Fabryq: Using phones as smart proxies to control wearable devices from the Web

Mozziyar Etemadi
Will McGrath
Shuvo Roy
Björn Hartmann

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-134
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-134.html

June 12, 2014

Copyright © 2014, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA. Additional support was provided by a Sloan Foundation Fellowship and a Google Research Award.

fabryq: Using phones as smart proxies to control wearable devices from the Web

Mozziyar Etemadi[1,2] **Will McGrath**[1,3] **Shuvo Roy**[2] **Bjoern Hartmann**[1]

[1]UC Berkeley SWARM Lab
490 Cory Hall, Berkeley, CA
{mozzi,bjoern}@berkeley.edu

[2]UCSF Bioeng. & Ther Sci
1700 4th St., SF, CA
shuvo.roy@ucsf.edu

[3]Stanford CS
353 Serra Mall Stanford, CA
wmcgrath@stanford.edu

ABSTRACT

Wearable ubiquitous computing devices are often size- and power-constrained, which prevents them from directly connecting to the Internet. A common pattern is therefore to interpose a smart phone as a router and to deliver graphical user interfaces for such hardware. However, implementing the entire pipeline from embedded device through a phone to the Internet and back requires a disjoint set of languages and APIs accessible only to experts. In this paper, we present fabryq, a new platform that handles the complexities of creating such applications. fabryq is especially aimed at supporting field deployments of prototype ubicomp hardware, e.g., for new interactive health devices. fabryq turns a smartphone into a bridge that connects the short range wireless technology of Bluetooth Low Energy (BLE) with our cloud service via the Internet. We introduce a *protocol proxy* programming model to find and control peripheral devices from Javascript; and describe a *UI pushdown* technique to render user interfaces on phones within reach of peripheral devices. To illustrate the utility of our platform, we also implement μfabryq, a breadboard prototyping platform similar to Arduino with functionality exposed over a JavaScript API built exclusively with fabryq.

Author Keywords

Toolkits; ubiquitous computing; swarm devices; prototyping.

ACM Classification Keywords

Human-centered computing– User interface toolkits

INTRODUCTION

In the predominant vision of ubiquitous computing, all kinds of devices, from large to small, become smart and networked. One important class of ubiquitous computing devices are small, wearable sensors — for example those used in medical and fitness applications. Unfortunately, these cannot just be connected to the Internet (e.g., via WiFi) because of size and power constraints. In practice, therefore, wearable ubicomp devices are often constructed using a three-level architecture consisting of: 1) a very energy efficient, embedded low power device with a short range radio; 2) a user's mobile phone, which shows a user interface but also acts as a router from body-area networks to the Internet; 3) server code for aggregating data and reasoning across multiple users and devices. We refer to this such applications as *MPC* (eMbedded–Phone–Cloud) apps (see Figure 1). For example, the FitBit fitness tracking device monitors a user's motions and periodically relays information to a companion application running on the user's mobile phone (or PC), which in turn communicates with servers that the FitBit company maintains. Building and maintaining such multi-language, multi-platform distributed systems is complex, error-prone, and requires skills in several diverse fields. Thus, experimentation in deployable, mobile wearable devices is largely reserved to experts, and implementation cycles are long and complex, which prohibits rapid prototyping. While research has introduced prototyping toolkits that significantly increase the speed of design explorations [9, 8, 18], these toolkits often make power or connectivity tradeoffs that restrict their use to lab settings or stationary, plug-in products.

In response to these issues, we introduce fabryq, a framework that facilitates the creation of new wearable Ubicomp devices by handling the complexities of creating new mobile device, server, and networking code. Specifically, fabryq takes the form of a mobile application and cloud service. The mobile application turns an ordinary smartphone into a bridge that connects the short range wireless technology of Bluetooth Low Energy (BLE) with our cloud service via the Internet. We chose BLE because it is the single short range wireless technology that is ubiquitously available on modern smart phones and can thus be widely employed. fabryq applications are written in Javascript and run in a web browser. fabryq introduces a *protocol proxy* programming model — developers write BLE protocol calls in Javascript as if the target device were locally connected and always available. The fabryq architecture then finds a mobile phone within radio reach of the target BLE device; passes the command(s) through the phone to the target device, and returns data to a the web application. This allows the creators of new devices to focus on writing the devices' firmware and creating new applications with the data from the devices, rather than writing complex and error-prone networking code (see Figure 2).

UC Berkeley EECS Technical Report, June 2014.

Figure 1. An example MPC application distributes logic and user interaction across embedded device, mobile phone and a cloud server.

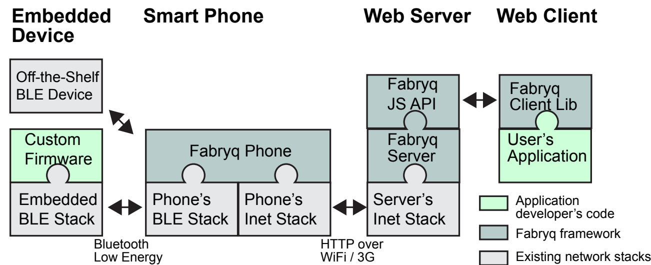

Figure 2. With fabryq, developers use off-the-shelf bluetooth devices or write firmware that exposes devices' inputs and outputs over Bluetooth; and high-level application logic in Javascript. The fabryq framework manages finding the desired hardware device and a mobile phone within range, and handles all message marshaling.

Our approach deals gracefully with situations where a BLE device may move into and out of the range of a mobile client device. The star network topology of BLE guarantees deterministic execution of commands in the correct order. fabryq handles the connection management and networking that make this abstraction possible in the background. User interfaces for fabryq applications such as visualizations of collected data are also authored in JavaScript, and they can be shown either on the web, or inside the fabryq application on a mobile device. For this purpose, fabryq introduces a *UI push-down* model where embedded data updates on the server can trigger the display of interfaces on the phone that was responsible for collecting the data.

fabryq makes it easier for developers to work with both existing off the shelf BLE devices and custom devices of their own design. To illustrate fabryq's ability to interface with off-the-shelf devices, we demonstrate a heart rate visualization application from an off-the-shelf BLE heart rate monitor. To demonstrate the robustness of fabryq in developing applications for use with highly custom BLE devices, we present µfabryq, a custom BLE device paired with a JavaScript API written on top of fabryq that make some of the most useful features of embedded processors such as analog to digital converters (ADCs), interrupts, digital input/output pins (GPIO), pulse width modulation (PWM), and a serial peripheral interface (SPI) available to web programmers via fabryq. This firmware mirrors the programming model of popular "maker" platforms such as Arduino, but offers the benefit that applications can be distributed across many wearable devices in many locations. In order to both validate fabryq and demonstrate its utility, we will describe the devices created by students using µfabryq during a hackathon.

