Faculty of Computing Science,
Business Administration, Economics,
and Law

Department of Computer Science

Cloud computing for mobile devices: Reducing energy consumption

Master's Thesis

Submitted by

Veronika Strokova

Supervisor:

Prof. Dr. Andreas Winter

Oldenburg, March, 2014

Abstract

Being powered by the batteries that are limited in their capacity is one of the main restrictions of the mobile devices. On the other hand, further enhancement of their characteristics and mobile Internet mounting speed incite the growth of user's demands. We want the most sophisticated applications to work rapidly and perfectly. Thus, we need a solution to improve the mobile devices' energy consumption.

This master's thesis presents the approach which is able to decrease the power consumption on mobile gadgets. The core idea lies in porting the parts of the application's functionality to the remote server in order to relieve the CPU load. Consequently, we are going to derive the benefit from cloud computing. For this purpose, the "heavy-loaded" code blocks are extracted from mobile apps and built into the server-side applications. Both mobile and "remote" versions of the apps are run, their power consumption is measured and compared. It is expected that energy spent on client-server communication is less than power needed to execute the task on a phone or tablet. Moreover, the performance change is estimated.

The experiments are conducted on three Android applications. The paper provides a technique to move the functionality to the cloud and to organize the communication between a client and a server. It is explained when and how it is required to adjust the approach to overcome the challenges that may occur. In addition, the restrictions and recommendations are denoted and performance changes are outlined. The applications are tested against various data volumes, Internet connection speed and use different ways to transfer data. Energy consumption for each case is measured and the results are presented.

Content

1. Introduction

The sphere of mobile devices is probably the most dynamic in the IT world today [1]. It is hard to imagine our everyday life without the technological tools on hand [2]. They are used not only for our private needs, but also in business, education, healthcare, sport and many other spheres. But the fact is, that the IT resources are confined [3] and it is essential to find the means to extend, customize them and bring them to a new level of energy consumption.

This thesis addresses fast mobile battery discharging [4]. High loading due to the complicated calculations and rich functionality restricts a user in their operations. And it is not the useless battery drain. Usually, we deal with well-written efficient code that we cannot get rid of [5], but that, nevertheless, consumes a lot of power. Thus, we need a solution that will be able to reduce the utilization of the device resources.

The problem of high power consumption is relevant to modern mobile devices. Improvement of characteristics and capacities and mobile Internet mounting speed incite the growth of user's demands. We want the most sophisticated applications to work rapidly and perfectly [6]. At the same time, we are unable to increase the battery capacity [7]. Consequently, we are forced to charge our phones and tablets once or twice a day or refrain from enjoying the favorite apps. Android AppStorm – a web site about Android applications development – conducted the poll with a question "How often do you charge your phone?" [8]. The results are on Figure 1.

