Energy Efficient Information Monitoring Applications on Smartphones through Communication Offloading

Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal

VU University Amsterdam,
De Boelelaan 1081A,
Amsterdam, The Netherlands
{rkemp,palmer,kielmann,bal}@cs.vu.nl

Abstract. People increasingly use a wide variety of applications on their smartphones, thereby putting an ever higher burden on their phone's battery. Unfortunately, battery capacity does not keep up with the energy demand of these applications. Various solutions have been proposed to get as much work as possible done with the scarcely available energy, among which offloading heavy weight computation to cloud resources.

In addition to offloading *computation* to cloud resources for computation intensive applications, we propose to also offload *communication* to cloud resources for communication intensive applications. In this paper we show that applications that monitor information on the Internet can offload the majority of their communication to cloud resources, thereby saving a significant amount of energy.

Along with discussing the principle of communication offloading, we detail the design and implementation of our communication offloading component that is part of the Cuckoo Offloading Framework. We evaluate this framework with an application monitoring a subsection of any given website based on image comparison and that communication offloading saves energy on the mobile device.

Keywords: communication, offloading, smartphone, energy efficiency.

1 Introduction

Today, smartphones have been widely accepted as primary personal communication devices. Key to this acceptance are improvements in phone hardware, such as better processors, integration of various sensors and higher quality touchscreens, together with improvements in mobile operating systems and networking technologies. These improvements have made the smartphone a compelling platform for a wide spectrum of applications, ranging from web browsers and games, to navigation and personal health applications, and much more. These applications increase the phone's energy consumption, while battery capacity remains limited and is not expected to grow significantly [14,16].

To address this energy problem, various solutions have been proposed on different abstraction levels, from low level energy efficient hardware components,

(e.g. modern low power smartphone processors) to increasing energy consumption awareness at the user level. Many solutions are orthogonal and can be used simultanuously. Ideally they will minimize the use of the scarcely available energy without reducing the user experience.

In this paper we will focus on energy efficiency solutions on the middleware level. One of the known solutions in this area is to offload intensive energy consuming computations from a smartphone to another computation resource, a technique known as *computation offloading* [12] or *cyber foraging* [3] and suitable for applications containing heavy weight computing, such as object recognition [10]. To ease the creation of these applications and to make intelligent run time decisions about whether or not to offload we created an offloading framework, called Cuckoo [8], which supports computation offloading. Although computation offloading can significantly reduce the energy footprint of compute intensive applications, those applications are not yet frequently used, and therefore the total energy saving is limited.

We therefore identified a class of applications that are, in contrast to the computation offloading applications, frequently used *and* involve significant communication, which is known to be very expensive in terms of energy consumption. This class of applications comprises *information monitoring* applications. These applications are typically continuously or periodically running, either as a home screen widget or as a background service, and monitor some source of information on the Internet. Examples are RSS readers, social network apps, sports score services, weather information widgets, traffic information widgets, and many more.

To find out whether certain information has been changed on the World Wide Web – inherently a *pull*-based architecture – these applications have to repeatedly pull a website to detect changes. Repeatedly pulling information – *polling* – from sources with an unpredictable update behavior has disadvantages. It will either cause unnecessary communication, in the case that information has not changed between two polls, or cause information on the device to be out of date, in case the information has changed, but no new poll has happened yet. Thus, setting the polling rate involves a tradeoff between energy efficiency on the one hand and accuracy on the other hand.

To avoid having this tradeoff, we introduce a new mechanism, *Communication Offloading*, that minimizes energy cost, while it maximizes the accuracy of monitoring applications. With this mechanism, communication intensive polling is offloaded to cloud resources, while communication between cloud and smartphone is done only when necessary with *push* communication.

To create a system that implements Communication Offloading several problems have to be addressed. In many cases, cloud initiated communication to a phone is difficult because of Network Address Translation and firewalls. Furthermore, the use of cloud resources involves additional cost and incurs scalability issues. Finally, having additional components in the cloud introduces end user problems such as vendor lock-in and privacy problems [9].

In this paper we will discuss the principle of Communication Offloading and how we added support for it into the Cuckoo offloading framework. We describe

the design and implementation of our Cloud Based Push Framework, a fundamental component needed for Communication Offloading, and evaluate the framework using a real life example application that monitors images of arbitrary web pages.

The contributions of this paper are:

– We introduce a new mechanism for energy efficient information monitoring applications on smartphones, called Communication Offloading.
– We discuss the requirements for and an implementation of a Communication Offloading Framework, which enables developers to offload communication to the cloud.
– We present a Communication Offloading framework that enables easy implementation and execution of Communication Offloading applications and addresses the problem of maintaining connectivity between cloud and phone.
– We evaluate the Communication Offloading Framework with a real life application that does Image Based Website Monitoring and show that Communication Offloading is more energy efficient than the traditional approach.

This paper is organized as follows. Section 2 discusses the Communication Offloading principle. Section 3 details the background of our work. Next, in Section 4 we describe the requirements, design and implementation of our Communication Offloading Framework. In Section 5 we demonstrate the framework with a real life application and in Section 6 we evaluate the framework with measurements of example scenarios and provide insight in the energy saved by Communication Offloading. In Section 7 we discuss the related work and we give our conclusions in Section 8.

