Pogo, a Middleware for Mobile Phone Sensing

Niels Brouwers, Koen Langendoen

Pogo, a Middleware for Mobile Phone Sensing

Niels Brouwers and Koen Langendoen

Delft University of Technology
{n.brouwers,k.g.langendoen}@tudelft.nl

Abstract. The smartphone revolution has brought ubiquitous, powerful, and connected sensing hardware to the masses. This holds great promise for a wide range of research fields. However, deployment of experiments onto a large set of mobile devices places technological, organizational, and sometimes financial burdens on researchers, making real-world experimental research cumbersome and difficult. We argue that a research infrastructure in the form of a large-scale mobile phone testbed is required to unlock the potential of this new technology.
We aim to facilitate experimentation with mobile phone sensing by providing a pragmatic middleware framework that is easy to use and features fine-grained user-level control to guard the privacy of the volunteer smart-phone users. In this paper we describe the challenges and requirements for such a middleware, outline an architecture featuring a flexible, scriptable publish/subscribe framework, and report on our experience with an implementation running on top of the Android platform.

Keywords: Mobile Middleware, Mobile Phone Sensing, Mobile Test Beds.

1 Introduction

Modern smartphones are rapidly becoming ubiquitous and are even supplanting the desktop PC as the dominant mode for accessing the Internet [7]. They are equipped with a powerful processor and a wide range of sensors that can be used to infer information about the environment and context of a user. These capabilities and the rapidly growing number of smartphones offer unique opportunities for a great number of research fields including context-aware computing [25], reality mining [11], and community sensing [5, 20]. Basically, the smartphone revolution will enable experimentation at scale in real-world settings; an exciting prospect.

To date most efforts have focused on building monolithic mobile applications that are tested in small-scale lab environments. Real-world deployment is a labor-intensive process, which involves recruiting participants, acquiring devices, deploying software updates, and so on. Because the barrier for deployment onto a large number of devices is high, many applications and experiments are never able to leave the desk of the researcher. This is a serious drawback that needs to be addressed as history has shown that small-scale systems often show quite different behavior when put to the test in the real world [21].

The challenge for running large-scale experiments is no longer the hardware, as affordable smart phones equipped with various sensors are ubiquitously available, but the software engineering involved in creating and installing the application code to read, process, and collect the desired information. Indiscriminately gathering all possible sensor data on the device and sending it back to a central server is infeasible due to bandwidth, power consumption, and privacy concerns. Hence, on-line analysis and filtering is required [6]. Since researchers rarely get their algorithms right on the first try, quick (re-)deployment of mobile sensing applications is essential for the experimental process, but typical application stores are not suitable for this. Finally, there is a large administrative overhead involved with managing large groups of test subjects, especially when multiple experiments need to be carried out, which is something which ideally should be hidden from scientists and end-users.

We strongly believe that providing an easy to use, large-scale testbed of mobile phones carried by ordinary people will be a game changer for many types of experimental research. Overall our aim is to unlock the true potential of mobile phone sensing by developing a research infrastructure that can be used by a broad range of researchers to easily and quickly deploy experiments. In this paper we introduce *Pogo*, a middleware infrastructure for mobile phones that provides easy access to sensor data for the research community. By installing the *Pogo* middleware, which is as simple as downloading an application from the application store, a phone is added to a shared pool of devices. Researchers can request a subset of those devices, and remotely deploy their own executable code onto them. We make the following contributions in this work:

1. We present the design rationale behind *Pogo*, motivate our choices, and compare them against related work.
2. We describe the implementation of our middleware and demonstrate its feasibility it using a real-world Wi-Fi localization experiment.
3. We propose and evaluate a novel scheme for automatically synchronizing data transmissions with that of other applications, dramatically reducing energy consumption.

The rest of this paper is structured as follows. We introduce related work in Section 2, and present our design choices in Section 3. Section 4 describes the implementation of *Pogo*. We evaluate our middleware in Section 5, and finally conclusions and future work are presented in Section 6.

2 Related work

In this section we introduce several sensor processing and collection frameworks that have been proposed in the fields of context-aware computing and mobile phone sensing, as well as some systems that have been developed for tracking smartphone usage. We will make a detailed comparison between these works and *Pogo* when we present its design in Section 3.

Context-aware computing is a field that uses sensor data to infer information about the *context* of a user. Examples of contexts are user location, emotional state, and transportation mode. Middleware built for this purpose aids developers by providing sensor abstractions and off-the shelf classifiers, and help reduce energy consumption by scheduling sensors intelligently.

Jigsaw [23] is a framework for continuous mobile sensing applications. It uses a pipeline architecture, with different pipes for each sensor, and has the ability to turn individual stages on or off depending on need and resource availability. It has classifiers for accelerometer and audio data built-in, and reduces power consumption by scheduling GPS sampling in a smart way. The *Interdroid* platform [19] aims to provide a toolkit for the development of 'really smart' applications, and focuses on integrating mobile phones and cloud computing. Applications contain a client and server part, the latter of which can be uploaded to a remote server in the cloud where it can run to support the client. *Mobicon* [22] proposes a *Context Monitoring Query* language (CMQ), which can be used by applications to specify the type of context information they require. The middleware then intelligently plans sensor usage in order to reduce power consumption. However, the data processing in context-aware systems is geared towards assisting the user, and therefore do not include functionality for collecting data at a central server or for remotely deploying sensing tasks.

Mobile phone sensing middleware aims to turn smartphones into mobile sensors. The aim is to collect data about user behavior or the environment in which a user moves around, and send it to a central point for further analysis. One such project is *AnonySense* [8]. Tasks are written in a domain-specific language called *AnonyTL*, which has a Lisp-like syntax. These tasks are matched to devices using predicates based on the context of the device, such as its location. Another relevant project is *Cartel* [4], which is a software and hardware infrastructure comprising mobile sensing nodes on cars. Remote task deployment, although limited in nature, is supported through runtime configuration of parameters like the type and rate of sensor information reported by the mobile devices; continuous queries written in SQL submitted to a central server further process and filter this data, providing additional adaptability. A much more flexible approach is offered by *PRISM* [10], which allows deployment of executable binaries at the mobile devices themselves. Method call interposition is used to sandbox running applications for security and privacy reasons. *Crowdlab* [9] proposes an architecture for mobile volunteer testbeds that allows low-level access to system resources, and employs virtualization technologies to run sandboxed applications concurrently with the host operating system.

