Code In The Air: Simplifying Sensing and Coordination Tasks on Smartphones

Lenin Ravindranath, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden
MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA

ABSTRACT

A growing class of smartphone applications are *tasking applications* that run continuously, process data from sensors to determine the user's context (such as location) and activity, and optionally trigger certain actions when the right conditions occur. Many such tasking applications also involve coordination between multiple users or devices. Example tasking applications include location-based reminders, changing the ring-mode of a phone automatically depending on location, notifying when friends are nearby, disabling WiFi in favor of cellular data when moving at more than a certain speed outdoors, automatically tracking and storing movement tracks when driving, and inferring the number of steps walked each day. Today, these applications are non-trivial to develop, although they are often trivial for end users to state. Additionally, simple implementations can consume excessive amounts of energy. This paper proposes *Code in the Air* (CITA), a system which simplifies the rapid development of tasking applications. It enables non-expert end users to easily express simple tasks on their phone, and more sophisticated developers to write code for complex tasks by writing purely server-side scripts. CITA provides a task execution framework to automatically distribute and coordinate tasks, energy-efficient modules to infer user activities and compose them, and a push communication service for mobile devices that overcomes some shortcomings in existing push services.

1. INTRODUCTION

Smartphones and other mobile devices now come equipped with an impressive array of sensors: multiple position sensors (GPS, WiFi, cellular radios), inertial sensors (accelerometers and gyroscopes), magnetic compass, microphone, light sensors, proximity sensors, and many more. These capabilities provide smartphones the ability to discover more about users and their activities than any other commodity computing device ever invented.

Application developers have recently started taking advantage of these powerful capabilities, leading to the first signs of a new class of *tasking* applications. These applications process data from multiple sensors, in a continuous fashion, to determine the user's context and activity, and take actions when certain conditions are met. They usually operate with almost no user input, and may require applications to *coordinate* their state with external conditions (e.g., turn off the ringer if I'm inside a movie theater) or with the state of other devices (e.g., send my phone a message if my spouse leaves work by 5 pm). Tasking applications are starting to get popular with end users [3, 8]. For example, Apple has integrated location-based reminders into its latest iPhone OS (iOS5) [4].

Our goal is to make it easy to develop and run new tasking applications. We believe that most tasking applications are relatively easy to state, often in a few words as "condition/action" rules, but are extremely difficult to implement today. In fact, this hypothesis is borne out by existing products, which only come with a set of pre-defined conditions and actions that users can configure, but cannot extend or personalize effectively.

Current approaches suffer from two problems:

1. **Poor abstractions.** Today, writing tasking applications requires grappling with low-level sensor data. Even something as easy to express as "is the user riding a bicycle" is difficult because one needs to process data from position, accelerometer, and/or gyroscope sensors to make this determination. An ideal solution would allow developers, and even end users, to use (and reuse) an `isBiking` primitive.
2. **Poor programming support.** Tasking applications are often inherently distributed. Writing them involves both server-side and smartphone code, and figuring out how to partition that code. A better approach for this type of application would be a *macroprogramming* approach, which *only* requires developers to write server-side code, with an execution framework that automatically partitions the code across one or more smartphones and the server. Additionally, a development framework that also supports end users, who have no interest or ability to write tasking code, is desirable. Such users should be able to combine existing capabilities to specify their own tasks. In this way, a user need not be dependent on, or wait for, a developer to create the task.

This paper addresses these shortcomings by proposing *Code In The Air* (CITA), a system that lowers the barrier

HotMobile'12 February 28–29, 2012, San Diego, CA, USA.
Copyright 2012 ACM 978-1-4503-1207-3 ...$10.00.