RELATED WORK

fabryq relates to prior work in ubicomp prototyping toolkits, research on working with sensor data on mobile phones, experience logging, and "Internet of Things" networking.

Ubicomp Prototyping

HCI research has contributed systems for rapid prototyping of Ubicomp devices and systems [8, 9, 4, 18]. While some focus exclusively on self-contained interactions, others such as .NET Gadgeteer [18] and Shared Phidgets [15] explicitly offer network connectivity to create Internet-connected devices. However, these devices tend to be tethered and not optimized for power consumption, so they cannot easily be deployed in mobile scenarios outside the lab. We focus on sensing systems that are easy to write and prototype with, yet can be given to users without supervision and deployed in real world scenarios for weeks to months at a time.

Connecting Sensors to Mobile Phones

A number of projects aim to make it easier to connect external devices and sensors to smart phones and use them in applications. iStuffMobile [2] augments existing phones with new sensors – for development speed, sensor mapping logic runs on a nearby desktop computer, limiting deployment options. Amarino [12] allows designers to access events occurring on a mobile phone from an embedded platform. HiJack [13] can power and exchange messages with an embedded microcontroller through a phone's audio jack. Open Data Kit Sensors [6] is an application framework that introduces abstractions to simplify the connection of multiple sensors with different communication channels and APIs (e.g., wired and Bluetooth) to a single mobile device. Dandelion [14] generates both smart phone and sensor node binaries from a common source and then uses remote method invocation to call sensor code from the phone.

In contrast to these projects, fabryq uses the phone as a smart router to relay commands from a Web server to BLE devices. This allows developers to change their sensor querying code at any time without having physical access to the phone, and applications can span multiple phones. fabryq is also agnostic to which phone is connected to which sensor—only communication between server and sensor matters.

Experience Logging

Experience sampling and logging tools such as Momento [5] and MyExperience [7] aim to capture a user's daily experience "in the wild" outside of the lab. These goals are aligned with our focus of creating prototypes that can be deployed with users and monitored remotely for extended periods of time. MyExperience, for example, allows capturing of sensor data based on declarative rules; data is automatically synced to a server. The architecture supports addition of external sensors (e.g., Bluetooth devices) by writing drivers for a particular phone, but this is difficult. The principal difference is fabryq's focus on supporting custom peripheral devices.

IoT Networking

Connecting resource-constrained embedded hardware to Internet servers is also a concern of "Internet of Things" researchers. One way to provide IP packet support to low-energy embedded devices is through IEEE802.15.4 networks using "6LoWPAN" (IPv6 over Low Power Wireless Personal Area Networks [1]). Alternatively, devices such as the XBee Internet Gateway marshall traffic between a local area network and the Internet. The main difference to much of this work is that it presupposes additional networking infrastructure which is not generally available yet. We instead target ubiquitous smart phones and their data networks.

ElectricImp WiFi modules aim to lower the threshold for developing Internet-connected appliances [11]. Like fabryq, Imp uses a hosted server that handles many lower-level networking tasks. However, Imp devices require direct WiFi access—power requirements make it infeasible to use them for mobile, wearable deployments.

MOTIVATING APPLICATIONS

To inform the design of fabryq, we surveyed the emerging market of hardware smartphone "accessories." Fitness trackers such as the FitBit, Misfit Shine, or ActiveReplay's Trace as well as devices such as the Automatic vehicle data link tend to follow a similar pattern: they use MEMS sensors and minimal display on device and they a smart phone to display the main UI to the user. They also use web backends to store, process, or share data e.g., to allow users to compete on daily step counts. Our work is also motivated by a collaboration with medical researchers at a local medical center and their needs for wearable patient monitoring devices (see Figure 3). These devices generally require small physical size and weight but battery life on the order of weeks or months so they can be given to patients without requiring recharging or other management. Researchers want to send patients home with these devices and remotely track gathered sensor data at their institution. Such deployments may require fabrication of a few dozen identical devices. Accordingly, a framework should facilitate deployment of prototype code to multiple users.

Finally, we draw inspiration from experiences gathered teaching the design of integrated interactive hardware/software devices at our institution. While teams of motivated undergraduate and graduate students can create working prototype devices such as the ones shown in Figure 4, much of the implementation difficulty lies in working with multiple different networking technologies and protocols simultaneously; managing intermittent wireless connections; and doing this in multiple different programming languages on different platforms (embedded, mobile, web) with different conventions, data encoding schemes, etc.

Design Guidelines

Common patterns in our device survey (Figure 5) yielded the following design guidelines:

Smartphone as proxy A modern smartphone is the one piece of infrastructure that can be assumed to be present. Therefore, leverage the smart phone as a proxy from the local, body-area network to the Internet.

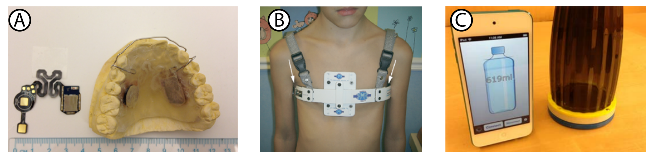

Figure 3. Three motivating wireless medical devices: A) A retainer that tracks wearer compliance using a built-in temperature sensor. B) A chest compression brace that tracks applied pressure over time. C) A water mug that tracks liquid consumption throughout the day.

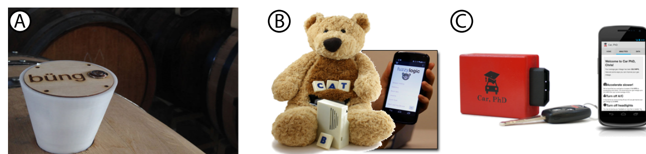

Figure 4. Example devices from our class: A) A wireless barrel gauge for distilleries and wineries; barrel readings are aggregated online. B) A toy to support literacy education; play statistics are collected on the phone and online. C) A car dongle that streams driving telemetry data to the phone and compares driving behavior on a central server.