Phone Usage Traces *SystemSens* [14] and *LiveLab* [26] are end-to-end logging tools for measuring smart phone useage, and can be useful for diagnosing other running applications. They collect data about wireless connectivity, battery status, cpu usage, screen status, and so on. Both offload the collected traces to a central server only when the phone is charging in order to save energy. *MyExperience* [15] is a more flexible system that can also capture sensor data and user context, and is even able to ask the user for feedback through an on-screen

survey. MyExperience can be configured using XML files with support for scripting, and behavior can be updated in the field by sending scripts through SMS or e-mail. Output is stored in a local database that is synchronized periodically with a central one.

3 Design

In this section we look at several design aspects of *Pogo*, compare alternative options and discuss how they fit in with the related work, and finally motivate our choices. Note that many of these considerations have architectural consequences, as is reflected in Section 4.

3.1 Testbed Organization

The most straightforward way to structure a testbed is to have a central server and a set of mobile devices in a master-slave setup. The phones collect and process data locally, and send it to the server where it is stored, possibly after further processing. This model is followed by most middleware, including PRISM [10] and AnonySense [8]. However, such a strongly centralized server component must also have a front-end where scientists can upload scripts, download data, and manage their device pool, which introduces a considerable implementation overhead. Moreover, since researchers share devices between them and multiple sensing applications run concurrently on each device there is an inherent many-to-many relationship between researchers and end-users.

We have therefore opted for a design where both parties, the researchers and the test subjects, run the *Pogo* middleware, with a central server acting only as a communications switchboard between them. This way researchers can interact directly with end-user devices without having to go through a web interface or logging into a server. There are three types of stake holders in a *Pogo* testbed. First, the *device owners* contribute computational and sensing resources to the system by running *Pogo* on their phones. The *researchers* run *Pogo* on their computers and consume these resources by deploying experiments. The *administrator* of the testbed decides which devices are assigned to which researchers. In a way the administrator acts as a broker who brings together people who offer and consume resources. The connections between researchers and device owners are double blind, with the administrator having only personal information about the researchers who use the system.

3.2 Deployment

An important consideration is how experiments are delivered to the mobile devices. Remote deployment is a basic functionality of any testbed and supports the development of new algorithms and techniques by enabling researchers to test hypotheses and benchmark solutions. It is, however, also a vital requirement

for running long-term sensing studies, which may need to deal with changing requirements, new hardware developments, and new insights, or simply require maintenance to correct programming errors.

Broadly speaking there are two methods of deployment found in literature. *Pull-based* systems present the user with a list of applications that can be downloaded. Common examples are the iPhone *App Store*[1] and Android's *Play Store*[2]. The choice for which application runs on which device lies solely with the user. In contrast, *push-based* systems allow researchers to send their applications to remote devices without interaction from the user. This can be manual, like in Prism [10] or Boinc [1], or automatic based on device capabilities or context, as is the case with AnonySense [8].

Note that pull-based systems often have a push component, in the form of application updates that are installed automatically. The Play Store has an updating mechanism where new versions can be pushed to Google's servers by developers. End-users that have the application installed will be notified of such updates and can choose to either update manually, or let Android manage this automatically. Depending on the updating method, it may take anywhere from a few hours to several days for a device to get the latest version. In our experience, these long update times are not suitable for quick redeployment and experimentation. What is more, the update process on the phone stops the application if it is running and it has to be restarted by the user. This means that automated updates result in regular downtime even with the most committed users due to the time it takes for them to notice that the application has been killed.

For *Pogo* we have chosen a push-based system because we believe it is most suitable for rapid deployment and experimentation. Of course, this means that users are not able to choose what kind of applications are running on their phones. We therefore allow users to select the types of information their wish to share, so that they retain full control over their own privacy.

3.3 Participation

Participation by the general public is an important aspect of our approach and we employ several strategies and incentives to attract users to our testbed. First of all the barrier for participation is kept as low as possible. The goal is to have volunteers just click once on the *Pogo* icon in their application store, which will automatically start the download, installation, and execution of the middleware on their phone. There is no registration process after the application has been installed. This implies an opportunistic approach in which the middleware runs silently in the background; only if a user wants to change the default settings (e.g., about privacy) or remove the middleware completely does he need to take action. We guarantee complete anonymity and give the user full control over what information he wishes to share, and these settings can be changed at any time from the application interface.

[1] `http://www.apple.com/itunes`
[2] `http://play.google.com/`

We expect that research institutions will recruit nodes among employees and students, possibly rewarding the latter group with study credit. We are also investigating monetary incentives such as Amazon's Mechanical Turk[3]. We have a central server that can keep track of when devices are online and what data they are sharing, which would be the basis for assigning rewards. A third option is to distribute smart phones for free with the understanding that the recipients run the middleware and share their data [17].

3.4 Experiment Description

There are several approaches to writing mobile sensing experiments. Runtime-configurable systems such as Cartel [4], and domain-specific languages like CMQ [22] and AnonyTL [8], are easy to execute and sandbox. Moreover, notation is generally short and concise, and accessible to researchers and programmers with little domain experience. On the other hand, deployment systems like PRISM [10] and CrowdLab [9] allow native applications to be deployed on remote nodes, giving total flexibility, but at the cost of requiring complex sandboxing techniques, like method call interposition or hardware virtualization, to keep malicious or malfunctioning code from degrading user experience or breaking privacy.

