Deployment and Controlled Execution of Sensing Tasks on Smartphones

PRESENTED THE September 25, 2014

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

UNIVERSITY OF THESSALY

FOR GRADUATION OF BACHELOR OF SCIENCE

BY

Emmanouil KATSOMALLOS

supervised by:

Spyros Lalis, Associate Professor
Dimitrios Katsaros, Lecturer

Greece
2014

Part of this work was presented in the proceedings of the 1st International Workshop on Crowdsensing Methods, Techniques and Applications (Crowdsensing), in conjunction with the 12th IEEE International Conference on Pervasive Computing and Communications (PerCom) 2014 under the title: “EasyHarvest: Supporting the Deployment and Management of Sensing Applications on Smartphones” [13] and co-author Spyros Lalis.

Acknowledgements

Upon the completion of my diploma thesis, I would like to express my deep gratitude to my research supervisor Dr. Spyros Lalis for his patient guidance, enthusiastic encouragement and useful critiques of this research work.

A special thanks to my department's faculty, staff and fellow students for their valuable assistance whenever needed and for creating a pleasant and creative environment during my studies.

Last but not least, I wish to thank my family and friends for their unconditional support and encouragement all these years.

Volos, September 25, 2014

Περίληψη

Σε αυτή την έρευνα, παρουσιάζουμε ένα πρώτο πρωτότυπο του πλαισίου *EasyHarvest*, που έχει ως στόχο να απλοποιήσει τη διάδοση και την ελεγχόμενη εκτέλεση εφαρμογών ανίχνευσης μεγάλης κλίμακας σε έξυπνα τηλέφωνα. Από τη μια, οι ιδιοκτήτες των εφαρμογών υποβάλουν στον διακομιστή εργασίες ανίχνευσης για διανομή σε έξυπνα τηλέφωνα, και συλλέγουν τα δεδομένα που παράγονται από αυτές με απλό τρόπο. Από την άλλη, οι ιδιοκτήτες των έξυπνων τηλεφώνων ελέγχουν την εκτέλεση των εργασιών ανίχνευσης στις συσκευές τους μέσα από μια ενιαία διεπαφή, χωρίς να χρειάζεται να κατεβάζουν, να εγκαταθιστούν και να ρυθμίζουν επανειλημμένα κάθε εφαρμογή ανίχνευσης. Η αλληλεπίδραση μεταξύ του έξυπνου τηλεφώνου και του διακομιστή πραγματοποιείται με διαφανή τρόπο, με ανοχή σε διακοπτόμενη συνδεσιμότητα και υποστήριξη για αποσυνδεδεμένη λειτουργία.

Λέξεις-κλειδιά

κινητός υπολογισμός; διάχυτος υπολογισμός; κατανεμημένος υπολογισμός; κινητή ανίχνευση; ανίχνευση πλήθους; έξυπνα τηλέφωνα; Android

Abstract

In this research, we present a first prototype of the *EasyHarvest* framework, which aims to simplify the deployment and controlled execution of large-scale sensing applications on smartphones. On the one hand, application owners submit to a server sensing tasks for distribution on smartphones, and collect the data produced by them in a simple manner. On the other hand, smartphone owners control the execution of sensing tasks on their devices through a single interface, without having to repeatedly download, install and configure individual sensing applications. The interaction between the smartphone and the server occurs in a transparent way, with tolerance to intermittent connectivity and support for disconnected operation.

Keywords

mobile computing; pervasive computing; distributed computing; mobile sensing; crowdsensing; smartphones; Android

Contents

List of figures

List of tables

Chapter 1 Introduction

An ever increasing number of sensors are integrated in different personal devices and public/private vehicles. Given the mobility and potentially large population of such sensing elements, their aggregated sensing capacity and area coverage might very well exceed that of planned static infrastructures. Indicative applications examples that illustrate the power of mobile sensing have already been reported in the literature, such as the monitoring of environmental parameters [1], the sharing of biker's routes and fitness data to capture the popularity and difficulty of routes [2], and the use of onboard car cameras to locate a license plate on demand [3].

The smartphone is crucially important to the proliferation of mobile sensing applications. Carried by millions of people, everywhere they go, it is about to become (if not already is) the most ubiquitous sensor node in the planet. Besides their onboard sensors, smartphones can also become proxies or hubs for an even broader range of wearable sensors, which can be connected to them via cables or short-range radio. People with their gadgets, going about their everyday lives, can thus constitute a valuable mobile sensor platform.

However, the development of applications to exploit the potential of mobile and typically privately-owned sensing devices is very challenging and a far cry from the already hard task of writing conventional distributed programs. This is due to the many new problems that arise in the context of this computing ecosystem, such as the asymmetrical and intermittent connectivity of mobile nodes, the battery and resource limitations of portable devices, the fusion of data from different sensors and sources, data ownership and privacy concerns, etc. To make matters worse, most applications are still developed from scratch, which makes such endeavors prohibitive for communities that do not have substantial human and financial resources at their disposal.

The situation is not much better for the people who have no interest in developing their own sensing applications yet would gladly participate in such applications by contributing their sensing resources. Namely, the owner of a mobile device or smartphone typically now has to download, install and manage each application separately. This administrative burden, already quite annoying and disruptive for most people as it is, becomes intolerable for a large number of applications. Clearly, this may compromise the massive deployment which is required for such applications to provide significant value.

Thus, the development and large-scale deployment of applications that can exploit the sensing potential of personal mobile devices, such as smartphones, remains a challenge. Many different yet interrelated problems need to be addressed, most notably: (i) dealing with asymmetrical and opportunistic connectivity; (ii) preserving battery lifetime; (iii) addressing security, privacy and trust concerns; (iv) simplifying the installation and management of sensing tasks; and (v) motivating ordinary people to actually participate in such sensing efforts.