2   Communication Offloading

2.1   Target: Polling Applications

Communication intensive applications contribute significantly to the total energy consumed by applications. Depending on the type of network used for communication and the computation intensity of the associated pre and post processing by the CPU, communication is costs more in terms of mJ per second than heavy weight computation (see Figure 1). For example Google's Nexus One can scale its CPU clock frequency from 245 MHz to 1 GHz, where a higher clock frequency results in more energy consumption. Any combination of communication in combination with computation, however, is at least 11%, but up to 177%, more expensive than running the CPU at its highest frequency. Thus communication intensive applications are an even more interesting target for energy saving measures than computation intensive applications. Reducing the amount of communication and/or the complexity of the associated processing will lead to less energy consumption.

Within the class of communication intensive applications one can identify two subgroups, differing in whether the communication is predictable or not. On the
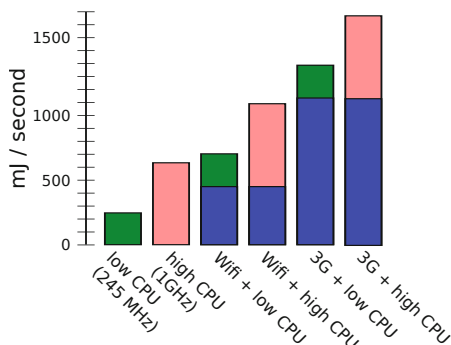
Fig. 1. Average cost of computation and communication on the Nexus One. Even the least energy consuming communication in combination with computation at the lowest clock speed is more costly than heavy-weight computation without communication. Values computed from an estimated fixed voltage of 3.8V and the manufacturer provided power profile which lists operations and the current they draw.

one hand there are the applications with unpredictable communication, such as web browsers, for which it is unknown beforehand when and which page will be requested to be retrieved, because this will be decided at run time by the user of the application.

On the other hand there are applications that do have a predictable behavior, typically monitoring a specific web resource with a fixed interval. These applications can be categorized as *information monitoring* applications (see Figure 2-i). Examples of such information monitoring applications are: weather notification, traffic monitoring, stock market monitoring, etc. and one can easily imagine many more applications. Many of these applications run permanently, for instance as a *home screen widget*.

In the remainder of this paper we will focus on permanently running information monitoring applications, since these applications consume a significant amount of energy and are therefore good candidates for energy saving techniques.

2.2   Pull versus Push

A naive energy reducing measure for information monitoring applications is to reduce the polling rate of the application. This will reduce the number of web requests and therefore the consumed energy. However, reducing the polling rate will also affect the accuracy. Information updates will be discovered later. Thus, using polling there exists a tradeoff between energy usage and accuracy of the information displayed.

Information monitoring applications pull information from web resources and then, when the data is locally available, inspect whether this data contains new information. If so, the application updates its state accordingly.

Note that, if the data on the web resource did not change during the polling interval, the energy spent on retrieving data from the resource does not affect the application's state and is thus effectively wasted. This energy waste is
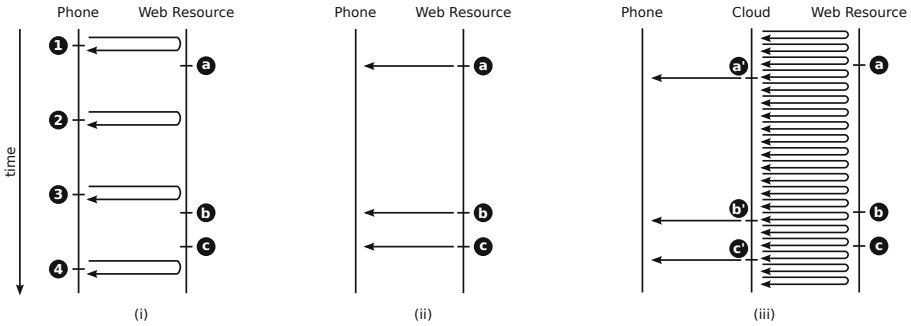
Fig. 2. Examples of Different Phone-Web Interactions. At the points (a), (b) and (c) the web resource updates its information. If the polling mechanism is used (i), updates are received on the phone after some delay – the time between (a) and (2), (c) and (4). Some updates are not received at all, e.g. (b). Using server based push notifications (ii), these situations will not happen. Cloud based push notifications (iii) use an intermediate cloud resource, that can poll the web resource at a much higher frequency and will therefore have a much shorter delay and is less likely to miss updates. Furthermore, the cloud based push mechanism moves energy expensive polling to the cloud.

unavoidable for phone based polling solutions, since in contrast to polling, information updates are irregular and unpredictable.

To prevent this energy waste, the phone ideally should communicate with the server only when data has changed. This can be done using a server based technique called *push notifications*. Then the server informs clients when specific data has changed (see Figure 2-ii). Push notifications are an excellent solution to have energy efficient applications on mobile phones that show web information.

Implementing push notifications, however, requires server code modifications and in many cases the application developer does not have the rights to alter code on the web server.

2.3   Our Proposal: Cloud Based Push

We propose a new alternative mechanism for information monitoring applications that exploits the energy efficiency of push notifications, but does not require any server code to be changed and thus can be used by third party developers.

We propose to add an intermediate cloud layer in between the client application on the phone and the code on the server. This intermediate layer consists of a small application that runs on a cloud resource. This cloud application communicates with the phone using the energy efficient push notification messages (see Figure 2-iii), while it uses polling at a high rate to retrieve updates from the web resource. Then, the energy is spent on the cloud, where energy is relatively cheap and abundant, while accurate information is available on the phone. There is no need to alter the code on the web server, instead a little extra code is put on a cloud resource. We call such an architecture *Cloud Based Push*, to underline the difference with existing server initiated push notification systems.