We feel that the expressiveness of general programming languages is neccesary if *Pogo* is to support the wide range of applications that we envision. The example application that we describe in Section 4.1 implements a clustering algorithm that could not be expressed in a simple DSL or query language. While it is true that middleware can be extended with new functionality if desired, doing so would require updating the application for the entire installed base, which is exactly the kind of deployment overhead we wish to avoid.

We argue that simplicity and flexibility do not have to be competing constraints. *Pogo* applications are written in JavaScript, a popular and accessible programming language. We expose a small, yet powerful programming API of only 11 methods that abstracts away the flow of information between sensors and scripts, and between phones and data collecting PCs. In this way, developers do not need to know anything about smartphones or Android in order to be able to write *Pogo* experiments. Sandboxing is straightforward as the scripting runtime can be used to control what functionality the application is allowed to use.

3.5 Programming Abstractions

The choice of a generic programming language over a DSL means that we must provide an application programming interface (API) to developers that exposes the kind of functionality required for mobile sensing applications. We identified the following requirements. First, applications need to be able to *access sensor data*, either by reading them directly or by listening for updates. Second, there

[3] `https://www.mturk.com/mturk/welcome`

should be a facility for *periodically executing code*. Third, a means of *communicating with a central point* (the researcher) is required so that findings can be reported. Finally, some means of *breaking up large experiments into smaller components* is not a functional requirement per se, but makes complex applications such as the one described in Section 4.1 more manageable.

Starting from the simplest option, we have considered exposing a rich API to the scripting runtime. This approach is taken by PhoneGap[4], a popular toolkit for developing portable mobile applications with HTML5 and JavaScript. For example, the accelerometer can be read by calling the `navigator.accelerometer.getCurrentAcceleration` method. However, such an API grows quickly as more sensors are added, and a lot of 'glue' code is required to interface between the native platform API and JavaScript. Moreover, consider the case where two scripts are running on the same device, and both are requesting a Wi-Fi access point scan at regular intervals. It would be sufficient to scan at the highest of the two frequencies to serve both scripts, but this energy-saving optimization cannot be made without some form of coordination.

The API requirements essentialy boil down to an exchange of *data* and *events* between loosely coupled entities; sensors, scripts, and devices. Two popular abstractions for this type of communication are *tuple spaces* [16] and *publish-subscribe* [13]. A tuple space is a shared data space where components interact by inserting and retrieving data. In a publish-subscribe system, components *publish* information to a central authority, the so-called *message broker*. Other components can then *subscribe* to this information and are notified when new data become available. In terms of capabilities, the two techniques are roughly equivalent [3]. We have chosen publish-subscribe for *Pogo* mostly because of implementation advantages. First, in a publish-subscribe system a sensor component can easily query whether there are other components interested in its output. If not, the sensor can be turned off to save energy. Second, subscriptions in *Pogo* optionally carry parameters that can be used to add details such as a requested sampling rate. Finally, the model is event-based, which fits our choice for JavaScript.

4 Implementation

In this section we describe our implementation of *Pogo*, which is written in Java and runs on Android smartphones as well as on desktop PCs and servers. The Android platform was chosen because of its ubiquity (at the time of writing Android had 50.9% market share[5]), and because it supports the type of background processing required for mobile phone sensing tasks. *Pogo* runs on Android 2.1 and up, and currently stands at 10,666 source lines of Java code, of which 5,170 lines are common, 2,948 are Android-specific, and 2,548 belong to the PC version.

[4] `http://www.phonegap.com`
[5] `http://www.gartner.com/it/page.jsp?id=1924314`

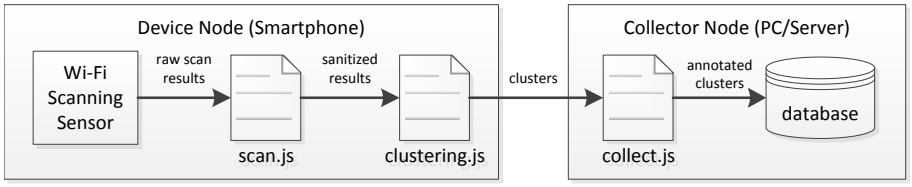

Fig. 1. Data-flow of the localization application. The `scan.js` script requests Wi-Fi access points scans from the Wi-Fi sensor, sanitizes them, and sends them to `clustering.js`. This script clusters the scans and sends cluster characterizations to the `collect.js` script running on the collector node, which in turn pushes them into a database.

4.1 Example Application

Before we describe our implementation in detail, we believe it is helpful to present an application to both give a concrete example of the type of experiments we envision, as well as to provide a running example with which to illustrate the various implementation details. We chose a meaningful, real-world localization application for this purpose. The goal of the application is to find locations where the user spends a considerable amount of time, such as the home, the office, and so on. We do this by periodically sampling Wi-Fi access points, and clustering these scan results based on similarity. The clusters found in this way characterize a 'place' where the user dwelled.

Figure 1 shows the data flow of the application. The `scan.js` script obtains scan results from the Wi-Fi scanning sensor. It sanitizes the raw results by removing locally administered access points, and normalizes received signal strength (RSSI) values so that 0 and 1 correspond to -100 dBM and -55 dBM respectively. These values are then picked up by the `clustering.js` script that extracts clusters (locations) using a modified version of the DBSCAN clustering algorithm [12]. The modification in this case is that we use a sliding window of 60 samples from which we extract core objects. Clusters are 'closed' whenever a user moves away from the place it represents (when a sample is found that is not reachable from the cluster). The distance metric used is the cosine coefficient. When a cluster is closed, a sample is selected that best characterizes the cluster[6] and sent to the server along with entry and exit timestamps. The `collect.js` script running on the collector node collects these cluster characterizations and uses Google's geolocation service [18] to convert them into a longitude, latitude pair. The annotated places are then pushed into a database.

This example illustrates a number of key features of *Pogo*. The location clustering is performed on the device so as to avoid sending raw access point scans to the collector and hence minimize communication cost, which shows the advantages of on-line processing. The flexibility of our scripting environment allows us to write complex sensing applications and even run custom clustering

[6] The nearest neighbour to the mean of all scan results is selected.