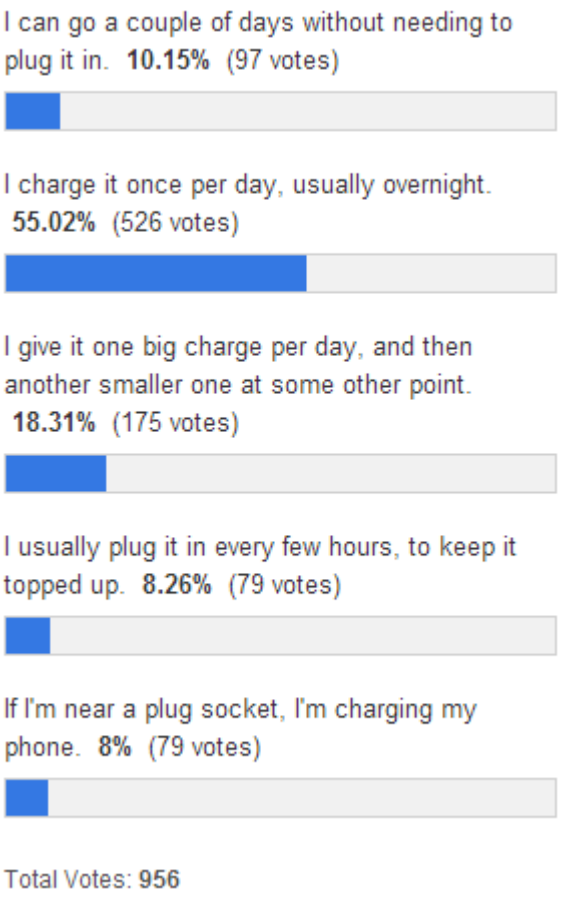

Figure 1. Frequency of charging the Android mobile devices (according to the android.appstorm.net's poll).

The applications with rich functionality contain "heavy-loaded" blocks of code that are responsible for the pivotal functions, but at the same time, take most of the energy consumed by the app. They may include some complicated objects transformation, calculations or data processing. One approach to deal with them is to find this code,

extract it and move to the cloud. In such a way, when a user runs an application, a part of it is executed on a remote server, providing the occurrence of the Internet connection. This approach is elaborated in this work. It is expected, that *the energy needed for data transmission from device to a server and receiving back a result can be less than the energy consumed for executing the same functionality at a device*. This is the first hypothesis of the work.

Another hypothesis is that in many more or less complicated applications there exist some *"heavy-loaded" blocks which can be extracted* without hurting an app and executed remotely leading to more energy efficiency.

The objective of this work is to define the value of delegating parts of functionality from mobile devices to the cloud. Will it improve the energy consumption?

To complete the goal, it is needed to solve a number of tasks:

- First of all, it is required to find the code that is suited for passing to the cloud, extract it without changing or crashing business logic of the application and transfer this functionality to the server. At this point, it is important to determine the most resource consuming blocks of code as we are not going to run the whole application in the cloud. A technique to define such code is not produced in this work, but in our experiments, we handle the methods which contain cycles or a lot of mathematic calculations which significantly load the CPU that, in its term, dominates the power consumption along with the screen [9]. After conducting the experiments, the deductions about how these operations are worth migrating to the cloud will be made. Everything is done on the basis of some case studies in order to verify hypotheses of the work.
- Secondly, it is required to measure the power needed to execute a function on a device and the energy consumed for data transmission to a server and getting back the result [10].
- Finally, these two values are compared and a conclusion about the efficiency of the method is made.

The developer tools used are described in the "Tools" chapter later.

In this work, two state-of-the-art concepts are combined: mobile technologies and cloud computing [11]. None of them is brand new but today they are on the peak of popularity [12] and the approach to use this combination for achieving more energy efficiency is novel.

To fulfil the tasks, we need Android [13] mobile applications with open source code. It should be the Java applications with some “heavy” functionality which can be harmlessly extracted. If we consider some resource consuming calculations, the following apps seem to be suitable:

- route building in navigation systems;
- building panoramic pictures;
- advanced engineering calculations;
- optical character recognition;
- graphical editors;
- specific systems (professional ones);
- speech recognition;
- music sequencers.

By now there are some examples of software delegating a part of their functionality to a remote server. For instance, the iOS Siri [14] application for interpreting and recognizing speech both connects to the cloud and evaluates data locally [15] [16] [17].

If the idea succeeds, we will be able to manage the problems by reducing power consumption, therefore providing more and more complicated and highly functional applications with longer battery live phase. In addition, it might be useful to evaluate *the variation in performance* if there is any. Performance might decrease as we cannot guarantee that a result from the server will be received rapidly. But even if the performance goes down, it can be acceptable and preferable for a user at the cost of battery live time increase. So, the performance trend is the third question of this research.

Globally, it will be possible to enjoy the favorite apps faster and longer, charging the battery rarely and without realizing any technical difference from what it was before. On the other hand, the software proprietors attain more competitive and popular products [18]. Moreover, it is environmentally friendly as less energy is consumed and less heat is produced that of significant concern.