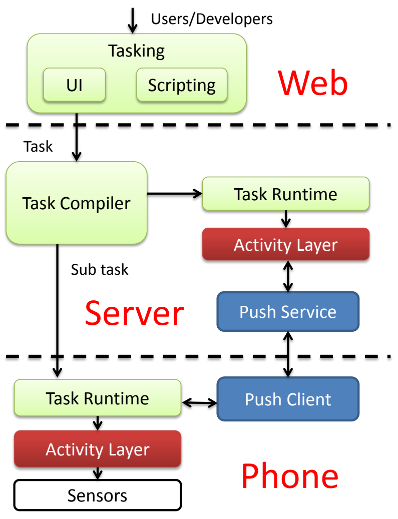

Figure 1: Code In The Air Architecture.

for programming and executing tasking applications. CITA helps two groups of people:

- Developers, who can create tasks by writing only server-side code, even for tasks that involve multiple end users and their devices, a variety of sensors, and the server. In our current implementation, these tasks are written in JavaScript.
- End users, who are able to specify their own tasks by "mixing and matching" available activities and tasks via a web UI (or a smartphone UI).

The CITA architecture, shown in Figure 1, has three interesting components. First, the *tasking framework* (in green) allows developers to write task scripts, compiles them into server-side and mobile code, and manages the task execution run-time on mobile devices. CITA provides a JavaScript interface in which developers can manipulate different phone devices as objects within a single program. The CITA back-end automatically partitions the code, deals with device coordination, and efficiently executes code on the devices. Second, the *activity layer* (in red) raises the abstraction for tasks from "sensor data hacking" to "activity composition" by providing extensible modules for higher-level activities (isBiking, isDriving, isOutside, enterPlace, leavePlace, and many more). The activity layer provides accurate, energy-efficient activity recognition. Third, the efficient push communication service (in blue) improves on the energy and load shortcomings of existing systems like the iOS and Android push services.

Using CITA, a variety of tasks become easier to express and run. These include single-device tasks (e.g., don't connect to WiFi when I'm outside and moving), multi-user tasks (make my phone silent when I'm meeting with my boss), tasks with complex activities (map my path when I bike or run but not when I drive or ride the bus), and multi-device tasks (put my laptop to sleep if I've not been in the same room for more than 15 minutes).

This paper outlines the CITA architecture and the three components. In addition to enabling novel tasking applications, our vision is to create an ecosystem where developers write interesting activity and task modules, and provide them in a marketplace for other developers as well as end users to use. The ability to "componentize" activities and compose them will enrich the tasks available to end users.

2. ACTIVITY LAYER

CITA includes an activity layer which makes it easy for developers to express conditions that reference high-level activities, such as the fact that the user is "in a meeting" or "leaving her workplace". It also provides an easy way to *compose* primitive activities to build useful higher-level activities.

Both accuracy and energy efficiency are non-trivial challenges. Our system recognizes a number of low-level activities such as *isWalking*, *isDriving*, *enterPlace*, and *leavePlace*. Other researchers have previously worked on many of these activity recognition problems in isolation [19, 20, 17, 12, 16, 11], and we use many of these techniques in CITA. Rather than describing each individual activity in detail, we describe two key new pieces of our system: accurate and energy-efficient place detection with location hierarchies, and activity composition.

2.1 Place Hierarchies

Consider the example of detecting that a person has left her office, and alerting her spouse when this happens. This is a subjective condition because the intent of the user can differ based on the context. If the goal is to alert her spouse when she leaves her office building for the day, the system should not alert her spouse each time she steps out of her office room for a walk or a meeting. On the other hand, if the goal is to recognize when she is in a meeting and not in her office room, this is exactly the behavior we want.

Hence, CITA uses an in-built location hierarchy that recognizes three different granularities of locations, in increasing order of "coarseness": *room level*, *floor level*, and *bounding box or building level*.

End users use the CITA UI to mark named locations, or "places" on a map and label them as 'office' or 'workplace' or 'home'. Developers can also programmatically create named locations in their apps using a special API. The phone collects sample localization information while creating the location (such as WiFi access point, cell tower, GPS and network location information) that it uses later to identify if the user is at the named location.

We require that a user or developer explicitly specify or choose the level in the location hierarchy that a named location refers to when creating it. For example, a user might go to their office and create 3 levels of named locations: one called 'My Office Room' that refers exclusively to his/her room, one called '5th floor' that refers to his/her floor or wing, and one called 'Convention Center' that refers to his/her entire office building.