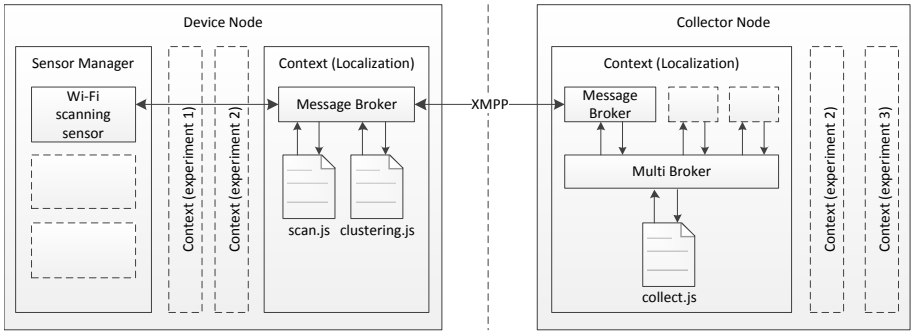

Fig. 2. A device- and collector node running the localization application.

algorithms when desired. Applications can easily be broken down into a set of cooperating scripts, and communication between them flows seamlessly across the wireless networking boundary.

4.2 Node Architecture

In *Pogo* both the researchers and device owners are running the same middleware; the only functional difference between them is that researcher nodes are operating in *collector* mode, which gives them the ability to deploy scripts. We therefore do not have to write an extensive server application. Instead, we use an off-the-shelf open source instant messaging server to manage communication between device- and collector nodes.

Figure 2 shows the anatomy of two *Pogo* node instances, a device and collector node running the example application presented in Section 4.1. Scripts belonging to a certain experiment run inside a so-called *context*, which acts as a sandbox; scripts can only communicate within the same experiment. Each context has a counterpart on a remote node, and communication between them flows over the XMPP protocol, as we will describe Section 4.6.

Each context has a *message broker* associated with it where scripts can subscribe to- and publish data. Since contexts have counterparts on remote devices, so do message brokers. The brokers on either end synchronize with each other so that the publish-subscribe mechanism works seamlessly across the network boundary. Since contexts on collector nodes can have more than one remote context associated with them, a *multi broker* is used to make the communication fan out over the different devices. Note that message exchange can only happen between a device- and collector node, device nodes can never communicate with each other directly.

Finally, sensors live inside a *sensor manager*. They are able to publish data to, or query subscriptions from, all contexts. All a script needs to do in order to obtain sensor data is to subscribe to it. This also works across the network; a script running on a collector node that subscribes to battery information will automatically receive voltage measurements from all devices in the experiment.

4.3 Publish-Subscribe Framework

Communication between sensors, scripts, and devices uses a topic-based publish-subscribe paradigm [13], where *messages* (events) are published on *channels*. For example, the Wi-Fi scanning sensor publishes its output on the `wifi-scan` channel, and scripts that wish to consume this data simply subscribe to this channel. Messages are represented as a tree of key/value pairs, which map directly onto JavaScript objects so that they can be passed between Java and JavaScript code seamlessly. Messages are serialized to JSON[7] notation when they are to be delivered to a remote node.

A subscription in *Pogo* can have a parameter object associated with it. Scripts can use this to be more specific about the information they are interested in. For example, a script may request location updates, but only from the GPS sensor. It can do this by subscribing to the `locations` channel using the `provider:'GPS'` parameter. Another example is the `scan.js` script in our running example, which requests access point scans every minute. The scanning interval in this case is also passed using the parameters (`interval:60000`).

Given the battery constraints of mobile devices it would be wasteful to have sensors draw power when their output is not being consumed. The framework therefore allows sensors to listen for changes in subscriptions to the channels they publish on. Sensors can enable or disable scanning based on this information, and change their behavior depending on the subscription parameters.

4.4 Scripting

Scripts are executed using Rhino[8], a JavaScript runtime for Java, which allows for seamless integration of the two languages. In the interest of security however, we hide the Java standard library and of course all of the Android API from the application programmer. Instead, we expose only a small programming interface, shown in Table 1.

The `setDescription()` and `setAutostart()` functions can be placed in the script body to set script parameters. If automatic starting of a script is turned off, it will not run until the user explicitly starts it through the UI. The description of the script will be shown in the UI as well. The `print()` function prints a debug message that can be viewed on the phone, while the `log` and `logTo` functions can be used to write lines of text to permanent storage. The `publish` and `subscribe` function expose the message passing framework to a script. The `parameters` argument to `subscribe` is optional and is used to add parameters to a subscription. For example, the following line:

```
1 subscribe('wifi-scan', handleScan, { interval : 60 * 1000 });
```

requests a wifi scan result once per minute. The returned `Subscription` object can be used to control whether a subscription is active or not. The `release` method deactives a subscription, while `renew` can be used to reactivate it at

[7] `http://www.json.org/`
[8] `http://www.mozilla.org/rhino/`

Table 1. *Pogo* JavaScript framework API

```
setDescription(description)
setAutoStart(start)
print(message1[, ...[, messageN]])
log(message1[, ...[, messageN]])
logTo(logName, message1[, ...[, messageN]])
publish(channel, message)
Subscription subscribe(channel, function[, parameters])
freeze(object)
object thaw()
String json(object)
setTimeout(function, delay)
```

a later time. Note that these methods have no effect when the subscription is inactive or active respectively.

An object can be 'frozen' with the `freeze` function, which means it will be serialized to permanent storage. Each script can have only one such object at any given time, and `freeze` will always overwrite any preexisting data. This stored object can be retrieved using `thaw`. These two methods make it possible to have data persist through script stop/start cycles and updates. The `json` function serializes an object to a string using JSON notation. The `setTimeout` method works in much the same way as it does in a browser, allowing a function to be scheduled for execution at some point in the future.