	Embedded		Phone		Server	
	Sensing	Display	UI	Relay to cloud	Aggregation/ Reporting	Web UI
Consumer Devices						
FitBit	✔	✔	✔	✔	✔	✔
Medical Devices						
Retainer	✔	—	—	✔	✔	—
Smart Water Bottle	✔	—	✔	✔	✔	✔
Student Projects						
Barrel Gauge	✔	—	—	—	✔	✔
Driving Suggestions	✔	—	✔	✔	✔	✔

Figure 5. Features of some motivating examples. fabryq focuses on supporting embedded sensing, relaying data through a phone, and aggregating and displaying info on the Web (green columns).

Prototyping "in the wild" Enable prototyping of wearables that can be taken outside of the lab and that can run for at least 24 hours, but ideally weeks or months.

Multiple devices Facilitate development and management of multiple identical prototypes that can be distributed to a group of users (e.g., patients for feasibility studies)

Display Enable developers to show collected data and other user interfaces both on the web and on a phone.

Abstract networking details Shield developer from network connection management and data transfer details.

Lower threshold, high ceiling Enable users who have limited programming experience (e.g., clinical researchers) to write simple data collection scripts in a single language; enable experts to create new, complex devices.

Conversely we chose to avoid supporting the following cases:

No low-latency, high-throughput apps We focus on working with intermittently read sensor data where milliseconds of latency are not important.

No offline operation without a cloud server We target prototype deployment where a designer/experimenter is still in the loop; we do not target the scenario of BLE devices

talking to phone applications without code running in the cloud.

FABRYQ ARCHITECTURE

fabryq's core is an abstraction of the Bluetooth Low Energy (BLE) protocols. We first briefly introduce some BLE terminology necessary for explaining the architecture. We then introduce a scenario that demonstrates how a developer writes and deploys a fabryq-enabled application. We then describe the fabryq implementation that enables this workflow.

BLE fundamentals

BLE is a wireless protocol for communication between a *central device* (i.e., a mobile phone) and one or more *peripheral devices* (i.e., wearables). Peripheral devices expose *BLE characteristics* — short, named pieces of information (typically 1-20 bytes) similar to variables. Central devices can can perform three operations on characteristics: `GET`, `SET`, and `NOTIFY`. A `GET` signifies that the BLE central device would like to retrieve the contents of the characteristic from the BLE peripheral. A `SET` means that the BLE central would like to modify the contents of the characteristic. A `NOTIFY` signifies that if the value of the characteristic should change on the peripheral, the central would like to be notified of this.

Operations happen at a set *connection interval*, a precise time window when the central and peripheral have decided to communicate. Though it is beyond the scope of this work, the connection interval is critical to maintaining deterministic power consumption—a longer interval means less data can be communicated but less power is used. By adjusting the interval, a battery life of greater than one year can be achieved from a coin cell battery: one of the hallmarks of BLE peripherals.

The set of characteristics of a device is described in a Generic ATTribute profile (GATT). The peripheral device is also known as the GATT server; the central device as the GATT client. Characteristics are identified by universally unique identifiers (UUIDs)—four digit UUIDs are reserved by the Bluetooth Special Interest Group (SIG) for common use cases, and 128-bit UUIDs are used for custom, developer specific characteristics. Characteristics are further grouped by belonging to services, which are collections of characteristics. A particular BLE peripheral often contains multiple services, for example, a "device identification service" (`0x180A`) and a "heart rate service" (`0x180D`) would be typical of a commercially available heart rate monitor. Characteristics and the services they belong to make up the "characteristic tree." A more detailed description of BLE can be found in [10, 3].

Scenario: how to write and deploy a fabryq app

Developer facing fabryq: defining the application configuration

fabryq enables application developers to work with commercial off-the-shelf devices (with defined behavior exposed in 4-digit UUIDs) and novel, developer-defined devices (with custom firmware and custom characteristics that use 128-bit UUIDs). We now present the workflow for writing and deploying a fabryq-enabled web application in either case.

Object	Type	Instance
App	App Type	App Install
BLE Peripheral	Device Type	Virtual Device

Table 1. Types and instances in the fabryq application configuration interface. Instances are created in a "user portal" whereas types are created in a "developer portal." Types listed in italics are identified by either a standard (4-digit) or custom (128-bit) BLE UUID. Other types and all instances are identified by fabryq-specific identifiers.

Application developers must take two major steps: first, they define the BLE device requirements to run their application using the fabryq application configuration interface. Second, they then write application code (using the fabryq JavaScript API) that references this hardware configuration profile. For example, a hypothetical *HydrateForHeartHealth* (HFHH) application may track a user's liquid intake over the course of a day and correlate it with heart rate variability. It may require information about liquid level in a smart cup (a custom device also created by the developer) and data from a heart rate monitor (a commercial device).

fabryq application configuration requires developers to operate at a level of abstraction because the defined application should be able to run for multiple users and multiple devices (and, multiple applications may need to use data from the same device). In our running example, clinical researchers may want to give smart cups and heart rate monitors to two dozen patients—each patient's data will be collected by an instance of the HFHH application that communicates with particular hardware devices. In other words, while application developers define the hardware requirements to run their application (e.g., the application requires BLE devices that expose heart rate and temperature), it is only the end user who will associate their particular hardware device with the application (i.e., their smart cup and their commercial Heart Rate monitor). Therefore, in the fabryq architecture, developers specify abstract application types that require abstract device types. This is done in the fabryq application configuration interface and will now be explained in more detail.

Our HFHH developer, knowing that she would like her application to use both an off-the-shelf heart rate monitor and a custom smart cup that she has also made, begins the development process by entering the information for her fabryq application. Identification in the application configuration interface is performed using a two-tiered system of "types" and "instances" which are listed in Table 1. Specific elements of application configuration are as follows. Device types are a collection of service types and characteristic types (the characteristic tree discussed previously, but abstracted as to not imply a particular physical instance of a device). Analogous to device types, application types are containers of device types—they define the peripheral requirements for an application. These relationships are depicted in Figure 6.