The three levels of the hierarchy have different underlying implementations. Room-level location matches a named location if the WiFi signal strength signature is within some "distance" of the signature(s) observed when creating the named location.

Floor-level location is usually recognized by looking for *any overlap* in WiFi signal strengths i.e. if any access point matches any of the access points seen when creating the named location.

A bounding box is the most coarse-grained level of location that refers to an entire building, set of buildings or a larger bounding box on a map. The implementation explicitly obtains a GPS or network location request and checks if it lies in the bounding box. A user explicitly draws the bounding box on the UI to specify how big they want the named location to be. It could span anywhere from a single building to an entire city.

CITA provides two in-built activity detectors for each named location: `enterPlace` and `leavePlace` that are triggered when a user enters and leaves a named location respectively. In the example mentioned, if "office" is a room-level location, the `leavePlace(office)` condition is triggered when the phone's WiFi signature is no longer close to any of the observed signature(s) in "office", and CITA executes the action (alerting the user's spouse). If office is a floor-level location, CITA alerts her spouse only if none of the access points seen in her "office" are currently seen by the phone. If office is a larger bounding box, CITA alerts her spouse only if the GPS or network location coordinate goes outside the bounding box.

Energy. Energy efficiency is a key challenge in location detection. It is not desirable to continuously sample GPS location, or to continuously scan WiFi on a smartphone because it drains the battery quickly. CITA uses a simple energy-efficient algorithm to detect when a user enters or leaves a named location. The location can fall into any of the above three levels in the hierarchy.

We begin by sampling location sensors with the lowest energy cost first to find approximate location. The algorithm switches to a more energy-intensive sensor *only* if the user gets closer to the named location than the *maximum* possible error in the location sensor being currently used. The algorithm also uses adaptive sampling to vary the sampling rate for a given sensor each time we get a new location sample. It estimates the minimum amount of time to reach the named location, and samples only after that period of time has elapsed.

For example, if trying to detect whether a user has entered the Empire State building, the system first uses the cheapest sensor, cellular localization, to find approximate location, and estimate the minimum time to reach the location of interest. For example, if cellular location reveals the user to be in Japan, it would take at least 24 hours to reach New York by air, so it is sufficient to obtain a location sample after 24 hours. When the system finds a location sample less than a kilometer from Empire State, it switches to using WiFi localization because the expected error from cellular location is greater than the estimated distance to Empire State. The system switches to using the most expensive location sensor — GPS only when close enough to the named location "Empire State".

An additional optimization we use is to use the accelerometer to detect movement: we only obtain new location samples if a user has moved since the last location sample.

2.2 Activity Composition

CITA allows developers and users to *compose* lower-level activities using logical predicates to create high-level activities. This is very powerful because it allows developers and end users to reuse activity modules created by other developers to quickly write their own activities.

For example, consider the problem of detecting that a user is outdoors and walking, and switching to 3G from WiFi when that happens. Since "user is outdoors" and "user is walking" are primitive activities, we can compose them using an `AND` predicate in CITA to build the complex event "outdoors and walking".

Another example: Alice wants her phone to go into silent mode automatically if she is in the meeting room of her building with either her boss John or her colleague Bob. This is a *multi-phone* activity predicate, which can be expressed in CITA as follows:

- If Alice is in the meeting room, `AND`
- John is in the meeting room, `OR` Bob is in the meeting room.

then make Alice's phone silent.

CITA supports `AND`, `OR`, `NOT` as well as the following additional composition primitives:

- `WITHIN`: The event `A WITHIN 10 sec` remains true for 10 seconds after the time when A becomes true. A `WITHIN` clause is usually used with an `AND` clause to permit some leeway when event detection has a time lag (such as a user leaving a building). For example, the event `A WITHIN 10 sec AND B WITHIN 10 sec` is true if both `A` and `B` were true within a 10 second window of each other.
- `FOR`: The event `A FOR 10 sec` becomes true whenever A is true for at least 10 seconds, and is triggered once when this happens. This is useful to express tasks like "turn the computer screen off if Mary has been away from her office for at least ten minutes".
- `NEXT`: The event `A NEXT B` becomes true if A changes its state from true to false, and B is true now. This is usually used to monitor state changes — for example, to trigger a task to execute whenever a user commuted from home to work.