We are developing a framework called *EasyHarvest*, which aims to lower the barrier of entry both for developers and participants of crowdsensing applications that are based on smartphones. This thesis presents a first prototype, for Android phones. With respect to the aforementioned problems, our prototype tackles (i) and (iv). In a nutshell, application-specific sensing tasks are submitted on the *EasyHarvest* server, optionally providing a time-space region of interest. The execution of the sensing tasks on the smartphone occurs under the control of a resident application program, the *EasyHarvest* client. The client downloads the code of tasks and uploads the data being produced by them behind the scenes, dealing with intermittent connectivity, while supporting disconnected operation. Tasks can be paused, restarted and withdrawn via the server. Respective status updates are asynchronously received by the client, which handles the locally executing tasks accordingly.

Our work primarily targets communities and/or business organizations with registered users. The idea is that each community maintains its own *EasyHarvest* server, which can be accessed through the existing authentication and access control infrastructure. More important, we suppose that the sensing applications are developed by and for the community,

hence are considered to be non-malicious and free for inspection by community members. Thus our prototype does not focus on security issues, nor does it address the typical security problems due to dynamic code (task) installation on the smartphone. However, the *EasyHarvest* client provides a few settings through which users can control the execution of sensing tasks on their smartphones. These settings apply to all tasks so that the user does not have to be notified about and configure every single task that lands on his device.

The rest of the thesis is structured as follows. Chapter 2 gives an overview of the *EasyHarvest* system architecture and operation. Chapter 3 discusses key implementation aspects, and describes the programming conventions that have to be followed by the developers of application sensing tasks. Chapter 4 briefly describes how the user interacts with the *EasyHarvest* client and server. Chapter 5 provides results from our evaluation. Chapter 6 discusses related work. Finally, Chapter 7 concludes the thesis and identifies directions for future work.

Chapter 2 System Overview

2.1 Key characteristics

EasyHarvest provides a generic mechanism for the automatic deployment of sensing tasks on smartphones, and the collection of the data that is produced by them. The key characteristics of *EasyHarvest* are as follows:

- Each task is treated as a separate application kernel, which is instantiated on as many devices as possible in order to collect data from a large number of sources.
- Besides sensing, tasks can also perform application-specific data processing, directly on the smartphone.
- The different instances of a given task are logically independent, and are executed in a decoupled way.
- Task deployment and data collection is performed in a lazy, asynchronous and incremental fashion, tolerating intermittent connectivity.
- Task data is aggregated at a central repository on the server, from where it can be retrieved by external application logic that implements further data processing and visualization.
- Data flow is unidirectional, from the task instances to the data repository, and from there to the external application logic (if any).

The typical scenario we envision at this point is for an application to be comprised of a single sensing task that will run on smartphones for a long time, and some external data post-processing logic. But note that the application owner can submit, start/pause and withdraw tasks at any point in time.

2.2 Architecture

The high-level architecture of EasyHarvest, shown in Figure 1, follows a client-server approach. The server maintains a separate entry for each sensing task runs on a reliable and always available infrastructure, managed by the community that wishes to run sensing applications on the mobile phones of its members. The client is a smartphone application, that downloads task binaries and then instantiates/executes tasks on the smartphone. Community members have to install the client (once) so that their device can receive and execute sensing tasks. Application owners use the task management user interface of the server in order to submit, start/pause and withdraw sensing tasks. In turn, the client allows users to control task execution and communication on their smartphones. Task data is uploaded to the server and updates of the task's properties are retrieved from the server in an opportunistic way. Data post-processing is left for external, application-specific subsystems.

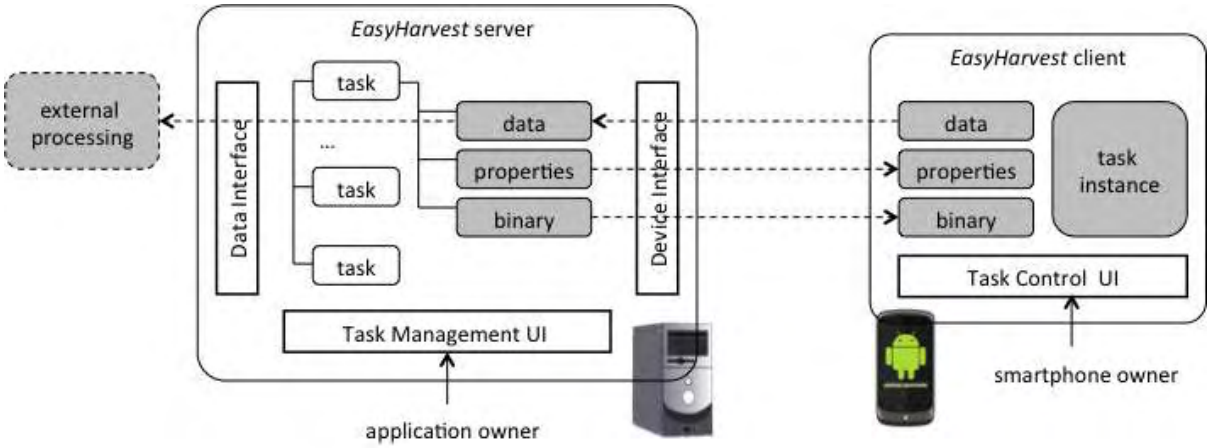

Figure 1: High-level architecture of the *EasyHarvest* framework

The server has two machine-to-machine interfaces: the device interface, and the data interface. The device interface is used by clients to download task binaries on the smartphone, to upload on the server the data that was generated by locally running tasks, and to retrieve updates on the task's status and other dynamic properties. The data interface is used to retrieve task data from the server. It is intended for external programs that process, analyze and visualize data according to the application's requirements. Note that the server does not interpret or process task data in any way.