2.4   Requirements for Cloud Based Push

While individual developers could employ the principle of cloud based push themselves, the complexity of implementing such a system requires additional skills from developers and lengthens the development time in a market where rapid time-to-market is essential. Therefore, we believe that there is a need for a simple cloud based push framework. Furthermore, a single push framework allows multiple applications on a single phone to share the same push notification connection.

A cloud based push framework should consist of:

- a push system that deals with the communication between the phone and the cloud application.
- a component that maintains connectivity between cloud and phone
- a simple programming model where developers can plug in their monitoring code.

Furthermore, we favor code bundling for distributed applications, such that compatibility between the components is ensured and vendor lock-in is prohibited, thus we add a fourth requirement:

- a deployment system so that cloud code available on the phone and bundled with the client application can be deployed on cloud resources.

3   Background

We believe that the current generation of smartphones is ready to run real *smart applications* – applications that use personal, social and sensor information to make smart decisions to enrich and simplify the life of the user. While we already see some of these applications entering the markets today, there are still several problems for developers to create such applications, among which energy usage is a key problem. In order to ease the process of such applications we are building a toolkit called *Interdroid* (see Figure 3).
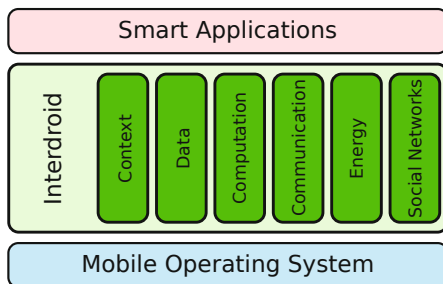
Fig. 3. Abstract overview of the Interdroid project. We strive towards a layer that enables application developers to use social and context information, intensive computing and communication, and data services while keeping energy usage to a minimum.

In this toolkit we offer developers easy ways to access social information and context information [19], to share, version and replicate information and to compute in an energy efficient way on this data. We use offloading to solve part of the energy problem in our toolkit.

3.1   Cuckoo

To encourage developers to use energy efficiency techniques, we strive towards a simple, complete and uniform platform at the middleware layer that we can offer to developers. This will maximize the probability that developers will really employ known energy saving techniques in their application, where time to market is short. In this section we outline our earlier work on offloading in the Cuckoo project [8], one of Interdroid's sub-projects.

Our initial effort in the Cuckoo Offloading Framework is a computation offloading component that can be used by developers to easily create energy efficient applications that contain compute intensive parts. The computation offloading component allows developers to generate both local and remote implementations of compute intensive code. The offloading component can then at run time decide to move heavy weight computation to remote resources, such as clouds.

In the Cuckoo offloading project we believe in a user centric approach, where the user experience should not degrade because of energy saving techniques. Therefore Cuckoo supports both local and remote implementations to have a fallback mechanism for the case when there is no network connectivity to the remote resource *and* to allow for remote implementations that take advantage of being different, for instance being implemented to use parallelism. Furthermore, Cuckoo bundles local and remote code on the phone and installs remote code when needed on the fly on remote resources, so that users are sure that their local code is compatible with the remote code.

4   Cuckoo Communication Offloading

We developed a communication offloading component as part of the Cuckoo Offloading Framework. The communication offloading component consists of:

– a Push Server that runs on cloud resources
– a Push Client library that handles the communication with this server
– a background application (service) that runs on the phone and manages the incoming push notifications.

In this section we will detail the different components, the design decisions and important implementation aspects.

We start with motivating our choice for Android as implementation platform, describe the traditional approach of writing an information monitoring app and explain what changes from a developers perspective.

4.1   Android: Implementation Platform

In order to make a real world implementation of the framework we have selected Android as our underlying implementation platform, because it is widely

adopted, has support for existing Java libraries, such as the Ibis communication libraries [2] that we use for communication between the client and the server, and has been used for the earlier Computation Offloading component of the Cuckoo framework. Furthermore, Android supports multiple application types that could be used for web based information monitoring applications, namely:

- *home screen widgets*: lightweight applications that are permanently visible on one of the home screens.
- *background services*: applications with no UI that run in the background.
- *broadcast receivers*: lightweight applications that start when a particular broadcast message (*Intent*) is sent to the system.
- *activities*: regular applications that have a UI and execute a particular task, like listing email, capturing a photo, etc.

4.2   The Traditional Approach

When one writes a web based monitoring application for the Android platform, one would in general take the following approach. Depending on the type of application that wants to receive updates from the web resource, one creates such an application, be it a home screen widget, a broadcast receiver, or even an activity (although the latter is less likely, because of its short lived nature).

Then, to make sure polling happens in the background, one creates a service, which does the polling. This service will be started by either the widget, the receiver or an activity and then monitor the web resource. Once it finds an update it will send this update via an *Intent* message to the appropriate destination. The widget, broadcast receiver, or activity will receive this message and change its behavior according to the message (e.g. display the new information, alert the user, etc.). A schematic overview of this process is shown in Figure 4. Pseudocode for it is shown in Figure 5.
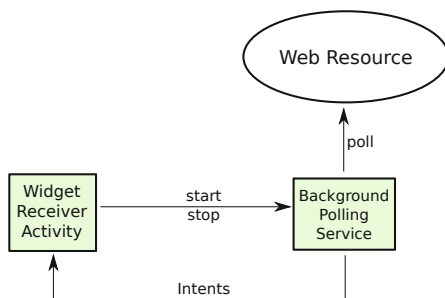
Fig. 4. Schematic overview of a traditional implementation of a monitoring app. An Android component (homescreen widget, broadcast receiver or activity) starts a service on the phone that repeatedly fetches information from a web resource. Once an update is found, an Intent message is sent back to the component.

```
/*********** Service Code ************/
MyService extends Service
// (2), change to MyService extends PushService

while (true) {
    info = poll("http://...");
    if (changed(info)) {
        sendIntent(...); // (3), change to push(...);
    }
    sleep(interval);
}

/************ Widget Code ************/
onCreate() {
    startService(); // (1), change to startRemoteService();
}

onRemove() {
    stopService(); // (1), change to stopRemoteService();
}

onReceiveIntent() {
    updateUI();
}
```

Fig. 5. Pseudocode comparison between traditional and offloading code. The comment numbering corresponds with Figure 6.