With these definitions in mind, our HFHH developer enters the heart rate monitor into the fabryq application configuration interface as in Figure 7. She creates a new device type and identifies it as a generic branded "Cheap HR monitor." Next, she links the heart rate service type and heart rate characteristic types (both BLE standards) to the device

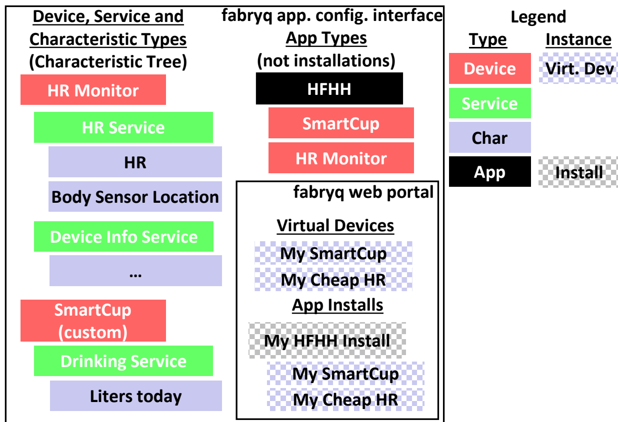

Figure 6. Overview of fabryq application configuration. Characteristic types are contained inside service types, which are contained inside device types. Application types are defined by a list of device types they require. Virtual devices are real-world realizations of device types; application installs require one virtual device for every device type listed for its application type.

type. She then repeats the process for her smart cup, this time (not shown) first defining custom service and characteristic types, then linking it to her smart cup device type. While these custom services and characteristics depend entirely on the firmware running on the smart cup, an example custom service may be a "water consumption" service with a characteristic of "liters drank today."

She now proceeds to add the application type, named HFHH. In a similar interface to the device type tree (not shown), the application type is defined and linked to device types. In this case, the HFHH application type is linked to the "Cheap HR Monitor" type and the "Smart Cup" type. The application is now fully configured in fabryq and our developer is ready to write code that interacts with the two device types she has required. She employs the fabryq JavaScript API (details of which are discussed in a subsequent section) to interact, through fabryq, with the server's virtual representation of the required peripherals as if they were "always" connected to the end user's web browser. fabryq will queue her commands and push them down to the actual devices when they are available.

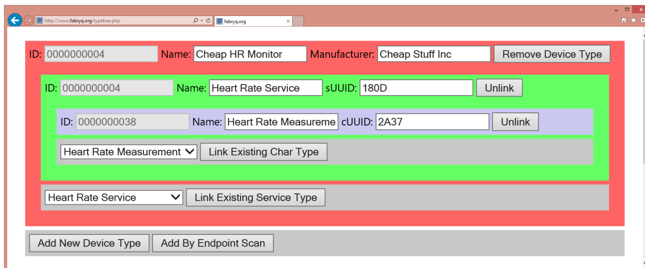

Figure 7. fabryq application configuration interface: characteristic types, service types, and device types can be defined and linked together. Here, a developer defines a Heart Rate Monitor device type (red) which exposes a standard Heart Rate service (green) and characteristic (blue), for use in their application. Data are stored in a simple MySQL backend.

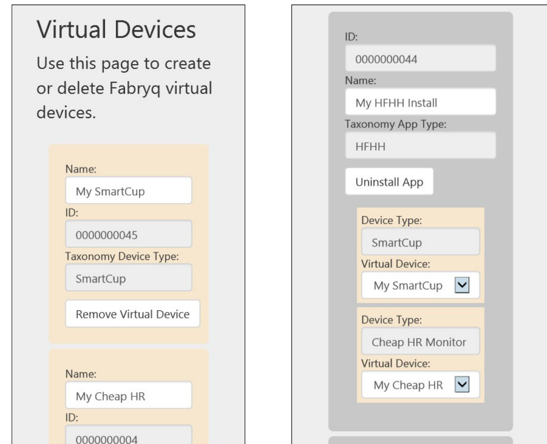

Figure 8. Screenshots of the fabryq web portal. (left) the user indicates they have devices of type "Cheap HR monitor" and "SmartCup" by creating virtual device of those types. (right) The user installs the HFHH app and associates it to a virtual device.

User facing fabryq: unboxing and running the application

A user of a fabryq-enabled web application must perform three major steps: first, they must install the "fabryq mobile" iOS application on all of the iOS devices they desire to use to "connect" their BLE peripherals to fabryq. This step happens once per iOS device, regardless of the number of fabryq-enabled applications being run. Second, they must indicate possession of their peripherals and "install" the web applications they desire: both of these actions take place on a website referred to as the "fabryq web portal." Finally, they must "run" the web application by navigating to the developer's web application URL. Upon the first run of the web application, all iOS devices that have never connected before to the peripheral of interest will have to connect with and register the id of the target peripheral, through user interaction in the fabryq mobile application. Steps two and three of the above summarized process are now explained in more detail.

Following our example scenario, a user has received the developer's smart cup and a generic heart rate monitor, and would like to use the developer's application for the first time. First, the user will "install" the HFHH application on the fabryq web portal (Figure 8). Next, the user will indicate that he or she owns the peripherals by adding them in the virtual devices tab. Then, they will associate those devices with the new application install, indicating to fabryq that when they "run" this instance of the HFHH application, they would like it to interact with those two exact peripherals.

Next, assuming fabryq mobile is already installed on the user's iOS device(s), the user points his or her web browser[1] to the developer's application. The very first time the application is run, an identification request is made to the new virtual device created when the user added their new device on the web portal. The user's fabryq mobile application, seeing this request and knowing that they have never physically

[1] In our current implementation, the web browser must not be on the same phone the one with fabryq mobile installed, because fabryq mobile runs in the foreground. This is addressed in the limitations section.

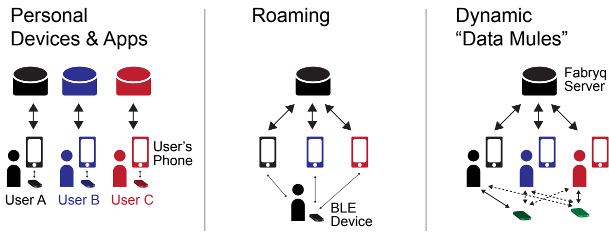

Figure 9. Possible application configurations enabled by Fabryq: A) single user, single device. B) A user's peripheral accesses fabryq through more than one mobile device in a "roaming" pattern C) Users who agree to jointly run an application can also act as "data mules" and pick up data from environmental sensors whenever they walk by such a sensor.

connected to this virtual device before, will initiate a BLE scan and query the user to identify which physical peripheral corresponds to this virtual device. For every instance of the fabryq mobile app, once the user selects the peripheral, subsequent requests from fabryq will pass through directly to the peripheral without user intervention. The details of this implementation are depicted in Figure 10 and discussed in more detail in a subsequent section.

