Poster: System Architecture for Programmable Connected Devices

Maria Laura Stefanizzi*, Luca Mottola+, Luca Mainetti*, Luigi Patrono*
*University of Salento (Italy), +Politecnico di Milano (Italy) and SICS Swedish ICT
Contact author: laura.stefanizzi@unisalento.it

Abstract

We present a system architecture to enable running a slice of a mobile app's logic onto connected devices such as proximity beacons, body-worn sensors, and controllable light bulbs. These are normally black-boxes: their functionality is fixed by vendors and typically accessed only through low-level APIs. We overcome the limitations of this design by providing a generic programmable substrate on the connected device. Mobile apps can dynamically deploy arbitrary *tasks* implemented as loosely-coupled actor-like components. The underlying run-time support takes care of the coordination across tasks and of their real-time scheduling. Our current prototype indicates that our design is not only feasible, but incurs in very limited system overhead.

1 Overview

Motivation. Wireless embedded sensing and personal mobile computing are blending. Emerging wireless technologies such as Bluetooth Low Energy are backing this trend by enabling seamless data exchange between energy-constrained embedded devices and, for example, mobile phones. As a result, networking stacks and interoperability frameworks [2] appear to implement applications using *connected devices* such as proximity beacons [4], body-worn sensors [10], and controllable light bulbs [5].

Most such efforts share the assumption of the connected device as a black-box. Manufacturers ship the devices pre-programmed with a generic low-level API that mainly enables extracting raw data from sensors and/or controlling basic actuator functionality. Changes to the on-board software are limited to firmware updates shipped by the manufacturers to improve protocol compatibility or to patch security flaws.

Such a state of affairs entails that: *i)* even the simplest functionality requires *intense wireless interactions* across mobile and connected devices, affecting resource consumption; *ii)* mobile apps need to be developed based on *vendor-specific APIs*, preventing portability and seamless operation in heterogeneous settings; *iii)* app functionality are limited to the *time of actual wireless connection*, that is, disconnected operations are fundamentally hampered.

Approach. To overcome these limitations, we design and implement a system architecture that allows connected device to *i)* run arbitrary app logic in an on-demand fashion, and *ii)* host data on behalf of applications.

The problem strikingly differs from traditional sensor networking. Applications are supplied by third parties. Their characteristics, such as processing load and memory requirements, are difficult to anticipate. Multiple such applications with distinct requirements may need to operate concurrently. Processing is also expected to be largely event-driven; for example, being dictated by on-the-fly connections of mobile devices, rather than occurring periodically. These aspects shape the system challenge into a new form.

Our answer to this challenge rests upon two pillars: a custom programming model and a dedicated run-time support that replaces the existing device firmware.

1.1 Programming Model

To tame heterogeneity and vendor lock-in, we shift the interactions across mobile and connected devices to data rather the devices themselves. Central to this is the notion of *task* as a programmer-defined relocatable slice of app logic. In a fitness app, for example, programmers may define a task that computes burned calories based on the range of sensors available on modern fitness trackers. The mobile phone may opportunistically deploy such a task on the fitness tracker itself, so to limit the data exchanges to a single semantically-rich quantity rather raw data.

To enable interactions among functionality supplied by different parties, tasks are decoupled w.r.t. each other both in time and data, in an actor-like fashion [1]. Tasks are fully defined by the data they consume, the processing they perform, and the data they produce. Their processing is meant to be entirely reactive. For example, we discourage the use of long-running threads whose fair scheduling may become difficult in the setting we target. The input data of a task may come from the sensors aboard the device, or be the result of a different task. For example, a health-monitoring app may employ the burned calorie information of the fitness app to augment the long-term time analysis it performs. Tasks that

International Conference on Embedded Wireless Systems and Networks (EWSN) 2016
15–17 February, Graz, Austria
© 2016 Copyright is held by the authors.
Permission is granted for indexing in the ACM Digital Library
ISBN: 978-0-9949886-0-7

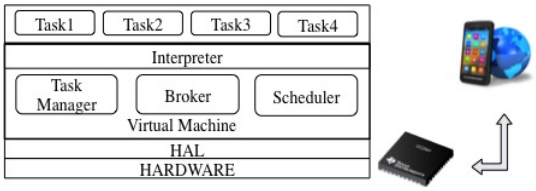

Figure 1. Overall architecture.

provide vendor-independent interaction paradigms and data formats can be similarly developed.

The execution of tasks is also decoupled from the connection to the mobile device. Tasks may, for example, reside on the connected device also whenever the mobile device that originally deployed them moves away, enabling disconnected operations. The same applies also to data. Tasks may produce data to be consumed by any mobile device, as long as a matching app is running on the latter. Data resides on the connected device according to programmer-defined criteria, rendering the device itself a general-purpose environment-immersed data store. This is useful, for example, when implementing pervasive games [8].