4.5 Event Scheduling

Handling (timed) events requires some special attention on mobile devices because power management has to be taken into account. When the screen is turned off and there are no ongoing activities such as a phone call being made, Android will put the CPU to sleep to conserve energy. Applications can prevent the CPU from going to sleep by acquiring a *wake lock*, and this is essential for many asynchronous sensing tasks. Consider the example where an application requests a Wi-Fi access point scan. If the CPU is not kept awake during the 1-2 seconds the process generally requires, the application will not be notified upon scan completion. When the CPU is in deep sleep, it can be awoken only by events such as incoming calls, or the user pressing a hardware button. Alternatively, an application may want to schedule a wake-up call periodically, which it can do by setting a so-called *alarm*.

The *Pogo* framework abstracts away the complexities of setting alarms and managing wake locks through a *scheduler* component that executes submitted tasks in a thread pool, and supports delayed execution. Using a thread pool has the advantage that components that execute code periodically do not have to maintain their own threads and are therefore more light-weight. A typical example is a sensor component that samples at a given interval. When there are no tasks to execute, the CPU can safely go to sleep.

The scheduler is also used when calling JavaScript subscription handlers and functions that have been scheduled for execution using the `setTimeout` method. A script can have multiple subscriptions, so in theory multiple Java threads could execute code belonging to the same script. However, since JavaScript does not have facilities to handle concurrency, the threads are synchronized so that only a single thread will run code from a given script at any time.

To keep incorrect or malicious code from locking up the system and draining the battery, all calls to JavaScript functions by the framework must complete within a certain timeframe. If the JavaScript function does not return in time, it is interrupted and an exception is thrown. The default timeout is set to 100ms.

4.6 Communication

Pogo relies on the XMPP protocol[9], which was originally designed for instant messaging. Using an instant messaging protocol is helpful because associations between devices and researchers can be captured as buddy lists, or *rosters* in XMPP parlance. These are stored at the central server and can therefore be easily managed by the testbed administrator. The XMPP server we use, Openfire[10], has an easy-to-use web interface for this purpose.

Mobile phones frequently switch between wireless interfaces as the user moves in- or out of range of access points and cell towers. Unfortunately there is no transparent TCP handover between these interfaces, causing stale TCP sessions and even dropped messages. This message loss problem is recognized in the XMPP community and although several extensions have been proposed[11], these have yet to be implemented in popular server and client libraries. *Pogo* detects, using the Android API, when the active network interface changes and automatically reconnects on the new interface. We have implemented our own end-to-end acknowledgements on top of XMPP to recover from message loss.

Messages that are to be transferred over the XMPP connection are not sent out immediately for two reasons. First, when there is no wireless connectivity, messages should be stored and sent out at a later time when connectivity has been restored. Second, sending small amounts of data over a 2G/3G connection has been shown to be extremely energy inefficient due to the overhead associated with switching between the different energy states of a modem, as we will elaborate upon in the next section. We exploit the fact that data gathering applications generally allow for long latencies in message delivery. Messages are therefore buffered at the device and sent out in batches. Buffered messages are stored in an embedded SQL database to ensure that no messages are lost should a device reboot or run out of battery.

[9] `http://xmpp.org/`
[10] `http://www.igniterealtime.org/projects/openfire/`
[11] i.e. XEP184, XEP198

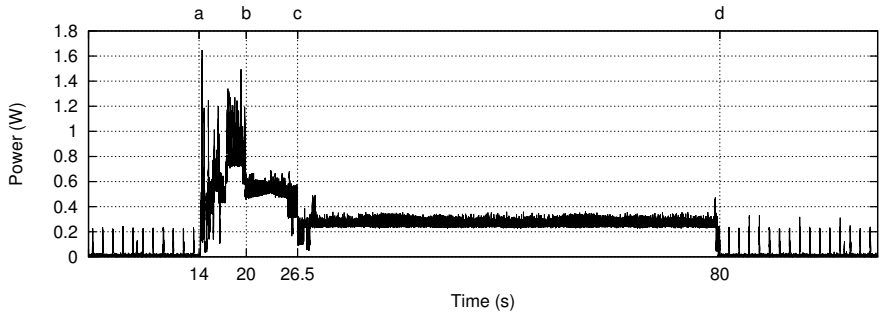

Fig. 3. Tail energy due to 3G transmissions. The 3G ramp-up starts at a. After data transmission has ended (point b), the modem stays in high-energy mode (DCH) for approximately six seconds until c. Finally, there is a long tail where the modem stays in medium-energy mode (FACH) for another 53.5 seconds between c and d. The small spikes before a and after d are due to the duty cycling of the modem. The trace was obtained under the same conditions as described in Section 5.2 on the KPN network.

4.7 Tail Detection

Data transmission over a 2G/3G internet connection is costly due to the tail energy overhead involved [2, 24]. In a nutshell, data transmission triggers the modem to go into a high-power state, where it stays for a considerable amount of time after the transmission itself has ended. Figure 3 shows an energy trace taken from a Samsung Galaxy Nexus smartphone. The event marked a shows the modem being triggered by a transmission (in this case the phone checking for new e-mail). It takes several seconds for the actual transmission to begin as the modem negotiates a private channel with the cell tower, leading to the so-called *ramp-up* time. After the transmission has ended at event b, the modem waits in high energy mode to see if there is further data until point c, after which it goes into a medium-energy mode, where it stays for a further 56 seconds. The time from b to d, 60 seconds in this example, is commonly referred to as the *tail-energy* of a transmission, and periodically transfering small packets of information could easily cause this overhead to dominate the overall energy consumption of the application.