In the simplest case, one application requires one peripheral and the user has one (or more) iOS devices (each running fabryq mobile) that allow data to flow from application to peripheral (Figure 9A). One application may also allow the user to "roam" with their BLE peripheral, periodically coming into contact with one of their multiple iOS devices, as in Figure 9B. Finally, users may opt to "share" their iOS device's BLE radio with other users. In this configuration, other users would be able to connect their peripherals to fabryq using iOS devices that are not in their possession. As a simple example, a cyclist's heart rate monitor could connect to his heart rate application while he cycles without an iOS device, provided he comes in contact with other user's devices. Depicted in Figure 9C, this "crowd sourced BLE internet access" has been implemented in specific consumer applications such as Tile [2].

Having presented an overview of the workflow to write and run a fabryq-enabled web application, we will now discuss the specifics of the fabryq implementation.

fabryq Implementation

The function of fabryq is analogous to that of an IP router: fabryq commands (`GET`, `SET`, `NOTIFY`) are requested of the server by the JavaScript application code running in the browser and "routed" to the appropriate BLE peripheral through the various fabryq mobile apps (installed on iOS devices), based on which user is currently running the application and what virtual devices they have associated with that application install. Beyond routing BLE commands, fabryq also has the functionality of routing a user interface to a mobile device connected to a BLE peripheral. The main challenge in implementing such a router is overcoming three primary limitations of the bluetooth low-energy protocol:

1. **Inconsistent identification of a user's peripheral devices** across all of their mobile devices (e.g., using one heart rate monitor with several phones and/or tablets). In the case of iOS (the only platform that we have so far used to implement the fabryq mobile app), a unique identifier is generated for every bluetooth peripheral *for every iOS application*, that is to say, three applications connecting to the same peripheral (even on the same device) will have a different unique identifier for that peripheral. This behavior allows some high security applications like car keys and bank cards but presents a major difficulty in managing peripherals across applications and devices.
2. **Poor correlation of command requests with responses**. In all current BLE central implementations, callbacks from `GET`, `SET`, and `NOTIFY` commands are shared and can return in a different order than their initiating calls. While these callbacks do contain the characteristic UUID that is being queried, if one is repeatedly calling `GET`, `SET`, and/or `NOTIFY` on a single characteristic, maintenance of proper callback order becomes a formidable task for the developer that must be reimplemented in a highly custom, characteristic-specific manner for each application.
3. **Redundant calls to rediscover characteristics**. In all current BLE central implementations, the characteristic tree must be traversed over the wireless link, down to the characteristics of interest *on each connection.* That is to say, on each connection, even if a characteristic is known to exist in a particular service, the service and characteristic must first be "discovered" over the wireless link prior to executing one of the three BLE commands. Ideally, *a priori* knowledge of the characteristic tree is known by the mobile application to ensure minimum traversal of the tree, which readily leads to non-portable "hard coding" of characteristics. The alternative is power inefficient, over-traversal of the tree. We address this BLE limitation in fabryq by downloading the tree from the application configuration, ensuring consistent, minimum traversal.

The fabryq mobile application and its JavaScript API have been implemented specifically to overcome these limitations.

fabryq mobile Implementation

The primary function of the fabryq mobile application is to execute commands (i.e. `GET`, `SET`, `NOTIFY`) requested of characteristics of virtual devices. Because of BLE's peripheral identification challenges, the fabryq mobile app must know which physical device corresponds to which virtual device **for every mobile device**.

To overcome this limitation we have devised the following control flow, shown in Figure 10, resulting in minimal extra user interaction with no additional developer code. Commands (i.e. `GET`, `SET`, `NOTIFY`) to be performed on virtual devices are placed in a server-stored "command log" by calls to the fabryq javascript API (detailed in the next section). If commands are found for the user's virtual devices, the fabryq mobile application first determines if it has an OS-specific BLE-identifier for that virtual device. If it does, the action is performed and the result returned to the command log. If not, and there are nearby, unknown peripherals that are exposing

[2] http://www.thetileapp.com

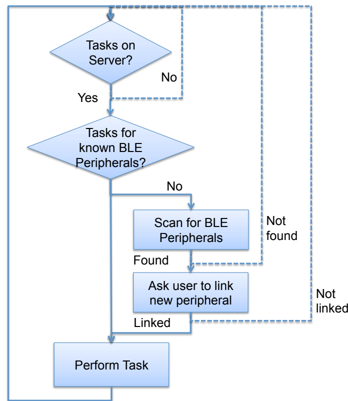

Figure 10. Control flow for fabryq mobile. If commands (i.e. GET, SET, NOTIFY) are found for the logged-in user's virtual devices, fabryq mobile first determines if it has an OS-specific BLE-identifier for that virtual device. If such a "link" between OS-specific identifier and virtual device exists, the action is performed and the result returned to the database. If not, and there are nearby, unknown peripherals that are exposing the characteristic type to be acted upon by the task, the user will be queried with the "friendly name" of the virtual device obtained from the application configuration, and a list of nearby BLE peripherals. If the user indicates one of these peripherals is a match, it will be stored permanently and the action performed.

the service type needed by the task, the user will be queried with the "friendly name" of the virtual device obtained from the application configuration, and a list of nearby BLE peripherals. If the user indicates one of these peripherals is a match, it will be stored permanently (on the same SQL server as the command log) and the action will be performed.

By decoupling the act of requesting `GET`, `SET`, and `NOTIFY` commands from the act of performing them, we also address the other two BLE-specific limitations: excess traversal of the characteristic tree and ordering of commands. The former is prevented because the for a given device type in the fabryq application configuration, the entire characteristic tree is known prior to the iOS device establishing a connection, without any "hard coding" of the UUIDs. Thus, upon connecting to a device, the bare minimum traversal of the characteristic tree automatically takes place, and the characteristics are available for direct query.

The latter is addressed by the nature of the design: the command log itself is in execution-order. Each time fabryq mobile is to return a result from an action, it only has one place to do it: in the correct spot in the log. As mentioned earlier, a limitation to BLE API callbacks from `GET`, `SET`, and `NOTIFY` commands on mobile phones is that they happen in an undetermined order and are often fixed to be the same callback function. Reconciliation of these callbacks with the appropriate row in the log table represented both the most formidable challenge and greatest benefit in developing fabryq, and in fact is a defining feature of the JavaScript API.