2.3 Operation

The server keeps a separate entry for each submitted task. This includes the task's binary and metadata, its properties (including its status), and the data that has been produced by its instances running on different smartphones.

When the client executes for the first time, it registers with the server by negotiating a device identifier to be used in all subsequent interactions. Smartphone owners may instruct the client to unregister when they no longer wish for their devices to be part of the sensing applications of the community. Also, the server will unilaterally unregister a client that does not show any sign of activity for some time; the assumption is that the user uninstalled the *EasyHarvest* client from his phone without properly unregistering it.

Once registered, the client periodically asks the server for a task to execute. The server picks one of the tasks that have been started, and sends its identifier to the client. Currently, the server evenly assigns tasks to smartphones so that all tasks are instantiated the same number of times. More advanced assignment policies, e.g., ones that take into account the number of instances required for each task, or exploit the mobility profiles of smartphones, are left for future work.

When the client receives a task identifier, it downloads the corresponding binary, instantiates the task and starts it. The data that is being produced by the task it is buffered locally. The client opportunistically tries to contact the server in order to upload the data when the smartphone is connected to the Internet. It also retrieves from the server updates on the dynamic properties of the task, and pauses, restarts or removes the task accordingly. The basic stages of the client's operation are shown in Figure 2. Remote interactions with the *EasyHarvest* server are shown in grey. Dashed rectangles indicate potentially long data transfers, which are performed in an incremental way.

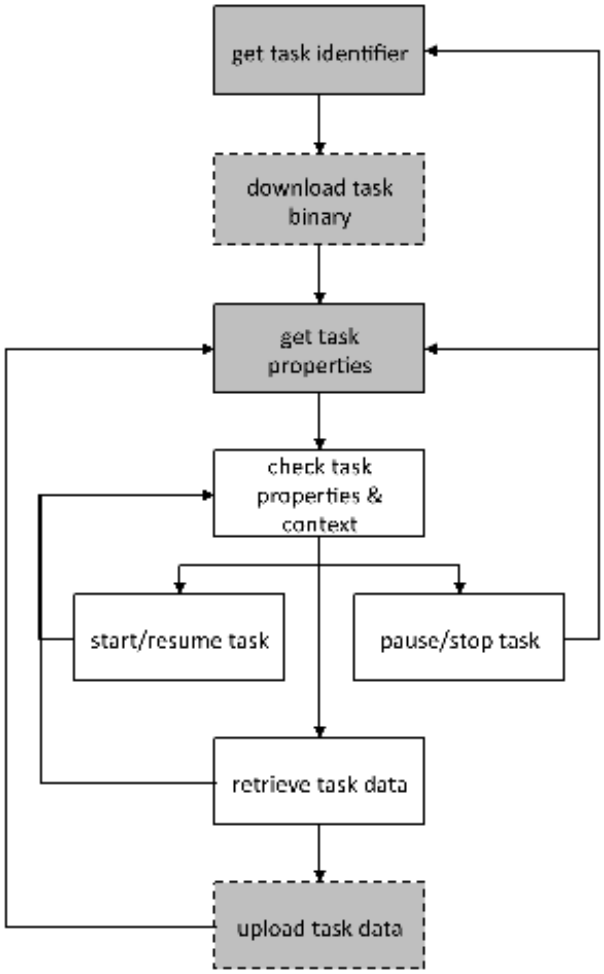

Figure 2: Operation of the *EasyHarvest* client on the smartphone

If the application owner decides to withdraw a task, the server permanently deletes the respective entry (and the data). A client that attempts to address a task that has been removed receives an error message (the respective identifier is invalid), and consequently stops the task and removes it from the smartphone. In our current implementation, the client hosts one task at a time; thus, a new task is requested from the server only when the previous task is removed. It is straightforward to modify the client to host several tasks (but this was not a priority in our prototyping efforts).

2.4 Intermittent connectivity and disconnected operation

The communication between the client and server occurs behind the scenes, transparently to the application tasks. Both programs are designed to tolerate disconnections and to work even if there is no communication between them for longer periods. In particular, potentially time-consuming transfers such as code downloads and data uploads are performed in an incremental way so that they can be resumed after a disconnection. This is important since Internet connectivity could be infrequent and short-lived. For instance, a person may encounter Wi-Fi hotspots a couple of times during the day, and remain connected for just a few minutes. Note however that this lazy and opportunistic mode of operation is not suitable for applications with real-time requirements.

2.5 Time and location-aware task execution

In many cases it could be desirable to limit task execution within certain periods of the day and/or specific geographical regions. As an example, an application concerning the city of Volos during nighttime should not run during the day or when the smartphone is in Athens.

While such a context-aware task execution could be left for the application programmer, this has drawbacks. Firstly, development becomes more complex as the programmer must include (mundane) checks in the task code. Secondly, letting the task perform these checks can waste resources, especially if the smartphone hosts several tasks. Thirdly, to change the time-space region of interest, the task would have to be withdrawn, modified, recompiled, resubmitted to the server, and downloaded from scratch on smartphones.

Instead, when submitting a task, the application owner can optionally specify the spatiotemporal region for which the task should be active. This information, much like the task's status, is treated as a dynamic property of the task. It is therefore refreshed when the client contacts the server, and is used to pause and restart the task as a function of the smartphone's time and location context (see Figure 2).

2.6 Privacy regions

The device owner may define a list of areas where no sensing (task execution) should happen. To add a privacy region, the user has to select a map area by dragging and zooming on a map interface and enter a label (home, office, etc.) to the selected area. Subsequently, the user can edit/delete the created regions.

After the user defines a new region, his/her preferences are saved locally on the device. The client, monitors user's current location and except for the tasks' location settings – defined by the task developer – takes into account the device owner's privacy regions settings, in order to start or temporally pause the sensing process.