To avoid generating many tails it is possible to either flush the transmit buffer at long intervals (i.e. once per hour), or simply delay transfer until the phone is plugged into the charger. However, there are typically many applications already present on a mobile phone that periodically trigger a 3G tail. Examples are background processes that check for e-mail, instant messaging applications, and turn-based multi-player games. *Pogo* detects when other applications activate the modem, and if it has data to send, takes advantage of this opportunity to push it out before the modem has moved to a lower power state. In this way *Pogo* is able to avoid generating tail energy of its own by synchronizing its transmissions with that of other applications. Since most users typically set

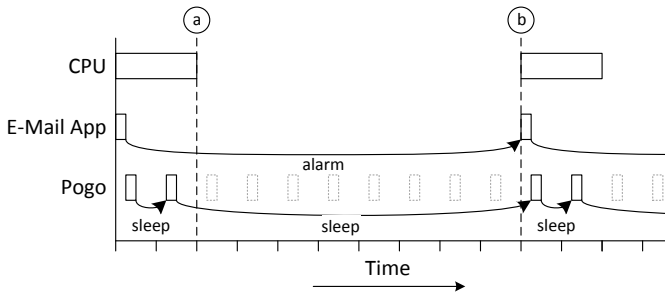

Fig. 4. *Pogo* running alongside an e-mail application that periodically checks for new mail. The horizontal blocks show when the CPU, e-mail app, and *Pogo* are active.

their phones to check for new e-mail every so many minutes, *Pogo* almost never generates its own tail.

The implementation of this scheme requires some special consideration. The general idea is to periodically read the number of bytes received and transmitted on the 2G/3G network interface using the Android API, and fire a transmission event when these numbers change. The exact length of the tail we are trying to detect depends on the mobile carrier, but we typically wish to catch the high-power tail which is measured in seconds.

Periodic sampling at such intervals becomes problematic due to energy overhead incurred when waking up the CPU. As explained in the previous section, Android will put the CPU to sleep when there are no wake locks held by any applications, and can only be woken up explicitly by setting an alarm. When the alarm fires, the CPU will be woken up and start executing pending tasks. The processor will stay awake for typically more than a second before going back to sleep, even if there is nothing for it to do. With a sampling interval measured in seconds the overhead from keeping the CPU awake would be considerable.

We therefore use a side-effect of how Java's `Thread.sleep` method is implemented on Android. When the processor is in sleep mode, the timers that govern the sleeping behavior are also frozen, which means that the thread will only continue to execute after the CPU has been woken up by some other process. We use this to detect when the CPU is woken up by another application, possibly a background service that wants to engage in data transmission.

Figure 4 shows a situation where an e-mail application periodically checks for new mail. The e-mail app uses the alarm functionality of Android to ensure that the CPU is woken up. *Pogo* checks for network activity every second, but uses `Thread.sleep` instead of alarms. At event *(a)*, the CPU goes to sleep because there are no wake locks preventing it from doing so. With the CPU sleeping, the *Pogo* thread is no longer running. At event *(b)*, the alarm set by the e-mail app fires and the CPU is brought out of sleep mode. The *Pogo* thread continues and is eventually unblocked when its timer runs out. It can then detect the network traffic and push out its own data.

Table 2. Code complexity for *Pogo* applications. Size is given in bytes.

Application	File	SLOC	Size
`Localization example`	`scan.js`	41	1,414
	`clustering.js`	155	4,096
	`collect.js`	18	469
	total	**214**	**5,979**
`RogueFinder`	`roguefinder.js`	28	799
	`collect.js`	5	100
	total	**32**	**899**

5 Evaluation

In this section we evaluate *Pogo* in three ways. We first validate the suitability of our programming model for mobile sensing applications in Section 5.1. In Section 5.2 we show, using power traces obtained from a modern Android smartphone, that our mechanism for avoiding tail-energy significantly reduces the energy overhead of *Pogo*. Finally, we present our experience with a real-world experiment in Section 5.3.

5.1 Program Complexity

We implemented the example application described in Section 4.1. Table 2 shows the source lines of code count[12] for the application. The `clustering.js` script is by far the largest, mainly due to the modified DBSCAN clustering algorithm, as well as functionality for calculating the cosine coefficient. Still, the entire application takes up only 214 lines of code.

We also wanted to compare our programming model against related work. Listing 1 shows the RogueFinder application written in AnonyTL, as it appears in [8]. This program sends Wi-Fi access point scans to the server once per minute, but only if the device is within a given geographical location (represented by a polygon). We implemented an equivalent program for *Pogo*, a fragment of which is shown in Listing 2. First, a subscription is created to scan for access points on line 5. This subscription is then immediately released because scanning should only be activated within the designated area (line 9). On line 11, the application subscribes to location updates, and toggles the Wi-Fi subscription based on the device location. Note that the `locationInPolygon` method was omitted for brevity. The total size for this application can be found in Table 2.

The RogueFinder application illustrates the trade-off between DSLs and our JavaScript-based approach. First, we had to implement the `locationInPolygon` function to simulate AnonyTL's `In` construct, as this was not a part of our API. Second, toggling the Wi-Fi scanning sensor depending on the user location required extra work (lines 11-16). Third, a second script running on the collector node (`collect.js`) is required to get the data off the device. Still, we would argue

[12] Empty lines and comments are not counted

Listing 1. The RogueFinder application in AnonyTL.

```
(Task 25043) (Expires 1196728453)
  (Accept (= @carrier 'professor'))
  (Report (location SSIDs) (Every 1 Minute)
    (In location
      (Polygon (Point 1 1) (Point 2 2)
      (Point 3 0))))
```

Listing 2. The RogueFinder application for *Pogo* (fragment).

```
function start()
{
  var polygon = [{ x:1, y:1}, { x:2, y:2 }, { x:3, y:0 }];

  var subscription = subscribe('wifi-scan', function(msg) {
    publish(msg, 'filtered-scans');
  }, { interval : 60 * 1000 });

  subscription.release();

  subscribe('location', function(msg) {
    if (locationInPolygon(msg, polygon))
      subscription.renew();
    else
      subscription.release();
  });
}
```

that this increase in complexity is an acceptable price to pay for the flexibility that *Pogo* offers over application-specific solutions such as AnonyTL.