Additional fabryq mobile command: `SHOWURL`

In some instances, the developer would like to interact with the user on the fabryq mobile application directly as opposed to the web browser. For this purpose, we created a fourth fabryq command without a BLE counterpart: `SHOWURL`. Like the other commands, `SHOWURL` requires that one of the user's fabryq mobile applications be connected to the particular peripheral. However, instead of passing a command over BLE, it shows a URL on the fabryq mobile application itself: we term this *UI pushdown*. In this way, peripheral-specific UI can appear on the mobile device hosting it. Since `SHOWURL` is simply opening a web page, the full complement of fabryq JavaScript API calls are also available.

Implementation of fabryq javscript API

We have implemented a JavaScript API to enqueue and dequeue commands and poll their results. These commands are intended for BLE peripherals but originate on the cloud: our *peripheral proxy* model. Each `GET`, `SET`, or `NOTIFY` has an explicitly defined callback that is passed to the originating JavaScript function call (and per common JavaScript coding practice can be defined in-line). A demonstrative use-case for the API is as follows.

Our example developer would like to access the heart rate characteristic of an off-the-shelf BLE heart rate monitor for part of her HFHH application. Having already entered in the application configuration as described earlier, she begins to write JavaScript code. She first obtains the fabryq application profile object using an AJAX query. The fabryq application object contains a list of all virtual devices accessible to the application, indexed by device type. Recall that this was a major design requirement: the developer does not need knowledge of particular, physical devices, but only their type. In this example, she would like her application to display the current user's heart rate on the screen using a `GET` command. Using fabryq's JavaScript API, this is one function call with embedded callback lambdas:

```
add_get_action(
  fabryq_object[_DEVICE_TYPE],
  _HR_SERVICE,
  _HR_CHARACTERISTIC,
  _POLL_INTERVAL,
  _TIMEOUT,
  function onCompleted(action)
  {
    if(action.result=='null') {
      //update the screen to indicate the
      //heart rate monitor could not be located
    }
    else {
      //update the screen with
      //action.result (the heart rate)
    }
  },
  function onQueued(action) {},
  install_id
);
```

The `_DEVICE_TYPE`, `_HR_SERVICE`, and the `_HR_CHARACTERISTIC` constants are already known by the developer as she recently entered them into the fabryq application configuration interface. `_POLL_INTERVAL` defines how often and `_POLL_TIMEOUT` defines how long the command log will be polled by the browser for the result of the action. A timeout indicates that the peripheral could not be found by any of the user's fabryq mobile applications. The `onCompleted` callback occurs after the command has been successfully performed or timed out. The `onQueued`

Figure 11. Demonstrative fabryq application showing the user's heart rate obtained from an off-the-shelf BLE heart rate monitor and automatically plotted on the currently connected fabryq mobile application.

occurs after the command has been placed in the command log—this is useful in cases where a number of commands must be executed in a particular order, but subsequent commands do not depend on the results of preceding commands, thus they can be immediately queued. The `install_id` is populated when the user logs into fabryq and runs the application. `SET` and `NOTIFY` commands are similarly implemented and not discussed here.

The principle limitation of the JavaScript API is the requirement to poll the command log to discover the status of an action. This is a limitation of our command log implementation: a SQL table with simple REST interface. Trading off simplicity and portability for latency, future implementations of the command log interface could utilize WebSockets or other push notification schemes to allow for instant callbacks to JavaScript as soon as fabryq mobile has interacted with the BLE peripheral and updated the log.

The core fabryq implementation was written in about 4500 lines of code split across iOS (fabryq mobile); PHP and SQL (fabryq server) and Javascript (fabryq API).The heterogeneity of the codebase exemplifies the complexity of development that fabryq seeks to overcome.

EXAMPLE APPLICATIONS

We have created several example applications to demonstrate working with both off-the-shelf and custom peripherals.

Example with off-the-shelf BLE device

As a demonstrative example, we created a simple fabryq application that can plot the heart rate of a user using any off-the-shelf heart rate monitor that exposes the standard BLE heart rate service in real time. In addition to showing the heart rate on the user's web browser, the application uses the `SHOWURL` fabryq command to also display the plot on the fabryq mobile application currently communicating with the heart rate monitor (Figure 11). The completion of the first functional prototype of this example application took under three hours, the majority of which was spent on the construction of the visualization. Conversely, obtaining the heart rate data in fabryq required only two function calls, one to start collecting the heart rate data and a second to query the server for the history of the user's heart rate.

End-to-end use case: µfabryq

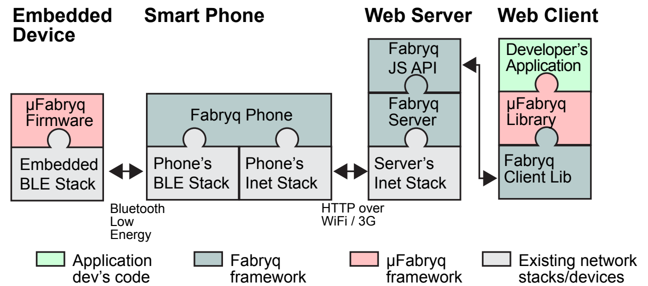

Figure 12. µfabryq offers direct access to embedded peripheral pins in a manner similar to the popular Arduino platform. It was implemented using Fabryq and shows the expressivity of the framework. It comprises a BLE firmware and a JavaScript library written using the Fabryq JS API.

fabryq JavaScript API	
Function	**Description**
add_get_action	Request a BLE characteristic
add_set_action	Set the value of a BLE characteristic
add_notify_action	Receive notification of a changed BLE char.
add_showurl_action	Show a URL on the connected fabryq mobile app.
µfabryq JavaScript API	
Function	**Description**
DigitalRead	Read a pin's binary value
DigitalWrite	Set a pin's output voltage to low or high
PinMode	Set a pin to input (w/ pullup) or output
AnalogRead	Read a pin's input voltage using an ADC
AnalogWrite	Set a pin's output voltage using PWM
AttachServo	Enable servomotor control on a pin
ServoWrite	Set a pin's servo to a particular location
AttachInterrupt	Attach a JS function to a pin interrupt
SPIbegin	Enable/configure the SPI on µfabryq
SPItransfer	Perform full-duplex SPI communication

Table 2. Function list for the microfabryq JavaScript API (top) and the µfabryq JavaScript API (bottom).