Chapter 3 Implementation Details

3.1 Device interface

The device interface of the *EasyHarvest* server includes a few simple request-reply interactions in the form of a RESTful API, summarized in Table 1. The main reason we went for a REST-based solution was that we could rely on an established technology for which implementations are readily available for many different platforms and smartphones. This way we were also able to exploit the built-in support for incremental file transfers. The same approach was followed for the data interface (not shown in Table 1), which includes a call for retrieving task data from the server.

Client - server primitives	Description
…/dev/register	Register (or unregister) a client (smartphone).
…/tasks/gettaskinfo	Receive the id and metadata of the task to execute.
…/tasks/taskID/getbin	Download the (rest of the) task binary.
…/tasks/taskID/putdata	Upload the (rest of the) data produced by the task.
…/tasks/taskID/checkdata	Check up to which point data upload was successful.
…/tasks/taskID/getprop	Receive the (updated) dynamic task properties.

Table 1: The REST-based device interface

The corresponding server logic is implemented as a backend to the Tomcat web server from Apache (which manages all communication and runtime aspects). Basically, all our code does is to maintain a separate directory for each task, where related information is placed. The task's binary, properties and data are kept in separate files, which are read and/or written as needed to handle client requests.

3.2 Client operation and task execution

The *EasyHarvest* client is implemented as an Android application. The task binaries are loaded and invoked dynamically, using the class loading mechanism of Android's Java environment (Dalvik). Task execution is controlled via specific methods of the task (described next).

We wanted the client to run with minimal user interaction; users should set their preferences regarding the execution of tasks on their phones (see Section IV) and then ideally forget about the *EasyHarvest* system. Accordingly, the client is implemented as a "background service". One problem we faced with this approach is that the client can apparently be killed at any point in time if the operating system runs low on resources (e.g., memory). What's worse, this termination is abrupt, without giving the client program a warning or a chance to save its state. To be able to gracefully resume execution after such a kill, the client proactively saves its state when changes occur.

A similar approach is followed for tasks, namely the client periodically asks the task to save its state (as if it were to be paused). The recorded state is used to re-start the task when the client restarts after a kill. Note that from the perspective

of the application programmer it makes no difference if the task is paused and then restarted (i) because the application owner manually pauses/restarts the task via the server, (ii) because the client automatically pauses/restarts the task based on the current time and smartphone location, or (iii) due to an abrupt termination and a restart of the client.

3.3 Task programming conventions

Sensing tasks can be developed like any other interface-less Java program for Android. Of course, the programmer must be familiar with the API for accessing the special resources and sensors of the smartphone (for which there is ample documentation and a good emulator that runs on PCs).

To keep task programming as close as possible to native Android development, we do not introduce any own libraries or APIs. A drawback of this approach is that the task cannot invoke any client functions. Instead, the client retrieves the task's data and state through explicit invocations, following a polling scheme. This comes with the known disadvantage of trading-off timeliness with (useless) resource consumption. However, given that *EasyHarvest* is not intended for real-time applications, a polling period in the order of several tens of seconds or even a minute is perfectly acceptable.

When programming a sensing task for *EasyHarvest*, a few programming conventions have to be followed. Specifically, the task must be a Java class that implements a set of specific public methods, listed in Table 2. These are briefly discussed in more detail below. Note that the client does not create a separate thread for calling each of these methods, but invokes them from its own thread.

Public Methods of a Sensing Task	Description
`void onStart (Contect c, ObjectInputStream s);`	Initialize the task state and install listeners for the sensors of the smartphone. If the task has previously saved its state, this can be retrieved via s(if not NULL) .
`void onStop();`	Release all the resources held by the tasks and unregister all listeners so that it can be gracefully stopped.
`ArrayList<String> getData();`	Return the data that was produced so far by the task; this will be eventually uploaded to the server behind the scenes and in a lazy and incremental way, subject to Internet connectivity.
`boolean saveState (ObjectOutputStream s);`	Save task state information that is required on a subsequent restart; the return value indicates whether any state was actually saved.

Table 2: Public methods expected from sensing tasks

Method `onStart()` initializes the internal state of the task and registers listeners for the sensors the application wishes to employ. It is invoked when the task is started for the first time, and in all subsequent restarts. The first parameter is the execution context of the Android environment (required to gain access to key system resources, e.g., sensors). The second parameter is a data stream for reading previously saved state (this is NULL if there is no state to read).

Method `onStop()` releases the system resources held by the task and unregisters all sensor listeners. This method is called when the client wants to pause or permanently remove the task.

Method `getData()` returns the data that was produced by the task since the last invocation of this method. To avoid serialization/code dependencies between the task, the server and external application programs, data is returned in the form of strings (their serialization does not require any user-level classes). It is the responsibility of the task to buffer data until they are retrieved by the client.

Method `saveState()` saves the state of the task, so that it can be retrieved in a restart. It is invoked by the client before a task is paused, but also at regular intervals in order to deal with abrupt terminations. Its parameter is a data stream for writing state information. The return value indicates whether any data was actually written.

There is no method for invoking the task's "core" sensing and data processing logic. Tasks are expected to exploit the event-oriented sensing model of Android in order to receive and process sensor readings using appropriately programmed event listeners, which are up-called under the control of the operating system.

```
public class Demo implements LocationListener {
    … // var declarations
    public void onStart(Context c, ObjectInputStream s) {
        data = new ArrayList<String>();
        …
        timer = new CountDownTimer(…) {
            … // other methods
            public void onFinish() {
                data.add(lastLoc.toString());
            }
        };
        …
        locMgr = (LocationManager) c.getSystemService(…);
        locMgr.requestLocationUpdates(…, this);
    }
    public ArrayList<String> getData() {
        ArrayList<String> tmp = data;
        data = new ArrayList<String>();
        return tmp;
    }
    public boolean saveState(ObjectOutputStream s) {
        return false;
    }
    public void onLocationChanged(Location loc) {
        lastLoc = loc;
        timer.cancel();
        timer.start();
    }
    … //other methods
}
```

Figure 3: Code snippets

Figure 3 shows code snippets from a simple sensing task that is part of an application that finds the locations where people stay for some time. The task uses the location sensor service of Android to get notified about position changes, and a timer that is reset when the position changes. In this case, there is no need to save/restore any state.