4.3   The Offloading Approach

To make it as simple as possible for developers to use communication offloading, we retained the traditional approach as much as possible. Only a few parts in the source code have to be changed (see comments (1) to (3) in Figure 5). Furthermore the developer should add the Cuckoo libraries to the project.

Figure 6 shows the schematic overview of a web polling application that is based on communication offloading. It differs on four points from the traditional approach:

- (1) *Starting and Stopping*: Whereas in the traditional approach components would start and stop a local service, they now have to start and stop remote services.
- (2) *Remote Service*: Whereas the polling service was originally implemented to be executed locally, it now has to be implemented to be executed remotely. Although local code is compiled with the Android libraries and the remote code is compiled against the standard Java libraries, polling code typically does not involve Android specific code. Typically, the original local code and the new remote code are largely the same.
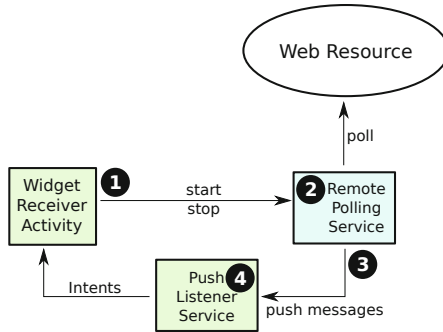
Fig. 6. Schematic overview of an implementation of a monitoring app using communication offloading. The same Android component as in Figure 4 starts and stops the polling, however, the polling now runs in the cloud. Once an update is detected in the cloud, the push listener service on the phone is notified.

- (3) *Informing the Source*: Traditionally, an Intent was sent to the source upon an update. With the offloading approach a message is sent over the network.
- (4) *Transforming the Push Message*: Once the push message is received by the Cuckoo provided listener service, the push message is translated into an Intent, which then as in the traditional approach is received by the corresponding Widget, Broadcast Receiver or Activity.

Our framework provides libraries with base classes that can be extended to implement remote polling services. The framework also provides a server application that can host the polling services, and both a push listener service application and a resource manager application that run on the phone.

4.4   The Push Server

The Push Server is the component of the framework that runs in the cloud and will poll the web resources. The Push Server has to be installed and started on a cloud resource either by a user or by the application provider, or a third party such as a network provider.

The requirements for the cloud resource to run a Push Server are minimal, since the Push Server is a regular Java application and can therefore run on any resource that has a Java Virtual Machine (JVM) installed. Furthermore, the resource needs to be permanently connected to the Internet.

Multiple resources can be bound to a phone, and multiple phones can be bound to a single resource. This allows users to use their own resources, such as home servers, together with resources provided by application providers.

Once a Push Server is up and running it can execute polling services. Such a service is the polling part of a phone application and will be implemented by the application developer and bundled with a web based monitoring application

as a plugin for the Push Server. This code is thus available on the phone, while it needs to be executed on the Push Server. However, this code can be installed on the fly on the Push Server through Java's dynamic class loading mechanisms. Once installed and started, it will execute the polling code and each time an update of the particular web resource is detected, a push message is sent to the phone.

The Push Server maintains a bookkeeping of which devices are using which service and what their actual address is. If a service wants to send a push message to a phone, it will inform its hosting Push Server, which will take care of the delivery of the message. Messages might be not deliverable because the phone is temporarily unreachable, in such a case the Push Server informs the polling service, which can for instance use an exponential backoff algorithm for resending.

Scalability. Depending on who provides the resources that are used for communication offloading, the server might experience scalability issues. When the load on a Push Server becomes too high, one can use elastic computing such as offered by Amazons EC2 [1] and start a new Push Server instance in the cloud. To further improve scalability – in particular down-scaling – it is necessary to be able to transparantly migrate monitoring code from one Push Server to another, so that the number of instances and thus the cost can be kept at a minimum. At the moment Cuckoo does not support migration of monitoring code between Push Servers.

4.5   The Push Listener Service

Up to now we have seen that the phone application needs to be modified slightly to start a remote service running on a Push Server. Once an update is detected the service on the Push Server sends a push message back to the phone. In this section we will outline what happens when such a message arrives at the phone.

Our communication offloading framework provides a Push Listener Service, which runs in the background and waits for push messages to arrive. When such a message comes in, the Push Listener Service analyzes the message and transforms it into an Intent. Once this Intent is broadcast and subsequently received by the application, the application can handle the update. Typically, this will involve updating the UI or alarming the user through sound or vibration.

The Push Listener Service can temporarily be turned off by the user, because users may want to temporarily sacrifice the possibility to receive push messages to save energy.

4.6   Maintaining Connectivity

There is a distinct problem for cloud servers when it comes to sending push messages to mobile phones. First of all, phones inherently are mobile devices, they switch from network to network thereby often changing network address.

In order to send push messages a cloud server has to know the current address of a phone.