Hence, the objective of this work is to conduct a number of experiments to verify three hypotheses:

- the energy needed for data transmission from device to a server and receiving back a result can be less than the energy consumed for executing the same functionality at a device;
- there can exist some “heavy-loaded” code blocks which can be extracted without hurting an application and executed remotely;

- performance can change to the worse as well as to the better.

The master’s thesis is divided into several chapters. First of all, it is necessary to find a suitable application, determine, extract and delegate functionality to the cloud server and measure energy needed to transfer data from a mobile device to the cloud. These experiments will serve as a basis for defining power-hungry features. The power consumption of the cloud is not measured. Secondly, it is necessary to measure power and pay attention to CPU resources required to run the applications on a mobile device considering the same application. After that, the comparisons are made and the approach is generalized. Finally, the conclusions about value and worthiness of the approach are derived.

2. Tools

For this work, the following tools are used: Java [19] as a programming language and application platform, the Android SDK, Little Eye for measuring power consumption, OpenShift [20] - the open cloud application platform by Red Hat as a server for the applications, Vert.x [21] – an application platform for the JVM – to facilitate the implementation of HTTP-communication between a client and a server, Gson [22] Java library for processing JSON used for transferring data between a client and a server. All of them and some alternatives are described in this section.

2.1. Android

Android is the most popular mobile OS in the world [13]. Android community provides the open source platform for free distribution, modification and development.

The Android system was developed considering the mobile devices' restrictions, including that they are battery powered and battery performance is not likely to get better [23] [24]. The Dalvik VM contains a Just-In-Time (JIT) compiler. “The JIT is a software component which takes application code, analyzes it, and actively translates it into a form that runs faster, doing so while the application continues to run. ... Code that is written to be rate-limited can get its work done using less time and less of the CPU (using less battery)” [25].

Android mainly uses Java as a platform for application development. The code and other data are compiled with Android SDK (software development kit) [26] and packaged in an .apk archive file. .apk files are directly installed on Android devices. The code of each application runs in isolation, that is, an application is a user as in Linux systems (Android actually bases on Linux kernel), it is granted its unique user ID when started, and it has no access to the data of other applications or system. However, it is possible to give such permission [27].

It is also possible to use native-code languages such as C managing them with Android NDK (native development kit). Sometimes, it can be useful to reuse already existing C-libraries.

By sight, a user can distinguish three layers of Android application. Typically, a combination of top level and detail/edit views is used. Top level presents core features and purpose of the application. The detail/edit view allows a user to receive or post data. If the navigation is supposed to be deep, category view is layed between top level and detail/edit views [28].

Programmatically, an Android application can contain four types of components: activities, services, content providers and broadcast receivers [27].

An activity can be referred to as a single screen with UI. Each activity, although working together, is independent of the others. Only

one activity can be run by the app at once. An activity is implemented by extending the class Activity.

Activity lifecycle is managed with the help of callback functions. One of the most important is onCreate() which is called by the system when activity is created. This function must be implemented. The user interface is setup here with setContentView(). Other callback functions regarding lifecycle are onStart(), onResume(), onPause(), onStop(), onDestroy(). Each one is called in accordance with the new application state [29].

A service runs in the background and executes long-running operations or supports remote processes. It is started or accessed by other components. A user interface is not provided by a service. A service is implemented by extending the class Service.

A content provider manages application data. There are some options to store data: the file system, a SQLite database, the web, etc. Depending on the content provider settings, other applications can read or modify the data. In addition to this, a developer can use content provider to read and write data which will not be shared with other apps. A content provider is implemented by extending the class ContentProvider.

A broadcast receiver listens to system-wide broadcast announcements. It does not provide a user interface, but can generate notifications to inform a user. A receiver often serves as a tool for other components detecting that an event has happened. A broadcast receiver is implemented by extending the BroadcastReceiver class and receives broadcasts as Intent objects.

Activities, services and broadcast receivers are activated by intents. Google suggests to think about intents as “the messengers that request an action from other components” [27], whether they belong to your application or another.