Chapter 4 User Interaction

4.1 Server

The server's task management interface (Figure 4) is accessed via a browser. To submit a task, the user uploads the source code, which is compiled and checked for the mandatory methods. The user can then specify a daily time period and geographical region for the task's operation, and start/pause or withdraw the task, as desired. The progress of task execution can also be inspected at any point in time.

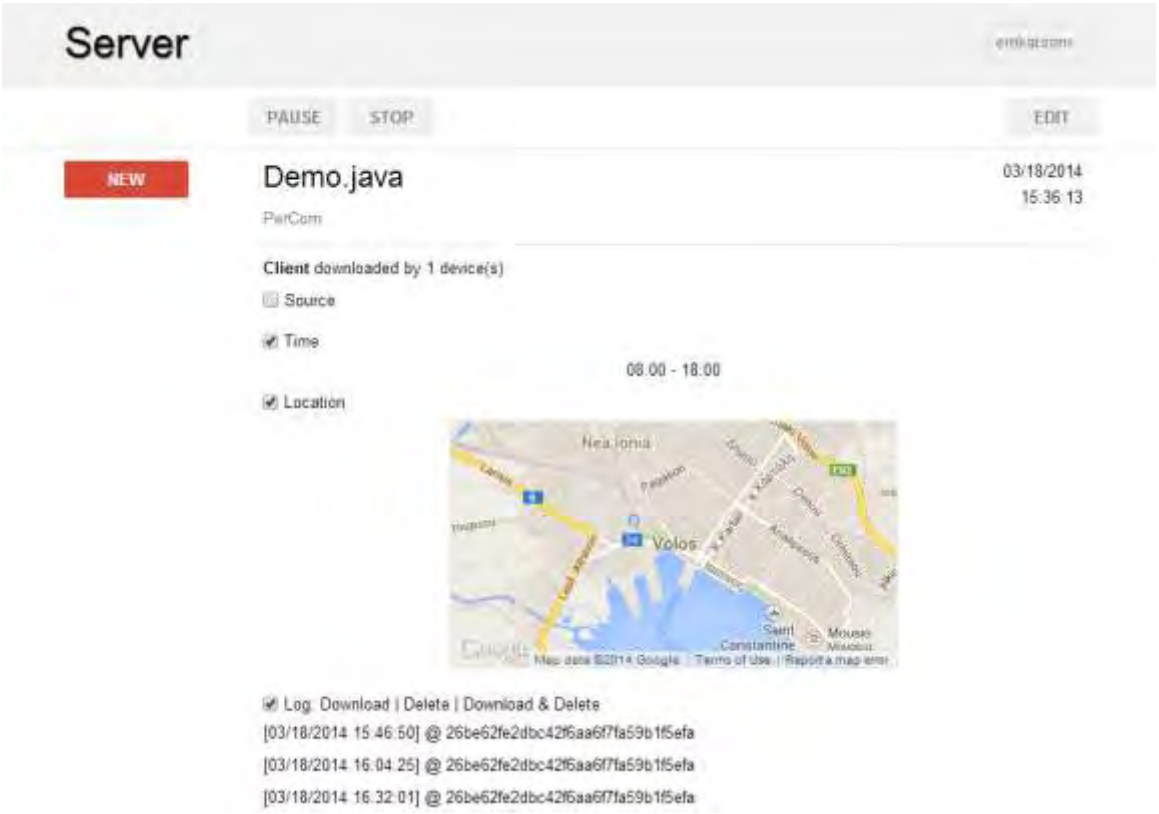

Figure 4: Server task management interface

4.2 Client

On the smartphone, the client provides a GUI for setting basic preferences regarding the local execution of tasks. These settings hold for all tasks. The idea is for the user to define them once (perhaps change them very infrequently), and not to be further bothered with the issue of task execution. Note that having different settings for each task would mean that the user would actually have to be notified about and configure every single task that executes on the smartphone.