Even when this address is known, it is likely that cloud servers cannot reach a mobile phone, because it has limited inbound connectivity due to Network Address Translation (NAT), which is widely used for both Wi-Fi and 3G/4G access points. Access points that use NAT will hand out private IP addresses to connected devices. These addresses are only reachable in the local network and cannot be used by cloud resources outside this private local network to set up connections.

Although cloud resources cannot reach phones, phones can reach cloud resources. A solution to the aforementioned problems is to maintain an open bidirectional connection that is set up by the phone. To prevent such a connection from being closed due to protocol time-outs, keep-alive messages keep the connection active. In our framework we use the SmartSockets library [13] that provides a socket-like library that operates on top of an internal overlay network, which automatically keeps connections active.

Although we can keep connections active with keep-alive messages, eventually a phone switches to another network, and then the active connection is inevitably closed. Connection re-establishment therefore is an essential part of maintaining connectivity. In our implementation the Push Listener Service registers with the mobile OS to get notified of each network change and establishes a new bidirectional connection if the previous one is closed.

With the combination of keeping open connections using SmartSockets and reconnecting upon network changes, the cloud resource always has a means to send messages to the phone as long as the phone is connected to the Internet. The cost of maintaining an open connection through sending the keep-alive messages is very small for Wi-Fi compared to having an idle connection (see Table 1), for 3G it is significantly costlier, but still acceptable since it is a one-time cost and not a per app cost.

**Table 1.** battery cost of maintaining connectivity

	idle	open connection	overhead
Wi-Fi	2.18 %/hr	2.36 %/hr	0.18 %/hr
3G	3.03 %/hr	4.61 %/hr	1.58 %/hr

5   Image Based Website Monitoring

In order to demonstrate the Cuckoo Offloading Framework we made a compelling web based polling application. This application is called *Web Page Widget* and can be used to have a live view of a specific area of a website in a homescreen widget. Figure 7 shows screenshots of this widget.

The first screenshot shows one instance of the widget. This widget shows the output of monitoring a Dutch news website with live traffic information, weather information, the gas price and the stock exchange index value[1]. The

---

[1] `http://www.nu.nl`
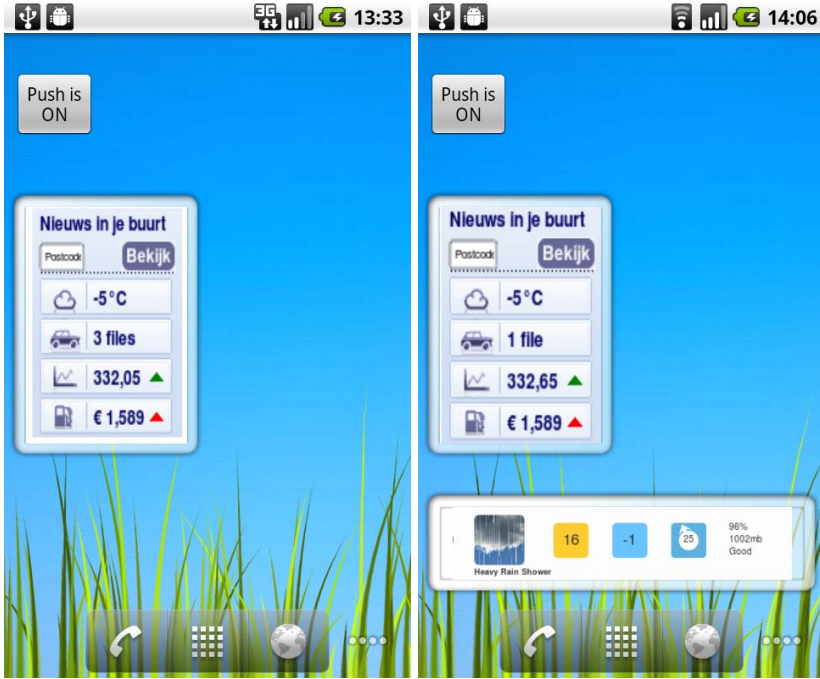
Fig. 7. Two screenshots of the Web Page Widget application. The first screenshot shows a single instance of the widget, the second shows the same widget half an hour later with updated values. Furthermore, a new instance of the same widget monitoring a different web resource is added. The screenshots also show the user control to turn delivery of push messages on or off.

second screenshot is taken about half an hour later and shows two instances of the widget. One of these is the same widget as in the first screenshot, but now with updated values. The other instance monitors weather information from the BBC weather webpage[2].

Each widget must be configured before it can be placed on a homescreen. The configuration consists of three steps and is visualized in Figure 8:

- *Select URL*: Select the URL of the webpage that is going to be monitored.
- *Select Area*: Select a rectangle around the area of interest on the webpage.
- *Select Scaling*: Select how the website area will be scaled in the widget area.

Once configured, the widget is placed on the homescreen and updates the part of the webpage in near real time whenever the contents of the website in that specific area change.

5.1   Implementation

While for some monitoring applications it is arguable to have server based push solution, that is the web site owner providing push support to mobile phones, it is

---

[2] `http://news.bbc.co.uk/weather/forecast/101`

Fig. 8. Configuration of a Web Page Widget. First a URL of a to be monitored website is entered. Then this webpage gets rendered at the remote resource an image is sent back to the phone. The user now selects the area of interest. Finally, the user can specify how the image will be mapped onto the area on the screen.

clear that this application cannot be implemented with server based push. There is no single authority of all websites, so polling is the only way to implement such a monitoring application.