AndroidManifest.xml file informs the system about an application activities and declares which components are used by the app, which permissions are granted, API libraries the application needs, application requirements, some hardware and software features, etc.

When an application is launched, the system creates a thread for it. It is the “main” thread which executes all the work including crucially important interaction with a user, that is why it is often called UI thread. For many applications, using one thread is enough, but there can be some tasks such as network access or database queries which can block the UI thread that it unacceptable from the standpoint of application development guidelines. The UI thread should never be blocked [30]. Thus such long lasting processes are run on a separate thread. One option to implement this approach is the AsyncTask class [31]. AsyncTask allows to run the code thread-safe on another thread and publish result to the UI without managing threads or handlers manually. The task in AsyncTask subclass goes through 4 steps:

- onPreExecute();
- doInBackground(Params...);
- onProgressUpdate(Progress...);
- onPostExecute(Result).

doInBackground() contains the functionality to run in a worker thread. Three other methods are invoked on the UI thread. It is guaranteed that the parameters passed from the UI thread will be used in doInBackground(), whereas a result will fall into onPostExecute() on the UI.

2.2. Dalvik

We can face the challenges while porting the Android application functionality to the non-Android server. That is why it is important to know how Dalvik virtual machine is different from JVM (Java virtual machine), which opportunities it offers and which restrictions it has. Only the features that can influence your code will be considered.

First of all, there is an important difference in libraries, although not very wide. Java AWT (Abstract Window Toolkit) and Swing user interface libraries were replaced with Android-specific user interface libraries. Android also adds few new features to standard Java while supporting most of Java's standard features. Android's Java set of libraries is closest to Java Standard Edition. In the application framework layer, you will find numerous Java libraries specifically built for Android [32].

Another minor feature is that you use android.util.Log or other libraries on Android instead of Java's System.out and System.err.

Secondly, the "fast" interpreter on Dalvik can be created out of C "stubs" if no assembly fragments are available for the current architecture [33]. A stub looks like a complete method to the client but actually just passes on the client request to the remote service. This is Dalvik runtime-specific and is not managed automatically in Java environment. The problem can be probably resolved by writing the absent code manually, but it is going to be tedious and worthless.

Thirdly, it is known that there are many other JVM languages except for Java such as Python, Ruby, Groovy, Scala and others. Initially, the Dalvik virtual machine did not support them. Then the Scripting Layer for Android (SL4A) [34] was announced. It brought Python, Perl, JRuby, Lua, BeanShell, JavaScript, Tcl, and shell to Android and it is planned to add even more. Scala code can also be executed on Dalvik [35].

The reason why you cannot run the same bulk of languages on JVM and Android is the difference between the executable files. JMV runs .class files and Dalvik - .dex (Dalvik Executable). For the compilation processes, see Figure 2. (JVM is on the left, Dalvik – on the right).

There are more considerable differences between two virtual machines, but they refer mostly to the low-level implementation and are not likely to directly affect your code. That is why they are not described here.