CITA implements composed activities by continuously running activity detectors for all parts of the predicate. If an activity spans multiple users as in the example with Alice, John and Bob above, the compiler uses static analysis on the activity predicate (based on which phone objects are referenced) to divide it into individual lower-level predicates among phones, and only send back data to the central server when a predicate actually becomes true.

In addition to in-built primitive activity detectors and composed activities, CITA's extensible activity layer also provides a way for sophisticated developers to build new primitive activity detectors by accessing raw sensor data (using JavaScript).

3. TASKING FRAMEWORK

In CITA, developers can write scripts using a web interface to build complex tasking applications. Activities and tasks contributed by developers are added to a UI (available on web and on phones) for end users to use. The end user interface allows users to specify complex conditions (using the Activity composition framework) and build "condition-action" tasks. (Figure 2 shows a mockup of the end user interface). The UI generates scripts written using the developer programming model. In this section, we focus on the developer's programming model.

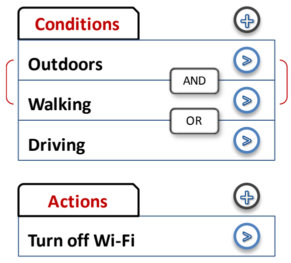

Figure 2: Mockup of the end user interface to specify "condition-action" tasks.

Tasks are represented as JavaScript programs. The CITA interpreter provides a way for JavaScript to deal with Java objects and call methods on them [7]. CITA exposes phone devices as objects — a developer can simply call methods on the object to manipulate it. This object model is convenient because it enables a developer to work with multiple phones in a single script. When a script creates multi-device tasks, CITA partitions the task into sub-tasks, and executes and coordinates sub-tasks across the devices.

We explain our programming model using three simple tasking applications: a single device task, a cloud-enabled task, and a multi-device task.

A Single-Device Task. The following shows a CITA script to express a simple application: "Bob wants his phone to use 3G rather than WiFi when outdoors":

```
p = getDevice("Bob", "Phone");
p.registerActivityCallback("outdoor", "
    trueAction", "falseAction");
var oldState;
function trueAction(e)
{
    oldState = p.wifi.getState();
    p.wifi.disable();
}
function falseAction(e)
{
    p.wifi.setState(oldState);
}
```

An end user can register multiple devices with CITA. The `getDevice` function accepts a userName and deviceName and returns the device object to work with.

The tasking runtime provides two types of event callbacks one can register for - an activity callback and a data callback. When registering for activity callbacks, developers can specify two callback functions - the function to be called when an activity condition changes from false to true, and a function to be called when the activity condition changes from true to false. The above example registers for an "outdoor" activity event requesting both callback functions.

`trueAction` function is called when the "outdoor" activity becomes true. The function saves the old WiFi state and disables WiFi. `falseAction` is called when "outdoor" becomes false, in which the previous state of WiFi is restored.

It is also possible to register callbacks for high-level or composed activities. Composed activities are created from basic activities or other composed activities using the Activity Composition framework described in Section 2. An example could be "if the user is outdoors and is near his office and is walking, then invoke a callback function that disconnects from WiFi".

When a developer creates such a task, the Task Compiler analyses the code and pushes the entire JavaScript code into the device since it involves only a single device (see Section 3.1). The phone object (`p`) is replaced with a local device interface object on the device.