To demonstrate fabryq's ability to simplify communication with novel BLE devices, we created a customized BLE device with functionality resembling that of an Arduino [16]. µfabryq, shown in Figure 13, is a breadboard-able bluetooth system-on-chip based on the BlueGiga BLE113 module. We developed custom firmware that allows Arduino-like commands to be used over BLE. For example, there is a custom BLE characteristic that controls the output of all GPIO pins (high or low) and another that controls the pin direction (input or output). Using only the fabryq JavaScript API and jQuery, we then created the µfabryq JavaScript API which essentially maps Arduino commands to JavaScript functions, executing them over fabryq and returning their result to the browser. A list of these µfabryq API functions, along with the fabryq JavaScript API functions, is in Table 2.

Custom Devices

As a preliminary test of fabryq and µfabryq, the authors constructed multiple prototype devices. Among them was a security device built using a Pyroelectric Infrared (PIR) sensor and an accelerometer with a SPI interface. When the corresponding script is run in a web browser, it sends commands to the device to test and initialize the accelerometer, set an acceleration threshold via an SPI command, and set interrupts on the interrupt pin of the accelerometer and the output of

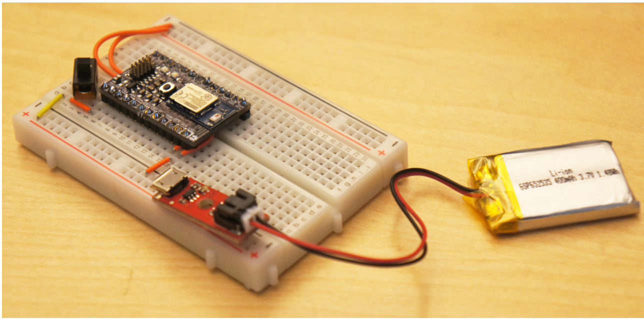

Figure 13. µfabryq breadboard provided to participants of the hackathon. The board contains a BlueGiga BLE113 breakout board, a microUSB battery charger, a LiPo battery and a power switch. The rest of the board is empty for prototypign.

the PIR sensor. Running this initialization from cloud to device takes roughly 15 seconds with some variability due to network and server latency. Whenever the sensors are triggered by a person walking by or moving the µfabryq board, the board pushes an update to the fabryq server, where it is polled by the web application. When the web application receives an interrupt message, it flashes a red screen. Latency from activation to browser display is about 3 seconds.

User Experiences with µfabryq

In order to evaluate and demonstrate fabryq's ability to connect to and transmit data from a number of peripherals, we held a 7-hour µfabryq hackathon. Attendees were expected to leverage prior Arduino experience to create novel BLE devices. Attendees were of various engineering backgrounds, namely, Bio-, Electrical, and Mechanical Engineeering as well as Computer Science. Twelve students in four groups were provided with a breadboard that contained a BLE module pre-loaded with the µfabryq firmware (see Figure 13). An ample supply of electronics and mechanical prototyping equipment was made available. All groups successfully downloaded and ran the GPIO demo and the majority collected sensor readings from a µfabryq board on a web server, all through two fabryq mobile applications placed in the room. Two groups went beyond small examples and built their own custom devices. At the conclusion of the hackathon, two groups remained and produced the following devices, seen in Figure 14.

Heart Rate Monitor: Students connected an off-the-shelf maker-focused photo plethysmography sensor[3] to an Arduino using the manufacturer-provided library to determine heart rate. As a simple way to bridge the 5V Arduino and 3.3V µfabryq board, the Arduino converted the heart rate into an analog voltage using PWM and an RC filter. This voltage was fed into an A/D pin on the µfabryq board, where it was converted to a digital value and sent to the browser through fabryq. Once on the web, the analog voltage was converted, in JavaScript, to heart rate and displayed in a webpage.

Tic Tac Toe: Students placed a grid of push-push, on-off switches and associated wiring on two breadboards and connected this to the µfabryq board via GPIO pins. The on-off

[3]http://pulsesensor.myshopify.com/

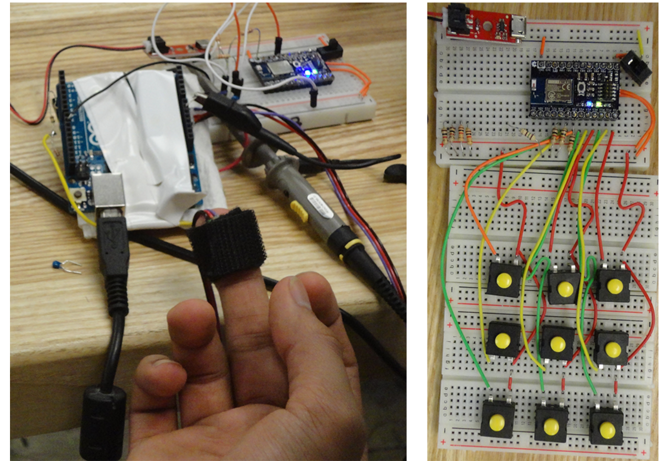

Figure 14. (left) Heart rate sensor managed by an Arduino, which is connected to a µfabryq board. (right) µfabryq breadboard wired to several pushbutton switches acting as an input device.

state of each switch was pushed to the web browser using fabryq, allowing for a tic-tac-toe board on the web that could be modified using the physical switches.

Qualitative feedback

The participants appreciated µfabryq's ability to simplify what would otherwise be a complex networking task. Instead of writing code for an embedded device, a mobile app, and a server, participants were able to write functioning applications using only JavaScript in the browser. One participant remarked, "It's really, really easy to use, especially compared to the complexity of what it accomplishes. I think the current API and the general spirit of the language's structure makes it really intuitive to use."

Participants spent most of their time designing and debugging the web user interfaces of their devices. Because few were expert web developers, the final complexity of their prototypes was limited. Nonetheless, we are encouraged by the results of the hackathon. Although the groups faced technical challenges in interfacing, web design, and debugging, they were all successful at communicating with the µfabryq boards. Additionally, when asked, "On a scale of 1 to 5 how likely would you be to use fabryq in your next project?" (where 5 was very likely), the participants responses were positive ($\mu = 3.6$).

The largest challenges that the participants faced involved understanding the status and failure situations of the back end infrastructure. This suggests that exposing more fine-grained system status when errors occur is an important usability consideration for future versions of the framework.