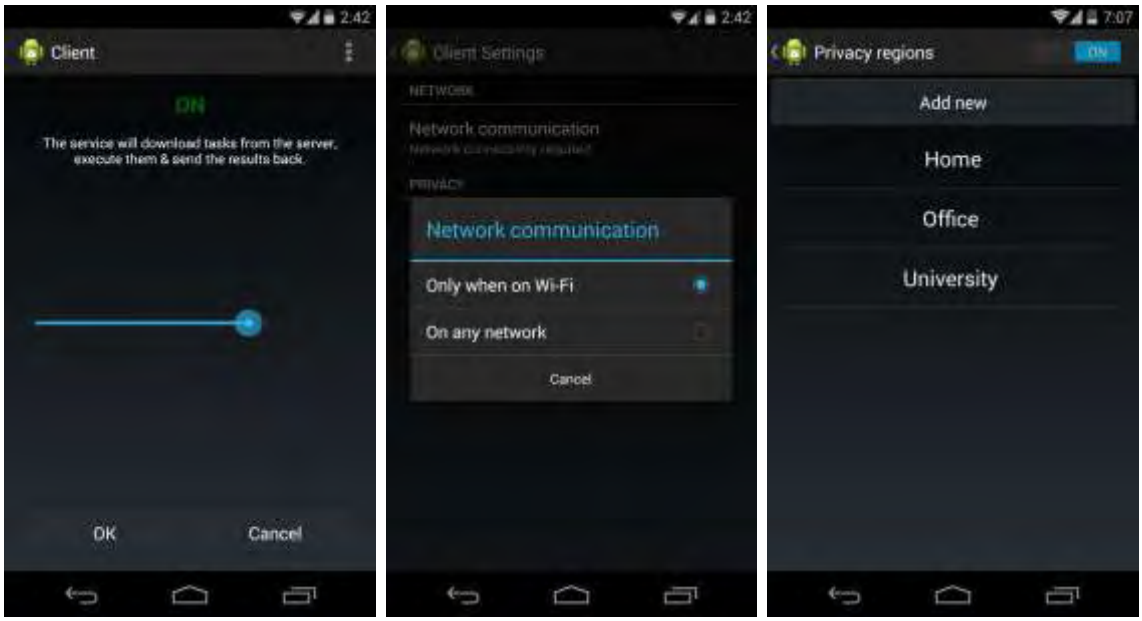

Figure 5: Client settings for the activity level, network and privacy regions options

Currently, there are three control aspects (Figure 5 shows indicative snapshots):

- Activity level (resource usage) of the client. The respective setting affects how often the client communicates with the server (when there is Internet connectivity), and the frequency at which the task is polled for its data and state.

- Network options for the client-server interaction. Specifically, the user can let the client communicate only via Wi-Fi, or over any network, including the cellular network.

- Privacy regions. The device owner may define areas where the client should stop any background activity. This feature is controlled by a dedicated switch under the privacy settings section, inside client's settings screen. By accessing the privacy regions setting section, the user can view the list of the areas he/she has defined, view/edit a specific region or add a new one.

Chapter 5 Evaluation

For the time being, we have deployed the *EasyHarvest* server on the cloud and installed the client on our smartphones. We have performed a number of small experiments to test and debug the framework. Here, we briefly report on one of those experiments.

```
[Wed Mar 19 08:13:44 EET 2014] @ 26be62fe2dbc42f6aa6f7fa59b1f5efa:
Location[gps 39.356651,22.957466 acc=53 et=+1d17h59m36s907ms alt=64.9000015258789 vel=0.0
{Bundle[mParcelledData.dataSize=44]}]
[Wed Mar 19 09:37:53 EET 2014] @ 26be62fe2dbc42f6aa6f7fa59b1f5efa:
Location[gps 39.359925,22.949063 acc=37 et=+1d19h27m8s500ms alt=46.79999923706055 vel=1.25 bear=249.1
{Bundle[mParcelledData.dataSize=44]}]
[Wed Mar 19 16:00:36 EET 2014] @ 26be62fe2dbc42f6aa6f7fa59b1f5efa:
Location[gps 39.357854,22.949569 acc=40 et=+2d1h51m33s67ms alt=42.70000076293945 vel=0.0
{Bundle[mParcelledData.dataSize=44]}]
[Thu Mar 20 08:17:52 EET 2014] @ 26be62fe2dbc42f6aa6f7fa59b1f5efa:
Location[gps 39.356537,22.957277 acc=129 et=+2d18h5m20s889ms alt=44.099998474121094 vel=1.25 bear=149.9
{Bundle[mParcelledData.dataSize=44]}]
[Thu Mar 20 09:42:09 EET 2014] @ 26be62fe2dbc42f6aa6f7fa59b1f5efa:
Location[gps 39.360885,22.949883 acc=25 et=+2d19h31m14s474ms alt=92.19999694824219 vel=0.0
{Bundle[mParcelledData.dataSize=44]}]
[Thu Mar 20 13:35:41 EET 2014] @ 26be62fe2dbc42f6aa6f7fa59b1f5efa:
Location[gps 39.361364,22.947560 acc=29 et=+2d23h22m11s921ms alt=43.70000076293945 vel=0.0
{Bundle[mParcelledData.dataSize=44]}]
[Thu Mar 20 14:38:33 EET 2014] @ 26be62fe2dbc42f6aa6f7fa59b1f5efa:
Location[gps 39.359780,22.949295 acc=76 et=+3d0h29m27s571ms alt=71.5 vel=0.25 bear=269.3
{Bundle[mParcelledData.dataSize=44]}]
[Thu Mar 20 16:15:00 EET 2014] @ 26be62fe2dbc42f6aa6f7fa59b1f5efa:
Location[gps 39.356737,22.957332 acc=31 et=+3d2h2m49s910ms alt=45.099998474121094 vel=0.0
{Bundle[mParcelledData.dataSize=44]}]
```

Figure 6: Experiment raw data output

For this experiment we used a sensing task (Figure 3) that records locations where the user stays for more than a specified amount of time. Figure 6 saws the raw data output generated, representing location coordinates that are actual places in Volos, part of a daily routine. Those coordinates, are pin-pointed on a map in Figure 7.