5.2 Power Consumption

We validate the tail detection mechanism described in Section 4.7 by taking detailed power measurements from a Samsung Galaxy Nexus phone. We set up a single e-mail account and configured it to be checked at 5 minute intervals. We and ran experiments both with- and without *Pogo* running alongside it. In the experiments where *Pogo* was running it was sampling the battery sensor every minute. Because of the synchronization mechanism these values were reported in batches of five whenever the e-mail application checked for updates. We inserted a 0.33Ω shunt between the battery voltage line and sampled the voltage drop over the shunt using a National Instruments NI USB-6009 14-bit ADC. The phone was running stock firmware, Android 4.0 (Ice Cream Sandwitch), with all background processes such as location services disabled.

Table 3. Power consumption with- and without *Pogo* running on a Samsung Galaxy Nexus with e-mail being checked every five minutes. When *Pogo* is running, it reports battery voltage sampled once per minute.

Carrier	Without *Pogo*	With *Pogo*	Increase
KPN	277.59 J	288.76 J	4.09%
T-Mobile	182.05 J	194.3 J	6.73%
Vodafone	205.47 J	218.98 J	6.57%

Table 4. Results of the localization experiment. The size columns show the size in bytes of the raw data set.

User	Scans	Size	Locations	Size	Match	Partial
User 1	25,562	6,278,929	230	89,514	95%	96%
User 2a	11,474	3,082,356	121	48,048	86%	90%
User 2b	6,745	2,139,525	93	44,154	97%	100%
User 3	33,224	9,064,727	1282	437,527	80%	83%
User 4	32,092	12,664,291	274	139,572	92%	97%
User 5	33,549	11,836,962	333	197,433	95%	98%
User 6	34,230	14,426,142	158	77,251	89%	96%
User 7	35,637	9,305,313	703	181,389	96%	98%
User 8	34,395	11,618,974	329	141,634	95%	97%

We obtained one-hour traces with- and without *Pogo* running and compared the energy consumption. Because the length of the 3G tail depends on carrier settings we repeated this experiment with each of the three major mobile carriers in The Netherlands. With each comparison we took the trace without *Pogo* running as the base line and calculated the increase in power consumption as a percentage of that value. The results are shown in Table 3.

The differences between the different carriers are substantial. We observed very long tails on the KPN network (Figure 3 shows such a tail), resulting in a higher total energy consumption than on the other two networks. On the other hand we found the differences in energy consumption on the same network due to *Pogo* to be marginal, with a maximum of 6.57% increase in total consumption on the Vodafone network. This shows that *Pogo* can report data regularly with minimal energy overhead by automatically synchronizing its tranmission with other background processes.

5.3 Experimental Results

We tested *Pogo* by deploying the localization application described in Section 4.1 and let it run for 24 days. Of the 8 participants, 6 were given a Samsung Galaxy Nexus to use as their primary phone. The other two preferred to use their own phone, a Sony Ericsson Xperia X10 mini and a Samsung Nexus S. The latter participant experienced some issues with his phone however and later switched to a Galaxy Nexus, and we denote this user's two sessions as 2a and 2b respectively.

One of the participants did not have mobile Internet and had to rely on Wi-Fi to offload his data periodically (user 7). The application additionally logged all Wi-Fi scan results to SD card, and these raw traces were collected after the experiment as ground truth.

Table 4 shows an overview of the results. In total we collected 246,908 access point scans for a total of 76,7MB of raw data, and found 3,525 user locations[13] for a total of 1.3MB of raw data. In other words, we reduced the total amount of data transferred by 98.3% by making use of on-line clustering as opposed to sending all data back to the collector node. To see what the quality of the data was like, we ran our clustering algorithm over the raw traces and compared the output with what was received at the collector node. We found that there were inconsistencies between the two data sets. First, some clusters were missing at the collector node, or had a later start time. This was due to the clustering algorithm being interrupted half-way through building a cluster, losing its program state. When *Pogo* resumed it would only report the latter half of the dwelling session, hence the difference in cluster start times. This would happen if a phone was rebooted, ran of out battery, or when we uploaded a new version of the script.

Furthermore, we found that for two users we were missing large numbers of clusters, specifically in certain time periods. This was because we had configured *Pogo* to drop messages older than 24 hours if there was no Internet connectivity. We had not anticipated that this would become an issue since all participants had regular Internet access. However, user 2a made a trip abroad and turned off data roaming for cost reasons, resulting in messages being purged after a day. User 3 experienced problems with his 3G Internet access resulting in two days of missing data.

The 'match' column in Table 4 shows the percentage of clusters found in the post-processed data set that exactly matched the ones gathered by the collector node. The 'partial' column shows the percentage of nodes that were matched only partially due to the problems described. We have since added the `freeze` and `thaw` methods to preserve application state across clean application restarts which will help reduce the problem of *Pogo* scripts being interrupted, and improve data quality.

6 Conclusions

The smart phone revolution is rapidly changing the field of mobile data gathering. Modern phones have very capable processing hardware, ubiquitous Internet connectivity, and a range of interesting sensor modalities, which makes them – in principle – an ideal platform for all kinds of information collection tasks. In reality though, experiments with large collections of mobile devices are rare, and only carried out by a handful of experts, due to a string of complicating factors.

In this paper we presented *Pogo*, our proposed middleware for building large-scale mobile phone sensing test beds. *Pogo* takes a pragmatic approach and

[13] Note that these are not unique locations, but rather sessions of a user staying at some place.

gives researchers a subset of the available mobile devices for them to deploy experiments on. These experiments are written in JavaScript, and use a publish-subscribe framework that abstracts away the details of communication between mobile devices and researchers' computers. Users are given fine-grained control over what sensor information they wish to share to protect their privacy

We have demonstrated the feasibility of our implementation with a real-world use case involving eight users and running for 24 days. We argue that the programming model we developed for *Pogo* is easy to use, yet flexible enough to build complex applications. Finally, we have shown, through detailed power measurements, that *Pogo* is capable of offloading its sensor data at a very low energy overhead – as little as 4% – by synchronizing its transmissions with other background processes present on the device.

Pogo is a work in progress. In the future we would like to implement power modelling to estimate the resource consumption of individual scripts. We would also like to automate the assignment process between devices and researchers based on information such as device capabilities and geographical location. Finally, we are planning on contributing *Pogo* to the mobile phone sensing community as an open-source project in the near future.