When we want to monitor websites we have to repeatedly execute HTTP requests. The data we get back is in HTML format and we can use text comparison to inspect whether it is the same as the previous value. Since we are interested in only a subpart of the webpage and there is no clear mapping to a rectangle on the screen and the HTML tags belonging to this rectangle, we render the webpage and then use pixel by pixel image comparison to detect changes. If a change is detected the widget is updated with a new image of the area of interest.

5.2   Traditional versus Offloading

When we implement this application in a traditional way, that is using polling and rendering on the phone itself, it will either drain the battery very rapidly or, when a low polling rate is used, have out of date information.

An offloading approach on the other hand will do polling, rendering and image comparison on the cloud resource and will only send the resulting image when it has changed. The size of the resulting image is significantly lower than the size of the HTML, images and scripts that are sent as a result of HTTP request, because it only covers the area of interest (AoI). Table 2 shows that for the two example widgets from the screenshots the images of the AoI are about 160 times (nu.nl) and 3000 times (bbc) smaller than the size of the full webpage. This alone will already save much of traffic to the phone. For instance, if the

AoI on the nu.nl webpage will change on average every 5 minutes, an offloading implementation can run for 12.5 hours and use an equal amount of data traffic as a single polling operation of a traditional implementation of this widget. Note that not only for this application, but for many monitoring applications users are interested in just a subsection of the overall information, however web resources do not offer interfaces to query for updates of subsections.

6   Measurements

Measuring energy consumption of mobile phones is a complicated task. First of all, there are many factors that influence the energy consumption. Some of these factors cannot easily be controlled, for instance the interference on the wireless network. Furthermore, measuring how much energy is consumed in a given period requires either special hardware or fine grained information from the hardware. Third, the resulting information is only valid for a specific phone on a specific location on a specific network. Running the exact same experiment from a different phone might give other results and going to another location (for instance closer to a cell tower) might also influence the experiment.

Although measuring energy consumption is a difficult task and measurements should be interpreted with care, we have executed several experiments to get insight in the energy cost of several operations related to communication offloading.

The phone we used for our experiments is a Nexus One, running on Android 2.2 with a 1.0 GHz CPU and a Li-Ion 1400 mAh battery. In our Wi-Fi experiments we used a 801.11n network, in the 3G experiments we used the Vodafone network in The Netherlands which uses HSDPA with a bandwidth up to 3.6 Mbps.

For each measurement we use the Android BatteryManager API to query the battery level, which runs from 100% (full) to 0% (empty). We execute our to be measured operation until the battery level has dropped by 5% and measure how long that takes.

In order to validate this methodology we have run a reference measurement to verify that the battery level as returned by the Android BatteryManager API indeed shows linear regression in time and is not merely a percentage of the maximum voltage of the battery, which is known to be non-linear. We found that the values reported by the BatteryManager indeed result in a linear correlation between energy use over time and the battery level.

6.1   Polling vs. Other Activities

Figure 9 shows the results of energy usage measurements of several common operations on smartphones. We found that video capture (720x480, H.264) is the most energy consuming operation, followed by browsing, which intensively uses the 3G radio, and navigation, which uses the GPS, the 3G connection and Bluetooth to stream the audio. Computationally intensive operations, such as playing a 3D racing game[3] also consume much energy, giving a total battery time of about 4 hours and 20 minutes for a user that just plays games.

---

[3] We used Raging Thunder Lite 2 from PolarBit.
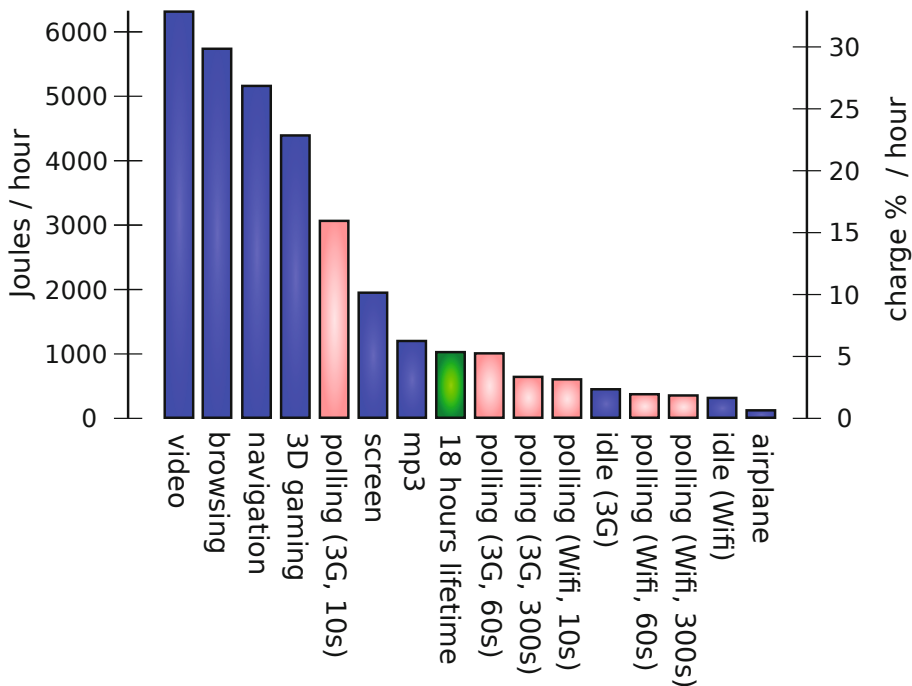
Fig. 9. Energy usage of various common smartphone operations. This graph shows the drop in the battery charge level and the Joules consumed for a particular operation in one hour.