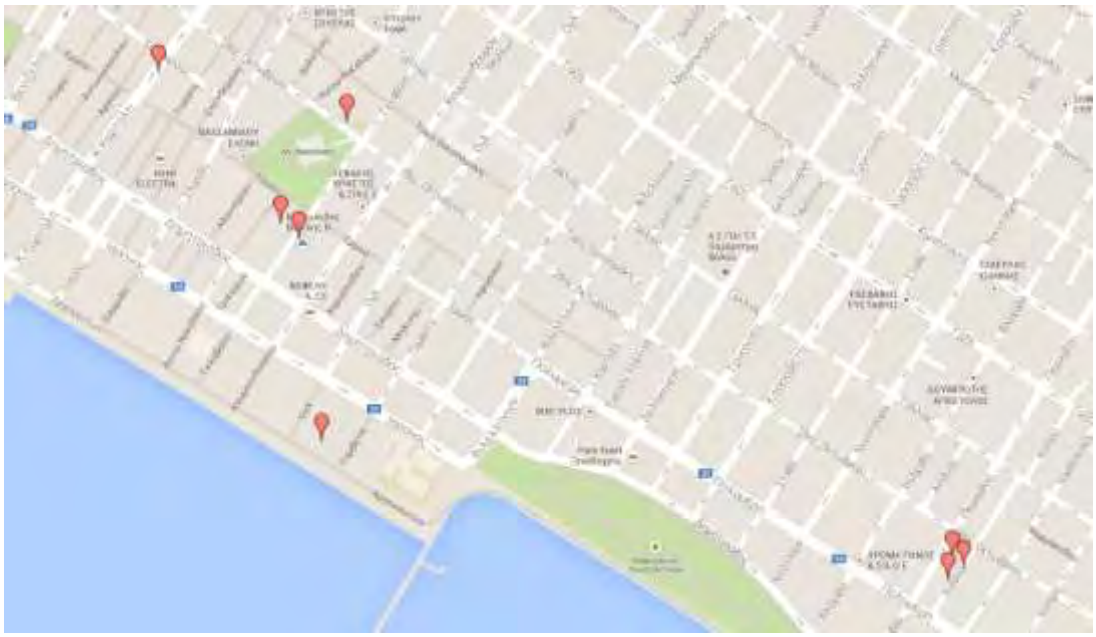

Figure 7: Experiment data visualized on map

Chapter 6 Related Work

6.1 Architecture

Locale [4] and Tasker [5] are Android applications that let users specify context aware tasks by defining conditions under which their phone's settings should change. They offer many pre-defined conditions and actions that users can select to build tasks through a suitable user interface. AIRS [6] is a life-logging platform for Android that can record data from a large variety of phone-internal and attached sensors, which can in turn be used to observe movement, environmental conditions, mood, social activities and interests. The user can monitor, review, process and visualize this data on the phone via the Storica application.

Microsoft's on{X} [7] allows users to control and extend the capabilities of their smartphones remotely. The system consists of a website and a native application: the former lets users select a script from a variety of "recipes", while the latter executes the script on the phone. Users can create their own recipes, from scratch or by modifying existing ones, and share them with others. Scripts are programmed in JavaScript and access the sensors of the smartphone via a corresponding API. The Code-In-The-Air platform (CITA) [8], allows high-level task/activity scripts to be composed out of lower-level ones. When a script is submitted to the system it is compiled into server-side and mobile code, with the latter being shipped and executed on the user's phone in a transparent way. The interaction between the server and the mobile device is done via an asynchronous message delivery service, in similar way "push notifications" currently work for smartphones [9].

The Mobile Sensor Data Engine (MOSDEN) [10] is a remote sensing framework designed for smartphones. It uses the concept of plug-ins to enable a flexible integration of on-board and external sensors without recompiling and/or redeploying the system. In addition, MOSDEN implements generic sensing and data storage functions directly on the smartphone, letting each device act as a server that can be used by one or more applications. Mobile edge capture and analysis (MECA) [11] is an infrastructure-based middleware for social sensing applications. The backend provides a high level abstraction of phenomena, which the programmer can use to express the application's data needs in a declarative way. Based on the application's needs, the backend sends data collection requests to the edge nodes of the system (e.g., base stations in cellular networks), which in turn manage the data collection process for the mobile devices in their jurisdiction, and perform simple data processing tasks. MineFleet [12] is a vehicle performance data mining system designed for commercial fleets. The devices installed on vehicles are custom-built, and process data streams locally using various statistical and data stream mining algorithms. Data is stored on the devices and is transmitted to an external server for further processing when network connectivity is available.

Locale, Tasker and AIRS target end-users who wish to exploit the sensing capabilities of smartphones for their own purposes. In contrast, *EasyHarvest* is about turning mobile devices into a large-scale sensing platform for applications that can benefit an entire community. We also wish for this to happen with a minimal involvement of the device owners.

Our design approach and programming model have many similarities with on{X} and CITA. However, their focus is on monitoring/controlling individual smartphones, rather than engaging a large number of devices to support crowdsensing applications. Another difference is that *EasyHarvest* tasks are written in Java for Android, instead of JavaScript based on a custom API. This offers a familiar development experience to the Android programmer, and enables the exploitation of new smartphone features right out of the box; of course, the downside is that one is limited to Android phones. Regarding the interaction between the mobile device and the server, unlike CITA's "push notification" service, we opt for a REST-based solution that allows the client to contact the server at its own discretion, while handling intermittent connectivity and disconnected operation problems in a transparent way.

Similar to *EasyHarvest*, MOSDEN and MECA target crowdsensing applications that can potentially involve a large number of mobile devices. Their main advantage is that sensor data, and even some basic processing in the case of MECA, can be shared between different applications. In case a large number of applications rely on the same devices, this achieves a better economy of scale compared to the more vertical approach of *EasyHarvest* where each application produces and has access only to its own (private) data. The advantage of *EasyHarvest* is that it does not treat mobile devices as dump data producers, and gives the developer more flexibility in terms of the application-specific processing that can be done locally, on the smartphone, before data is sent to the server (from where it can be retrieved for further processing. At the other end of the spectrum of large-scale sensing systems, MineFleet employs custom and dedicated devices with hardwired data processing logic. In contrast, *EasyHarvest* relies on privately owned smartphones to perform the sensing, and can be used to deploy different application-specific sensing and data processing logic in a flexible way.