References

1. Anderson, D.P.: Boinc: A system for public-resource computing and storage. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing. pp. 4–10. GRID '04, IEEE Computer Society, Washington, DC, USA (2004)
2. Balasubramanian, N., Balasubramanian, A., Venkataramani, A.: Energy consumption in mobile phones: a measurement study and implications for network applications. In: 9th ACM SIGCOMM Internet Measurement Conference. pp. 280–293. IMC '09, Chicago, ILL (Nov 2009)
3. Busi, N., Zavattaro, G.: Publish/subscribe vs. shared dataspace coordination infrastructures. is it just a matter of taste? In: WETICE '01 Proceedings of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises. pp. 328 –333 (2001)
4. Bychkovsky, V., Chen, K., Goraczko, M., Hu, H., Hull, B., Miu, A., Shih, E., Zhang, Y., Balakrishnan, H., Madden, S.: The CarTel mobile sensor computing system. In: 4th int. conf. on Embedded Networked Sensor Systems. pp. 383–384. SenSys '06, Boulder, Colorado, USA (Nov 2006)
5. Campbell, A.T., Eisenman, S.B., Lane, N.D., Miluzzo, E., Peterson, R.A.: People-centric urban sensing. In: 2nd int. conference on Wireless Internet. WiCon '06, Boston, MA (Aug 2006)
6. Chu, D., Kansal, A., Liu, J., Zhai, F.: Mobile apps: It's time to move up to CondOS. In: 13th Workshop on Hot Topics in Operating Systems. pp. 1–5. HotOS XIII, Napa, CA (May 2011)
7. Cisco: Cisco visual networking index: Global mobile data traffic forecast update, 2010-2015. `http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white\_paper\_c11-520862.html` (Feb 2011)
8. Cornelius, C., Kapadia, A., Kotz, D., Peebles, D., Shin, M., Triandopoulos, N.: Anonysense: privacy-aware people-centric sensing. In: 6th int. conf. on Mobile Systems, Applications, and Services. pp. 211–224. MobiSys '08 (Jun 2008)

9. Cuervo, E., Gilbert, P., Wu, B., Cox, L.: Crowdlab: An architecture for volunteer mobile testbeds. In: Communication Systems and Networks. pp. 1 –10. COMSNETS, Bangalore, India (Jan 2011)
10. Das, T., Mohan, P., Padmanabhan, V.N., Ramjee, R., Sharma, A.: PRISM: platform for remote sensing using smartphones. In: 8th int. conf. on Mobile Systems, Applications, and Services. pp. 63–76. MobiSys '10, San Francisco, CA (Jun 2010)
11. Eagle, N., Pentland, A.: Reality mining: sensing complex social systems. Personal Ubiquitous Computing 10, 255–268 (Mar 2006)
12. Ester, M., peter Kriegel, H., S, J., Xu, X.: A density-based algorithm for discovering clusters in large spatial databases with noise. pp. 226–231. AAAI Press (1996)
13. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys 35, 114–131 (Jun 2003)
14. Falaki, H., Mahajan, R., Estrin, D.: Systemsens: a tool for monitoring usage in smartphone research deployments. In: Proceedings of the sixth international workshop on MobiArch. pp. 25–30. MobiArch '11, ACM, New York, NY, USA (2011)
15. Froehlich, J., Chen, M.Y., Consolvo, S., Harrison, B., Landay, J.A.: Myexperience: a system for in situ tracing and capturing of user feedback on mobile phones. pp. 57–70. MobiSys '07, ACM, New York, NY, USA (2007)
16. Gelernter, D.: Generative communication in linda. ACM Transactions on Programming Languages and Systems 7, 80–112 (1985)
17. Glater, J.D.: Welcome, freshmen. have an ipod. `http://www.nytimes.com/2008/08/21/technology/21iphone.html?ref=education` (2008)
18. Google geolocation API. `http://code.google.com/p/gears/wiki/GeolocationAPI` (November 2009)
19. Kemp, R., Palmer, N., Kielmann, T., Bal, H.: The smartphone and the cloud: Power to the user. In: MobiCloud 2010. pp. 1–6. Santa Clara, CA (Oct 2010)
20. Krause, A., Horvitz, E., Kansal, A., Zhao, F.: Toward community sensing. In: 7th int conf. on Information Processing in Sensor Networks. pp. 481–492. IPSN '08, St. Louis, Missouri, USA (Apr 2008)
21. Langendoen, K., Baggio, A., Visser, O.: Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In: 14th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS). Rhodes, Greece (Apr 2006)
22. Lee, Y., Iyengar, S.S., Min, C., Ju, Y., Kang, S., Park, T., Lee, J., Rhee, Y., Song, J.: Mobicon: a mobile context-monitoring platform. Commun. ACM 55(3), 54–65 (Mar 2012)
23. Lu, H., Yang, J., Liu, Z., Lane, N.D., Choudhury, T., Campbell, A.T.: The jigsaw continuous sensing engine for mobile phone applications. In: 8th ACM Conference on Embedded Networked Sensor Systems. pp. 71–84. SenSys '10, Zürich, Switzerland (Nov 2010)
24. Qian, F., Wang, Z., Gerber, A., Mao, Z.M., Sen, S., Spatscheck, O.: Characterizing radio resource allocation for 3g networks. In: Proceedings of the 10th annual conference on Internet measurement. pp. 137–150. IMC '10, ACM, New York, NY, USA (2010)
25. Schilit, B., Adams, N., Want, R.: Context-aware computing applications. In: First Workshop on Mobile Computing Systems and Applications. pp. 85–90. Santa Cruz, CA (Dec 1994)
26. Shepard, C., Rahmati, A., Tossell, C., Zhong, L., Kortum, P.: Livelab: measuring wireless networks and smartphone users in the field. SIGMETRICS Perform. Eval. Rev. 38(3), 15–20 (Jan 2011)