1.2 Run-time Support

Fig. 1 shows the architecture of the run-time support underpinning the whole task lifecycle. It is composed of four components: *i)* a virtual machine (VM) layer, *ii)* a broker, *iii)* a scheduler, and *iv)* a task manager.

The *VM layer* is in charge of the concrete execution of dynamically deployed tasks. Using a VM rather than some form of run-time binary linking has pros and cons. A VM detaches the implementation of tasks from the specific hardware. Further, a VM lessens the programming burden by supporting high-level languages such as Python and Java. On the downside, VMs suffer some unavoidable processing overhead due to code interpretation. Given the heterogeneous setting we target, the non-safety critical nature of most mobile apps, and the availability of powerful MCUs with reduced energy consumption, we argue that the use of a VM brings more advantages than disadvantages.

The *broker* acts as a single intermediate point between tasks and the data they consume or produce. Data is output by physical sensors or by existing tasks along given *topics*. For example, "burned calories" may represent a topic for data output by the fitness app task, as much as "acceleration" is the topic that accelerometers employ for raw data. The broker matches data appearing on certain topics with the topics taken as inputs by existing tasks, triggering their execution. Topics are defined by tasks when installed on a device; each task carries a *manifest* that indicates the topics it consumes, the ones it produces, the rate of probing physical sensors if necessary, and the number of output values the tasks wants to make reside on the device.

The broker may, in principle, trigger several tasks concurrently if they all consume the same topic. The question is then how to schedule these tasks. Our *scheduler* implements an Earliest Deadline First (EDF) [7] policy. Information on the absolute deadline of a task is part of the manifest. We choose an EDF policy because of its real-time optimality [6], which entails that if a schedule able to meet all task deadlines exist, EDF definitely finds one. The potential issue in applying EDF is, however, its processing overhead. We argue this should not pose scalability problems in our setting, given we are generally not expecting an extremely large number of tasks to be simultaneously triggered.

Finally, a *task manager* takes care of the run-time deployment of tasks, handling operations such as the reception of the task bytecode over the network and its un-installation upon request from a mobile device.

Table 1. Program memory and RAM requirements.

Component	Program memory [Bytes]	RAM [Bytes]
Broker	8456	876
Scheduler	3912	584
Task manager	112	564
Complete system	154516	10076

2 Ongoing Work

Our prototype targets the STM NUCLEO-F091RC board, equipped with an ARM Cortex M0 MCU. To enable the communication to/from a mobile device, we attach an X-NUCLEO shield that provides BTLE 4.0 connectivity. We realize custom implementations of broker, scheduler, and task manager. We port PyMite [9], a reduced Python interpreter, as VM support. We choose Python because, unlike languages such as JavaScript, it compiles to bytecode, which reduces network traffic when deploying tasks. Moreover, Python supports multiple programming paradigms, including object-oriented, imperative, and functional.

Early results demonstrate the feasibility of our design with limited overhead. Table 1 reports on the memory overhead of the custom implementations of broker, scheduler, and task manager. The majority of the overhead is currently due to the PyMite port, which is however only meant to provide a working interpreter. Even with this limitation, our prototype leaves about 69% (41%) of RAM (program memory) available. The processing slow down due to interpretation is inline with existing results [3].

We plan to carry out a full assessment of the flexibility of our design against diverse requirements, and a careful system evaluation. We also intend to investigate how to arbitrate resources among different tasks. Next, we plan to conceive a dedicated data model to organize topics. This way, an app may dynamically gain knowledge on what data is possibly already produced by currently running tasks.

3 References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. PhD thesis, MIT Artificial Intelligence Laboratory, 1985.
[2] Allseen Alliance. Alljoin. allseenalliance.org.
[3] N. Brouwers et al. Darjeeling, a feature-rich vm for the resource poor. In *ACM SENSYS*, 2009.
[4] easiBeacon. ibeacon. www.easibeacon.com.
[5] Flux Smart Lighting. Smart LED light bulb. bluetoothlightbulb.com.
[6] M. Kargahi and A. Movaghar. A method for performance analysis of earliest-deadline-first scheduling policy. *The Journal of Supercomputing*, 37(2), 2006.
[7] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1), 1973.
[8] L. Mottola et al. Pervasive games in a mote-enabled virtual world using tuple space middleware. In *ACM NETGAMES*, 2006.
[9] PyMite. code.google.com/p/python-on-a-chip/.
[10] Xiaomi. Mi band. www.mi.com.