A Cloud-Enabled Task. In this task "Eve wants her phone to map her bike path whenever she goes for a long bike ride and store it on the server".

```
p = getDevice("Eve", "Phone");
p.registerActivityCallback("biking", "
    trueAction", "falseAction");
function trueAction(e)
{
    params = new Object();
    params.samplePeriod = 1;
    params.tag = getUniqueId();
    p.registerDataCallback("gps.data", "
        dataAction", params);
}
function falseAction(e)
{
    p.unregisterCallback("gps.data", "
        dataAction");
}
function dataAction(e)
{
    Server.store.add("bikingTracks", e.tag
        , e.data.time, e.data.lat, e.data.
        lng);
}
```

The script registers for a "biking" activity callback. When "biking" becomes true, `trueAction` function is called which in turn registers for a GPS data callback. It passes two parameters to the GPS data callback - `samplePeriod` tells the GPS sampling period and parameter `tag` which is passed back unmodified into the callback function. Here tag is initialized to a unique id (to represent a biking track).

In the `dataAction` function, the received GPS data is pushed into a server store. `Server` is a special object that lets the developer work with utilities on the server. CITA provides a primitive server store that lets developers store data.

The Task Compiler partitions this task into two sub-tasks. Tasks are partitioned at the function level into two separate JavaScript programs (see Section 3.1). In the above example, `dataAction` is placed on the server since it works with the `Server` object. The rest of the code runs on the phone. The `registerDataCallback` call in the phone code is rewritten to call the `dataAction` function in the server i.e, during "gps.data" callbacks, the callback data is serialized, sent to the server and the `dataAction` function is called on the server.

A Multi-Device Task. In this example, "Jim would like to be alerted whenever his spouse Alice leaves her workplace".

```
p = getDevice("Alice", "Phone");
q = getDevice("Jim", "Phone");
p.registerActivityCallback("leavePlace(
    work)", "trueAction", null);
function trueAction()
{
    q.vibrate();
    q.alert("Alice left work");
}
```

Here, the Task Compiler produces two sub-tasks to run on each phone and spawns a Task Coordinator at the server to coordinate the calls and callbacks between the subtasks.

We show the code produced by the compiler for the two sub-tasks below:

On Alice's phone

```
this.registerActivityCallback("leavePlace(
    work)", null, "Bob.Phone:trueAction");
```

On Jim's phone

```
function trueAction()
{
    this.vibrate();
    this.alert("Alice left work");
}
```

When the "leavePlace(work)" condition becomes true on Alice's phone, the sub task contacts the server to call `trueAction` on Bob's phone. The server delegates the call to Bob's phone which creates an alert.

When a task is created using the smartphone end user UI, the tasking runtime in the phone is capable of identifing whether the task involves only the current phone. If so, the execution stays completely local and does not make any round trips to the server. If the task involves multiple devices, the execution is delegated to the Task Compiler at the server.

3.1 Code Partitioning

Currently, the Task Compiler uses two heuristics to partition the code into sub tasks.

1. If a function works with only one device object, that function always runs on that device. If a function works with multiple device objects, it runs on the server.

2. If a function does not work with any device objects, we look at the call graph. If it is a callback function for a device activity or it lies in the sub-tree of such a callback function, we record that the function depends on that device. If a function depends on a single device, it runs on that device, otherwise it runs on the server.

When functions are partitioned, phone objects and callback methods are rewritten if necessary to inform the tasking runtime to delegate the calls. For example, in the multi-device task shown in the last section, note that the callback function in register callback was prefixed with where the function is present.

Even if the code partitioning heuristic does not efficiently partition the code, it *does not affect the correctness of the task execution.* For instance, in the single task example, the entire code could run at the server or even on some other phone. CITA would delegate the calls, callbacks to and from the phone and the server, and properly serialize the parameters and data. It would just be inefficient since it incurs unnecessary roundtrips across the network.

In addition to call-graph based partitioning, we are currently working on profiling-based partitioning techniques ([13], [18]) that would reduce bandwidth, computation and energy costs.

Handling Disconnections. One challenge in distributed task execution is handling disconnections between devices. When devices become disconnected, the callbacks between them cannot be done in real-time. In such scenarios, callbacks can either be delayed or dropped depending on the task. In the "bike path mapping" example shown above, it would be best to delay the client callbacks until the connection is restored to the server (so that no GPS data is lost). In the "alerting spouse" example, it might be better to drop the callback after a certain amount of time. CITA allows task developers to set both task-wide policies and fine-grained callback specific policies on how to deal with disconnections.