LIMITATIONS

fabryq has some limitations inherent in its architecture as well as shortcomings in the current implementation.

Design Limitations

Fundamentally, fabryq-enabled applications require complete a priori knowledge of all peripheral types prior to writing and/or running an application. This was explicitly chosen for robustness in connectivity across multiple mobile devices and

scalability of applications to many users and many peripherals. There exist a subset of applications where the peripherals are not known, for example, an application specifically designed to discover *any* nearby peripheral, or applications that interact with beacons that change their identifiers for security reasons. Such applciations are not supported.

Implementation Limitations

The current fabryq mobile application runs only on iOS, and only in the foreground. On iOS, there are stringent requirements placed on background applications, such as timing limitations. None of these requirements should fundamentally limit fabryq's current functionality, but we have not yet implemented a background mode.

An additional limitation is poorly–defined command latency. This is not a fundamental limitation; if the application configuration and command log were located on the mobile device, the command latency could be readily defined. Instead, our current implementation relies on polling to A) retrieve command logs for fabryq mobile and B) retrieve results of these commands by the JavaScript API. In future implementations, both of these polling mechanisms, which take place over REST-ful interfaces, could be replaced by TCP/IP socket schemes such as WebSockets.

Finally, in programming fabryq-enabled apps, the developer must currently edit the application configuration in a browser and write code in a local text editor. Testing this code requires opening the application in another browser window and having an iOS device running fabryq mobile physically near the peripheral and presumably the developer. This yields five points of interaction (two browser windows, a text editor, fabryq mobile, and the peripheral) for the developer which can become a challenge in and of itself, slowing down the development cycle. In future versions of fabryq, some of these points of interaction can be reduced or eliminated, e.g., by providing an integrated workbench that unifies writing, configuring and running applications, and by adding device simulation capabilities (e.g., through trace playback [17]).

CONCLUSION AND FUTURE WORK

This paper presented fabryq, a platform for rapidly writing and deploying *MPC* applications that use smart phones as proxies to control wearable devices from the Web. The development of fabryq was guided by a survey of commercial devices, medical wearable devices, and student projects. Two main techniques of fabryq are the *protocol proxy* model of executing BLE protocol calls from the Web; and *UI pushdown* to push interfaces from the Web to a phone. Future work on fabryq will focus on also making the firmware layer configurable or updatable from the Web. Just as characteristics and services have been abstracted into a virtual device and are available any time, future versions of fabryq will similarly absorb the firmware into the virtual device as part of a single application that runs on phone, cloud, and peripheral. Future fabryq applications will have a "complete picture" of the tasks to be executed on the peripheral, on the phone, and on the cloud. With this picture, individual tasks can be parceled out to the appropriate layers, depending on available network infrastructure, available hardware infrastructure, and the nature of the tasks requested.

ACKNOWLEDGMENTS

This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA. Additional support was provided by a Sloan Foundation Fellowship and a Google Research Award.

REFERENCES

1. 6LoWPAN. http://en.wikipedia.org/wiki/6LoWPAN.
2. Ballagas, R., Memon, F., Reiners, R., and Borchers, J. iStuff mobile: Rapidly prototyping new mobile phone interfaces for ubiquitous computing. In *Proceedings of CHI*, ACM (2007), 1107–1116.
3. Bluetooth Low Energy Specification Adopted Documents. `https://www.bluetooth.org/en-us/specification/adopted-specifications/`.
4. Buechley, L., Eisenberg, M., Catchen, J., and Crockett, A. The lilypad arduino: using computational textiles to investigate engagement, aesthetics, and diversity in computer science education. In *Proceedings of CHI*, ACM (2008), 423–432.
5. Carter, S., Mankoff, J., and Heer, J. Momento: Support for situated ubicomp experimentation. In *Proceedings of CHI*, ACM (2007), 125–134.
6. Chaudhri, R., Brunette, W., Goel, M., Sodt, R., VanOrden, J., Falcone, M., and Borriello, G. Open data kit sensors: Mobile data collection with wired and wireless sensors. In *Proceedings of the ACM DEV*, ACM (2012), 9:19:10.
7. Froehlich, J., Chen, M. Y., Consolvo, S., Harrison, B., and Landay, J. A. MyExperience: a system for in situ tracing and capturing of user feedback on mobile phones. In *Proceedings of MobiSys*, ACM (2007), 57–70.
8. Greenberg, S., and Fitchett, C. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of UIST*, ACM (2001), 209–218.
9. Hartmann, B., Klemmer, S. R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., and Gee, J. Reflective physical prototyping through integrated design, test, and analysis. In *Proceedings of UIST*, ACM (2006), 299–308.
10. Heydon, R. *Bluetooth low energy: the developer's handbook*. Prentice Hall, 2013.
11. Electric imp. http://electricimp.com/.
12. Kaufmann, B., and Buechley, L. Amarino: A toolkit for the rapid prototyping of mobile ubiquitous computing. In *Proceedings of MobileHCI*, ACM (2010), 291–298.

13. Kuo, Y.-S., Verma, S., Schmid, T., and Dutta, P. Hijacking power and bandwidth from the mobile phone's audio interface. In *Proceedings of ACM DEV*, ACM (2010), 24:124:10.
14. Lin, F. X., Rahmati, A., and Zhong, L. Dandelion: A framework for transparently programming phone-centered wireless body sensor applications for health. In *Wireless Health 2010*, ACM (2010), 74–83.
15. Marquardt, N., and Greenberg, S. Distributed physical interfaces with shared phidgets. In *Proceedings of TEI*, ACM (2007), 13–20.
16. Mellis, D., Banzi, M., Cuartielles, D., and Igoe, T. Arduino: An open electronic prototyping platform. In *Proceedings of CHI*, vol. 2007 (2007).
17. Newman, M. W., Ackerman, M. S., Kim, J., Prakash, A., Hong, Z., Mandel, J., and Dong, T. Bringing the field into the lab: Supporting capture and replay of contextual data for the design of context-aware applications. In *Proceedings of UIST*, ACM (2010), 105–108.
18. Villar, N., Scott, J., Hodges, S., Hammil, K., and Miller, C. .NET gadgeteer: A platform for custom devices. In *Proceedings of Pervasive*, Springer-Verlag (Berlin, Heidelberg, 2012), 216–233.