, An internet of things–based personal device for diabetes therapy management in ambient assisted living (AAL), Personal and Ubiquitous Computing 15 (4) (2011) 431–440.

[4] M. Brereton, A. Soro, K. Vaisutis, P. Roe, The Messaging Kettle: Prototyping Connection over a Distance Between Adult Children and Older Parents, in: Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI '15, ACM, Seoul, Republic of Korea, 2015, pp. 713–716.

[5] L. Yang, S. H. Yang, L. Plotnick, How the internet of things technology enhances emergency response operations, Technological Forecasting and Social Change 80 (9) (2013) 1854–1867.

[6] N. Taylor, U. Hurley, P. Connolly, Making Community: The Wider Role of Makerspaces in Public Life, in: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, CHI '16, ACM, Santa Clara, California, USA, 2016, pp. 1415–1425.

[7] K. Ashton, That "internet of things" thing, RFiD Journal 22 (7) (2009) 97–114.

[8] T. L. Koreshoff, T. Robertson, T. W. Leong, Internet of Things: A Review of Literature and Products, in: Proceedings of the 25th Australian Computer-Human Interaction Conference: Augmentation, Application, Innovation, Collaboration, OzCHI '13, ACM, Adelaide, Australia, 2013, pp. 335–344.

[9] M. Blackstock, R. Lea, IoT mashups with the WoTKit, in: 2012 3rd International Conference on the Internet of Things (IOT), 2012, pp. 159–166.

[10] D. De Roeck, K. Slegers, J. Criel, M. Godon, L. Claeys, K. Kilpi, A. Jacobs, I Would DiYSE for It!: A Manifesto for Do-it-yourself Internet-of-things Creation, in: Proceedings of the 7th Nordic Conference on Human-

Computer Interaction: Making Sense Through Design, NordiCHI '12, ACM, Copenhagen, Denmark, 2012, pp. 170–179.

[11] S. Mora, J. Asheim, A. Kjøllesdal, M. Divitini, Tiles Cards: A Card-based Design Game for Smart Objects Ecosystems, in: Proceedings of the First International Workshop on Smart Ecosystems cReation by Visual dEsign Co-Located with the International Working Conference on Advanced Visual Interfaces (AVI 2016), Vol. 1602, CEUR-WS, Bari, Italy, 2016, pp. 19–24.

[12] A. Crabtree, T. Rodden, S. Benford, Moving with the times: IT research and the boundaries of CSCW, Computer Supported Cooperative Work (CSCW) 14 (3) (2005) 217–251.

[13] B. A. Farshchian, M. Divitini, Collaboration support for mobile users in ubiquitous environments, Handbook of Ambient Intelligence and Smart Environments (2010) 173–199.

[14] S. Bødker, Third-wave HCI, 10 years later—participation and sharing, interactions 22 (5) (2015) 24–31.

[15] S. Greenberg, Toolkits and interface creativity, Multimedia Tools and Applications 32 (2) (2007) 139–159.

[16] A. Botta, W. De Donato, V. Persico, A. Pescapé, On the integration of cloud computing and internet of things, in: Future Internet of Things and Cloud (FiCloud), 2014 International Conference On, IEEE, 2014, pp. 23–30.

[17] J. Grudin, S. Poltrock, Computer Supported Cooperative Work: The Encyclopedia of Human-Computer Interaction, Encyclopedia of Human-Computer Interaction.

[18] T. Zachariah, N. Klugman, B. Campbell, J. Adkins, N. Jackson, P. Dutta, The internet of things has a gateway problem, in: Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications, ACM, 2015, pp. 27–32.

[19] I. Wagner, M. Basile, L. Ehrenstrasser, V. Maquil, J.-J. Terrin, M. Wagner, Supporting community engagement in the city: Urban planning in the MR-tent, in: Proceedings of the Fourth International Conference on Communities and Technologies, ACM, 2009, pp. 185–194.

[20] F. Gianni, M. Divitini, Technology-enhanced Smart City Learning: A Systematic Mapping of the Literature., IxD&A 27 (2016) 28–43.

[21] R. G. Hollands, Will the real smart city please stand up?, City 12 (3) (2008) 303–320.

[22] S. Mora, F. Gianni, M. Divitini, RapIoT Toolkit: Rapid Prototyping of Collaborative Internet of Things Applications, in: Collaboration Technologies and Systems (CTS), 2016 International Conference On, IEEE, 2016, pp. 438–445.