Table 2. web page size vs. image size

source	web page size	AoI size	reduction
nu.nl	1.4 MB	9 kB	159x
bbc	0.9 MB	0.3 kB	3072x

When we look at the energy cost of polling, we see that using the 3G network in our case consumes much more energy than using the Wi-Fi network. Note that various network properties, such as distance to the access point, interference, etc. influence the energy cost. We also see that, like we expected, the energy cost of polling is reduced when the polling interval is reduced.

Although the energy cost of polling per hour is not that high compared to the other tasks, polling happens continuously, while other activities, such as video capture, browsing, navigation, mp3 playback happen only for a limited time. Furthermore, we measured polling for just a single application. If multiple applications are polling concurrently, the energy consumption will increase even more. If the compound number of polls from the various applications is one poll every 10 seconds and we use 3G, the phone will not last for longer than 6 hours and 15 minutes on a full battery. This is well below the lowest accepted operation time of about 18 hours, where a phone is charged every night. Though

Wi-Fi gives better results, one cannot expect that a smartphone user will only use Wi-Fi networks during an entire day.

6.2   Polling vs. Pushing

The amount of energy saved by switching from polling to cloud based push for a given application depends on the characteristics of that application. In particular the following variables influence the energy consumption:

- Average Update Rate (UR): The average time between two updates on the web page.
- Polling Rate (PR): The rate at which the polling application checks for updates.
- Data Reduction Factor: How many times bigger the data for a single poll request is compared to a push message.
- Processing Reduction Factor: How much more processing has to be done for a poll request compared to an incoming push message.
- Accepted Update Latency: The latency with which an update arrives on the phone.

We performed experiments to compare the cost of polling to cloud based push using our example Web Page Widget application monitoring the AoI on the nu.nl web page as shown in Figure 8.

For this application, the data reduction factor is 159 times (see Table 2). Processing using push notifications takes 0.072 seconds, which is updating an

Table 3. Polling vs. Pushing. This table shows how the important properties of monitoring cost with varying polling rates (PR) as a multiple of the update rate (UR) and for push. The values are based on a single widget running for 9h22m monitoring nu.nl. Wi-Fi (W) bandwidth is 623 kB/s, 3G bandwidth is 100 kB/s. Energy consumption is calculated based on the power profile. The lower the polling rate, the lower the energy cost, but also the more updates missed and the longer of a delay until updates are propagated to the phone. The push energy reduction factors include the one-time cost for the open connection (conn), the values between parenthesis are the reduction factors for each subsequent widget. For instance, a single widget with push uses 1.5 times less energy on 3G compared to polling with an PR of .57UR. A second push widget will reuse the existing open connection and therefore the additional cost the energy reduction is 241 times compared to an additional polling widget.

PR	2UR	1UR	0.57UR	0.1UR	push	conn
missed updates	69.3 %	52.9%	42.1%	18.6%	0.0%	-
update latency	138s	103s	60s	12s	0s	-
CPU time/hr	22s	44s	77s	443s	1s	-
Data size/hr	10.3 MB	20.6 MB	36.1 MB	206.5 MB	133 kB	-
batt. %/hr (W)	.11%	.23%	.40%	2.27%	.004%	0.18%
push energy reduction (W)	0.6x (28x)	1.3x (57x)	2.2x (100x)	12.3x (568x)	-	-
batt. %/hr (3G)	.69%	1.38%	2.41%	13.78%	0.01%	1.58%
push energy reduction (3G)	0.4x (69x)	0.9x (138x)	1.5x (241x)	8.7x (1378x)	-	-

image in a widget. The polling implementation has to do significantly more work, since it has to render the full web page and then extract the subimage, which takes in total approximately 3 seconds. The processing reduction factor therefore is 41 times.

We collected an update trace for the AoI on the nu.nl web page. The total trace time is 9 hours and 22 minutes, in which we detected a total of 140 updates, giving an average UR of 4 min 4 sec.

In the experiments we varied the PR of the polling version, which influences the accuracy and the energy consumption (see Table 3). If we apply a PR that is equal to the UR, we only see 66 out of the 140 updates (47%) and the average delay between the update and its appearance on the phone is 1:43 minutes, the energy consumed with polling is 0.9 (3G) to 1.3 (Wi-Fi) times more than with push.

Increasing the PR will decrease the average delay and the number of missed updates at the cost of spending more energy. If our accepted update latency is 1 minute, we have to set our polling rate to 0.57 times the UR. Using this polling rate, we will only see 58% of all updates and energy consumption goes further up to 1.5 (3G) to 2.2 (Wi-Fi) times more than push. When we also want to increase the number of updates we have to increase the polling rate even more. Polling 10 times during the average update interval results in only 18.6% of the updates missed and gives an acceptable age of only 12 seconds when an update arrives on the phone, but energy consumption is dramatic, especially when using 3G. In a single hour, about 14% of the battery will be spent on monitoring this web page, giving a maximum and unacceptable operation time of only 7 hours on a full battery. If we would have more widgets monitoring other web pages the energy consumption increases accordingly and operation time decreases even more.

Monitoring based on cloud based push however, has a much smaller energy footprint. Most of the energy consumed is spent on maintaining the connectivity. This cost however will not increase when multiple widgets are used on a single phone. The cost of running a single instance of a web page widget will only use 0.01% (3G) or even 0.004% (Wi-Fi) of the battery for a full hour. Even if one runs tens of web page widgets concurrently, the operation time of a full battery will only be limited by the small cost of a single open connection, while not missing any update and getting the updates instantly.