6.2 Privacy

Privacy-preserving participatory sensing has been widely addressed by the research community in the past [14] including privacy of data itself, of data source identity and of user location. Mechanisms based on k-anonymity [15] can be applied to protect the location privacy of the participants who upload reports. The key idea behind k-anonymity is to build groups of k participants or reports such that they share a common attribute (e.g. k participants located in the same district), rendering them indistinguishable from each other. Different methods can be used to find an appropriate and common attribute in order to construct groups of k users. These methods can be classified into the two main categories of generalization and perturbation [16].

A risk to k-anonymity is the possibility of homogeneity attacks, as outlined in [17]. Such attacks exploit the monotony of certain attributes to identify individuals from the set of k participants. The authors thus present an extension to k-anonymity, termed l-diversity, which additionally requires the group members to provide at least l different values for the sensed attribute of interest. As a result, at least l distinct values for the sensitive attributes are present within each user group, which represents an effective countermeasure to homogeneity attacks.

The user-side privacy-protection scheme presented in [18], dynamically adjusts its parameters in order to meet personalized location-privacy protection user preferences. This approach estimates locally, in real-time, the expected privacy level at the node, which enables it to adapt its privacy parameters with respect to its mobility and satisfy individual privacy constraints. Furthermore, an event linkability graph is maintained at each node where an observable event corresponds to a data item associated to a particular location, which is sent to the aggregation server (AS). Finally, location obfuscation is employed for confusing the AS about the actual location of the sensed data.

EasyHarvest is meant to be used by communities with registered users who trust each other. Thus, we assume the absence of adversaries and malicious actions. k-anonymity and l-diversity apply to a large scale of users and require that some specified standards are met in order to allow the dissemination of data. User-side adaptive protection is more similar to our current approach, which offers users personalized custom privacy according to their preferences while it does not affect the accuracy and precision of the reported data. However, our platform does not obfuscate the location of the user which enhances the reliability of client-server communication.

Chapter 7 Conclusion

The *EasyHarvest* prototype is a first step in supporting crowdsensing applications using smartphones. Developers can write sensing tasks without caring about sporadic and intermittent Internet connectivity, and smartphone owners can control task execution on their device while being relieved from repeated downloads, installations and configurations of individual sensing applications. Our prototype is designed for long-lived, opportunistic sensing and data collection scenarios, without any real-time constraints.

As next steps, we are considering different task assignment policies, and wish to support more client control options, e.g., to take into account the battery and temperature level, or let the user define quiet hours while no sensing should be performed in the background. We also think about running application-specific logic on the server, with support for a backwards control flow towards task instances, possibly as an extension of the property update scheme. Furthermore, we plan to investigate the use of a virtual machine to achieve task portability and to let the client control tasks in a better way. Last but not least, we wish to evaluate our system with some real users.

References

[1] M. Duchon, K. Wiesner, A. Mueller, C. Kinnhoff-Popien. Collaborative Sensing Platform for Eco Routing and Environmental Monitoring, Intlernational Conference on Sensor Systems and Software, 2012.

[2] S. Eisenman, E. Miluzzon, N. Lane, R. Peterson, G.-S. Ahn, A. Campbell. Bikenet: A Mobile Sensing System for Cyclist Experience Mapping, ACM Transactions on Sensor Networks, 6(1), 2009.

[3] M. Fereira, R. Fernandes, H. Conceicao, P. Gomes, P. d'Orey, L. Moreira-Matias, J. Gama, F. Lima, L. Damas. Vehicular Sensing: Emergence of a Massive Urban Scanner, International Conference on Sensor Systems and Software, 2012.

[4] Locale web site, www.twofortyfouram.com.

[5] Tasker web site, tasker.dinglisch.net.

[6] AIRS/Storica web site, tecvis.co.uk.

[7] on{X} web site, www.onx.ms.

[8] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. Code In The Air: Simplifying Sensing and Coordination Tasks on Smartphones. HotMobile 2012.

[9] Apple Push Notification Service, developer.apple.com/technologies/ios/features.

[10] P. P. Jayaraman, C. Perera, D. Georgakopoulos, and A. Zaslavsky. Efficient Opportunistic Sensing using Mobile Collaborative Platform MOSDEN, 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, 2013.

[11] F. Ye, R. Ganti, R. Dimaghani, K. Grueneberg, and S. Calo. MECA: Mobile Edge Capture and Analysis Middleware for Social Sensing Applications, 21st International Conference Companion on World Wide Web, 2012.

[12] H. Kargupta, K. Sarkar, and M. Gilligan. MineFleet: An Overview of a Widely Adopted Distributed Vehicle Performance Data Mining System, 16th ACM International Conference on Knowledge Discovery and Data Mining, 2010.

[13] M. Katsomallos and S. Lalis. EasyHarvest: Supporting the Deployment and Management of Sensing Applications on Smartphones, 1st IEEE Workshop on Crowdsensing Methods, Techniques, and Applications, 12th IEEE International Conference on Pervasive Computing and Communications, 2014.

[14] D. Christin, A. Reinhardt, S.S. Kanhere, and M. Hollick. A Survey on Privacy in Mobile Participatory Sensing Applications, International Journal of Systems and Software, vol. 84, no. 11, 2011.

[15] L. Sweeney. K-anonymity: A Model for Protecting Privacy, International Journal of Uncertainty, Fuzziness, and Knowledge-Based Systems, vol. 10, no. 5, 2002.

[16] K. L. Huang, S. S. Kanhere, and W. Hu. Preserving Privacy in Participatory Sensing Systems, International Journal of Computer Communications, vol. 33, no. 11, 2010.

[17] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. L-diversity: Privacy beyond K-anonymity, ACM Transactions on Knowledge Discovery from Data, vol. 1, no. 1, 2007.

[18] B. Agir, A. Papaioannou, R. Narendula, K. Aberer, and J.-P. Hubaux. User-side Adaptive Protection of Location Privacy in Participatory Sensing, International Journal of GeoInformatica, vol. 18, no. 1, 2014.