[23] A. Millner, E. Baafi, Modkit: Blending and Extending Approachable Platforms for Creating Computer Programs and Interactive Objects, in: Proceedings of the 10th International Conference on Interaction Design and Children, IDC '11, ACM, Ann Arbor, Michigan, 2011, pp. 250–253.

[24] D. Mellis, M. Banzi, D. Cuartielles, T. Igoe, Arduino: An open electronic prototyping platform, in: Proceedings of CHI Extended Abstracts, ACM, 2007, pp. 1–11.

[25] J. Maloney, M. Resnick, N. Rusk, B. Silverman, E. Eastmond, The Scratch Programming Language and Environment, ACM Transactions on Computing Education (TOCE) 10 (4) (2010) 16–15.

[26] J. Sadler, K. Durfee, L. Shluzas, P. Blikstein, Bloctopus: A Novice Modular Sensor System for Playful Prototyping, in: TEI '15: Proceedings of the Ninth International Conference on Tangible, Embedded, and Embodied Interaction, ACM, 2015, pp. 347–354.

[27] S. Sentance, J. Waite, S. Hodges, E. MacLeod, L. Yeomans, Creating Cool Stuff: Pupils' Experience of the BBC micro: Bit, in: Proceedings of the

2017 ACM SIGCSE Technical Symposium on Computer Science Education, ACM, 2017, pp. 531–536.

[28] W. McGrath, M. Etemadi, S. Roy, B. Hartmann, Fabryq: Using phones as gateways to prototype internet of things applications using web scripting, in: Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, ACM, 2015, pp. 164–173.

[29] Q. Zhu, R. Wang, Q. Chen, Y. Liu, W. Qin, IOT Gateway: Bridging Wireless Sensor Networks into Internet of Things, in: Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference On, 2010, pp. 347–352.

[30] S. Nastic, S. Sehic, M. Vögler, H.-L. Truong, S. Dustdar, PatRICIA–A Novel Programming Model for IoT Applications on Cloud Platforms, in: 2013 IEEE 6th International Conference on Service-Oriented Computing and Applications, IEEE, 2013, pp. 53–60.

[31] F. Khodadadi, R. N. Calheiros, R. Buyya, A data-centric framework for development and deployment of Internet of Things applications in clouds, in: 2015 IEEE Tenth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2015, pp. 1–6.

[32] M. Kovatsch, S. Mayer, B. Ostermaier, Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things, in: 2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), IEEE, 2012, pp. 751–756.

[33] A. Pintus, D. Carboni, A. Piras, The anatomy of a large scale social web for internet enabled objects, in: Proceedings of the Second International Workshop on Web of Things, ACM, 2011, p. 6.

[34] K. Schmidt, Divided by a common acronym: On the fragmentation of CSCW, in: ECSCW 2009, Springer, 2009, pp. 223–242.

[35] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of Things (IoT): A vision, architectural elements, and future directions, Future Generation Computer Systems 29 (7) (2013) 1645–1660.

[36] O. Shaer, E. Hornecker, Tangible User Interfaces: Past, Present, and Future Directions, Foundations and Trends in Human–Computer Interaction 3 (1-2) (2009) 1–137.

[37] S. Mora, F. Gianni, M. Divitini, Tiles: A Card-based Ideation Toolkit for the Internet of Things, in: Proceedings of the 2017 ACM Conference Companion Publication on Designing Interactive Systems, DIS '17 Companion, ACM, Edinburgh, Scotland, 2017, pp. 587–598.

[38] I. P. Cvijikj, F. Michahelles, The toolkit approach for end-user participation in the internet of things, Architecting the Internet of Things (2011) 65–96.

[39] P. P. Pereira, J. Eliasson, R. Kyusakov, J. Delsing, A. Raayatinezhad, M. Johansson, Enabling cloud connectivity for mobile internet of things applications, in: Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium On, IEEE, 2013, pp. 518–526.

[40] P. H. Carstensen, K. Schmidt, Computer supported cooperative work: New challenges to systems design, in: In K. Itoh (Ed.), Handbook of Human Factors, Citeseer, 1999, pp. 619–636.

[41] R. Ashkenas, There's a Difference Between Cooperation and Collaboration, Harvard Business Review.

[42] S. Houben, C. Golsteijn, S. Gallacher, R. Johnson, S. Bakker, N. Marquardt, L. Capra, Y. Rogers, Physikit: Data engagement through physical ambient visualizations in the home, in: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, ACM, 2016, pp. 1608–1619.