Privacy Model. End users must explicitly grant fine-grained permissions to other users to register for activities or run specific tasks on their mobile device. For example, Alice must grant permission to her spouse to monitor `leaves(work)`. Permissions are currently enforced dynamically, at run time.

4. DIAL-TO-DELIVER PUSH SERVICE

CITA provides an asynchronous message delivery service from the CITA server to the mobile devices, similar to the way in which "push notifications" work on current smartphones. However, current approaches suffer from three problems. First, when the amount of information to be pushed is tiny compared to the typical web page download, as is commonly the case, the current approach of using a long-running TCP connection initiated from the mobile ties up a data channel in the wide-area cellular wireless network for much longer than is needed, and is inefficient.

Second, because many mobile devices connect to the server from behind a NAT, long-standing TCP connections without application-level keep-alives time out because the NAT clears out state in a way that may be hard for the mobile device to predict. The server, which generally sends "push" data gets a TCP RST, but the mobile has no knowledge that the connection has timed out. Periodic keep-alives are one solution, but they consume excess energy and also tie up data channels. Moreover, the timers used in these keep-alives depend on the specific timeouts used in the NATs in different carriers, and are not easily visible to the ends (current practice is to try and reverse-engineer these values for each provider/city combination!). Third, current push notifications heavily restrict notifications to specific types, whereas we are interested in task-specified notifications.

To avoid these three problems, we developed the *dial-to-deliver* service, which uses the signaling mechanism in the cell telephone network. It works as follows.

CITA has a set of registered phone numbers. Using a standard IP telephony service, the CITA server initiates a call to any smartphone with a voice plan or VoIP capability. On the smartphone, the CITA client has a hook into the OS to listen for phone calls (we have implemented it currently in Android). When a phone call is received, the CITA client gets a callback; it checks for the phone number and if the phone number matches a registered CITA number, then it rejects the call right away, wakes up the CITA client, and processes the call. The task can encode a small amount of information in the phone number used; if there are k CITA numbers, up to $\log_2 k$ bits of information can be encoded in a task-specific way to notify the client sub-tasks of the event. Some of these notifications may cause the CITA client to initiate a TCP connection to the CITA server to deliver or download data, which will cause the 3G or WiFi radio to turn on. But in many applications where the server notifies the client, the client may only need the notification and may find that local conditions don't require any further data transfers; in these cases, dial-to-deliver reduces the load on the 3G data network and also reduces the energy consumption on the mobile.

In our measurements, the latency of pushing data using this technique is around 5 seconds; this value is consistent with the typical latency between the initiation of a telephone call and the establishment of the call. As such, dial-to-deliver is suitable for cases when a few seconds of latency is tolerable. Examples including notifying the mobile of new tasks being available, and cases when the local conditions to be checked don't change fast enough (e.g., whether the user is outside or inside a building), etc.

One potential issue with dial-to-deliver is that it increases the signaling load on the voice network, though it reduces the load on the data network. We believe that this trade-off is the right one with current designs because the data networks seem to reserve resources anticipating that the data transfer will last a while and only rely on timeouts to clear this state, while a rejected call clears out the state in the voice network immediately. That said, a more careful investigation of the resulting load is required before concluding that dial-to-deliver is not only energy-efficient (which it is), but also more scalable in today's networks.

5. RELATED WORK

Tasking applications are getting popular with end users. Locale [3] and Tasker [8] are Android applications that let users specify context aware single-device tasks. TouchDevelop [9] is a Windows Phone Application that lets users "script" simple tasks. These applications come with many pre-defined conditions and actions that users can select to build tasks. Unlike CITA, these applications expose only raw sensor data (or primitive activities) and hence haven't gained popularity outside of power users. In contrast, CITA exposes higher level activities that normal users can understand and lets them build complex activities. CITA also dramatically simplifies cloud-enabled and multi-device programming.