7 Related Work

Computation offloading is a well known technique in the field of distributed computing. Already in the era of dumb terminals, resource constrained devices used help from more powerful machines to execute certain tasks. With the introduction of resource constrained mobile devices, computation offloading has been used to allow compute intensive, memory intensive and energy intensive applications to run on mobile devices in such a way that the expensive part of the operation gets executed on remote resources. Before the introduction of cloud computing this technique was known as *cyber foraging* [17], where resource constrained mobile devices used nearby powerful machines – *surrogates* – to offload

computation to. Later work, such as *CloneCloud* [5], uses cloud resources to support computation offloading. In [11] Kumar et al. discuss whether computation offloading can save energy for mobile devices. Some of the existing offloading systems rely on annotations by the developers, while others do automatic partitioning [7,6]. None of these systems support *communication* offloading and allow remote implementations to send push messages to the mobile device.

In this paper we introduce a communication offloading framework. Recently, we have seen a commercial company offering a push based app for Twitter updates, called TweetHook [18], where they run an intermediate layer application in the cloud to query the Twitter API. Google has also started a project called *PubSubHubbub* [15], which uses intermediate applications (hubs) to periodically query publishers of content. Once updates are found all subscribers are informed. PubSubHubbub requires subscribers to be at a fixed location and to be able to receive HTTP Post messages and is therefore not very suitable for communication offloading from mobile devices. PubSubHubbub hubs, however, are excellent sources of information where a communication offloading app can retrieve information from.

Whereas in this paper we use the Cuckoo Push framework, Google also offers a push service called Cloud To Device Messaging (C2DM) [4]. In our future work we will integrate Cuckoo and C2DM, and compare this to a pure Cuckoo implementation.

8   Conclusions

In this paper we proposed a new offloading technique to reduce the energy usage of information monitoring applications on smartphones. We offload expensive polling of web based resources to a cloud resource, that in turn sends push messages to the smartphone only when the information has changed. We have built an extension to the Cuckoo offloading framework to support this *communication offloading* and drastically simplify the process of developing such an application.

We evaluated the technique and the framework with an example application that does image based webpage monitoring and we showed that offloading polling to the cloud can have significant impact on the operation time of a single battery charge.

References

1. Amazon EC2, `http://aws.amazon.com/ec2/`
2. Bal, H.E., Maassen, J., van Nieuwpoort, R.V., Drost, N., Kemp, R., Palmer, N., Wrzesinska, G., Kielmann, T., Seinstra, F., Jacobs, C.: Real-World Distributed Computing with Ibis. IEEE Computer 43, 54–62 (2010)
3. Balan, R., Flinn, J., Satyanarayanan, M., Sinnamohideen, S., Yang, H.-I.: The case for cyber foraging. In: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop, EW 10, pp. 87–92 (2002)
4. Android Cloud to Device Messaging, `http://code.google.com/android/c2dm/`

5. Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: Proceedings of the Sixth Conference on Computer Systems, EuroSys 2011, pp. 301–314 (2011)
6. Cuervo, E., Balasubramanian, A., Cho, D.-K., Wolman, A., Saroiu, S., Chandra, R., Bahl, P.: Maui: making smartphones last longer with code offload. In: Proc. of the 8th Int'l Conference on Mobile Systems, Applications, and Services, MobiSys 2010, pp. 49–62 (2010)
7. Giurgiu, I., Riva, O., Juric, D., Krivulev, I., Alonso, G.: Calling the Cloud: Enabling Mobile Phones as Interfaces to Cloud Applications. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 83–102. Springer, Heidelberg (2009)
8. Kemp, R., Palmer, N., Kielmann, T., Bal, H.: Cuckoo: a Computation Offloading Framework for Smartphones. In: MobiCASE 2010: Proc. of The 2nd International Conference on Mobile Computing, Applications, and Services (2010)
9. Kemp, R., Palmer, N., Kielmann, T., Bal, H.: The Smartphone and the Cloud: Power to the User. In: MobiCloud 2010: Proceedings of the First International Workshop on Mobile Computing and Clouds (2010)
10. Kemp, R., Palmer, N., Kielmann, T., Seinstra, F., Drost, N., Maassen, J., Bal, H.: eyeDentify: Multimedia Cyber Foraging from a Smartphone. In: International Symposium on Multimedia, vol. 11, pp. 392–399 (2009)
11. Kumar, K., Lu, Y.H.: Cloud computing for mobile users: can offloading computation save energy? IEEE Computer 43(4), 51–56 (2010)
12. Li, Z., Wang, C., Xu, R.: Computation offloading to save energy on handheld devices: a partition scheme. In: Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2001, pp. 238–246 (2001)
13. Maassen, J., Bal, H.: Smartsockets: solving the connectivity problems in grid computing. In: Proceedings of the 16th International Symposium on High Performance Distributed Computing, pp. 1–10. ACM (2007)
14. Palacín, M.: Recent advances in rechargeable battery materials: a chemists perspective. Chemical Society Reviews 38(9), 2565–2575 (2009)
15. PubSubHubbub, http://code.google.com/p/pubsubhubbub/
16. Robinson, S.: Cellphone Energy Gap: Desperately Seeking Solutions, Tech report, Strategy Analytics (2009)
17. Satyanarayanan, M.: Pervasive computing: vision and challenges. IEEE Personal Communications 8(4), 10–17 (2001)
18. TweetHook, https://tweethook.com/
19. van Wissen, B., Palmer, N., Kemp, R., Kielmann, T., Bal, H.: ContextDroid: an Expression-Based Context Framework for Android. In: International Workshop on Sensing for App Phones, PhoneSense (2010)