Simplifying smartphone programming in general has been the focus of some commercial and research efforts. Google's AppInventor [1] provides a visual programming interface, PhoneGap [6] provides an HTML/Javascript framework, and AppMakr [2] lets users program web-feed based apps without writing traditional code. Research efforts have focused on crowdsourcing and remote sensing applications (e.g., mCrowd [5] and PRISM [14]). CITA provides explicit tasking support unlike many commercial efforts, and targets both developers and end users with its tasking and activity framework unlike all these efforts.

Researchers have worked on activity detection algorithms for many kinds of activities on mobile phones [19, 20, 17, 12, 16, 11]. SeeMon [15] provides a composition framework using a query language for detecting multiple activities together. CITA's contribution is a multi-device programming model that can be layered on top of an energy-efficient activity detection and composition layer.

Cell2Notify [10] uses GSM signaling similar to CITA's Dial-to-deliver to wake up WiFi on the phone for VoIP calls. Dial-to-Deliver is a more generic push mechanism that can be used to transmit tiny bits of information to applications.

6. CONCLUSION

We introduced CITA, a system that eases the development and running of tasking applications on smartphones. These applications combine sensing and coordination. The key components of the CITA architecture are a tasking framework that enables developers to write server-side programs in lieu of distributed programs, an energy-efficient activity layer, and a dial-to-deliver push service. We are currently completing the implementation of the system and will soon make it available to developers and end users.

Acknowledgments

This work was supported in part by NSF Grant 0931550.

7. REFERENCES

[1] App Inventor. http://appinventor.googlelabs.com/about/.

[2] Appmakr. http://www.appmakr.com/.

[3] Locale. http://www.twofortyfouram.com/.

[4] Location based reminders in iOS5. http://www.apple.com/ios/features.html.

[5] mCrowd. http://crowd.cs.umass.edu/.

[6] PhoneGap. http://www.phonegap.com/.

[7] Scripting for the Java Platform. http://java.sun.com/developer/technicalArticles/J2SE/Desktop/scripting/.

[8] Tasker. http://tasker.dinglisch.net/.

[9] Touch Develop. https://www.touchdevelop.com/.

[10] Y. Agarwal, R. Chandra, A. Wolman, P. Bahl, K. Chin, and R. Gupta. Wireless wakeups revisited: energy management for voip over wi-fi smartphones. In *MobiSys*, 2007.

[11] B. N. Waber and D. O. Olguin and T. Kim and A. Mohan and K. Ara and A. Pentland. Organizational Engineering Using Sociometric Badges. In *NetSci*, 2007.

[12] Consolvo, S. and McDonald, D. and Toscos, T. and Chen, M. and Froehlich, J. and Harrison, B. and Klasnja, P. and LaMarca, A. and LeGrand, L. and Libby, R. Activity Sensing in the Wild: A Field Trial of UbiFit Garden. In *SIGCHI*, 2008.

[13] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making smartphones last longer with code offload. In *MobiSys*, 2010.

[14] T. Das, P. Mohan, V. N. Padmanabhan, R. Ramjee, and A. Sharma. PRISM: Platform for Remote Sensing using Smartphones. In *Mobisys*, 2010.

[15] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. Seemon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *MobiSys*, 2008.

[16] H. Lu, W. Pan, N. Lane, T. Choudhury, and A. T. Campbell. Soundsense: Scalable sound sensing for people-centric applications on mobile phones. In *MobiSys*, 2009.

[17] Miluzzo, E. and Lane, N. and Fodor, K. and Peterson, R. and Lu, H. and Musolesi, M. and Eisenman, S. and Zheng, X. and Campbell, A. Sensing meets mobile social networks: The design, implementation and evaluation of the CenceMe application. In *SenSys*, 2008.

[18] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based Partitioning for Sensornet Applications. In *NSDI*, 2009.

[19] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava. Using Mobile Phones to Determine Transportation Modes. *Transactions on Sensor Networks*, 6(2), 2010.

[20] A. Thiagarajan, J. Biagioni, T. Gerlich, and J. Eriksson. Cooperative Transit Tracking Using GPS-enabled Smart-phones. In *SenSys*, 2010.