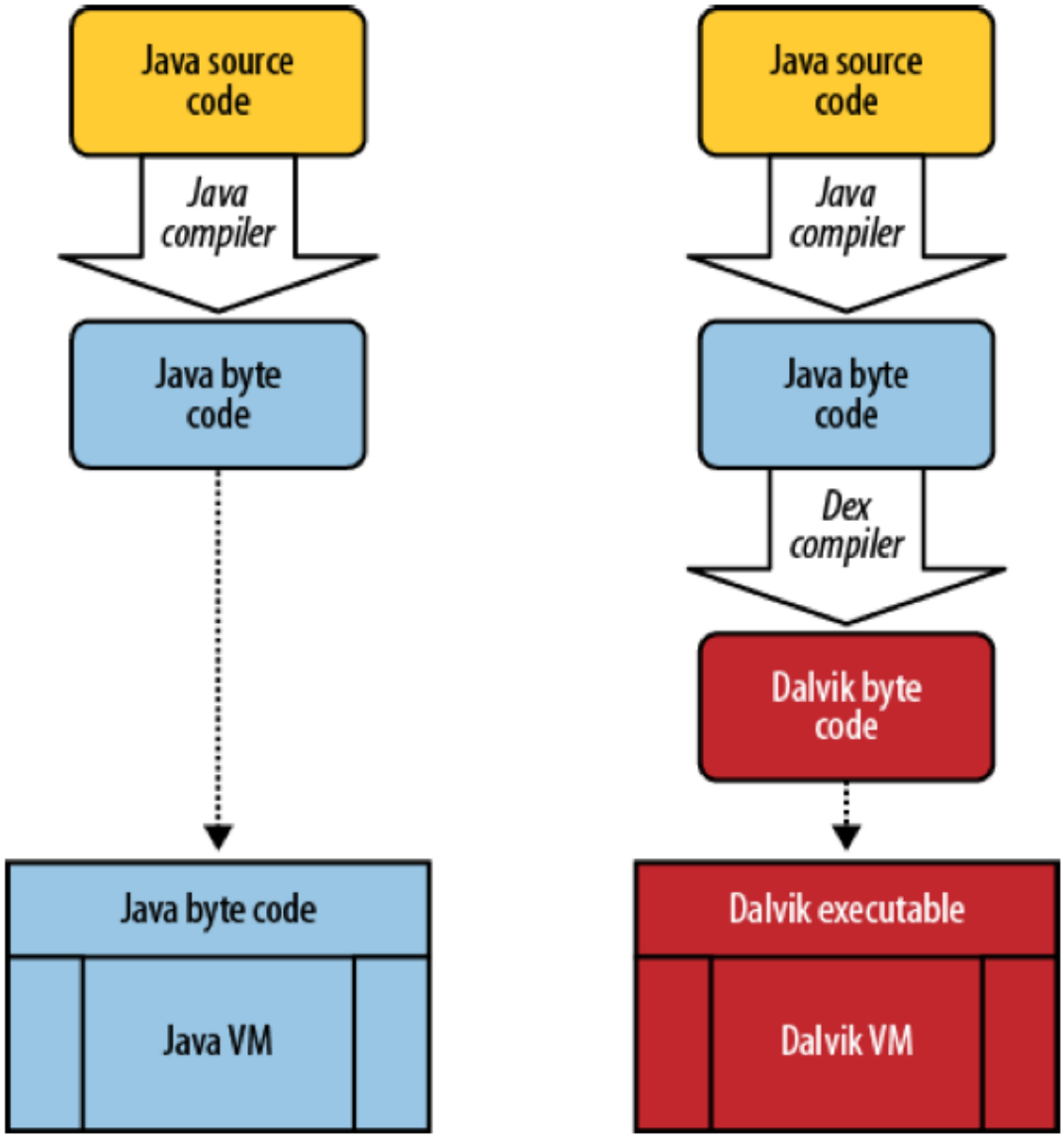

Figure 2. Java VM versus Dalvik [60].

2.3. Power measurement tools

We are going to talk about a number of power measurement tools in this section. They are all quite different and have their own peculiarities. Little Eye was chosen for this work, and the reasons are mentioned below. After that, we will take a brief look at some alternatives.

It is admitted that the currently provided power profiles may not be fully reliable. However, whatever tool we use for our measurements we do not bother about that, since our goal is to compare the power consumption before and after the employment of our approach, but not to get the precise actual numbers.

2.3.1. Little Eye

Little Eye [36] is a desktop performance analysis tool by Little Eye Labs for Android applications. It measures app's power consumption (separating total amount, CPU, WiFi and display usage), network data transfer and memory usage. Little Eye also shows the key events happening while the application is going on, such as grabbing and releasing the lock, display going on and off, the current app state, etc. After the measurement is stopped, it is possible to return to any time point and explore what was happening. The great feature is an option of building graphic report which is divided into power, data and memory sections.

The user interface of Little Eye on the desktop looks like at Figure 3.

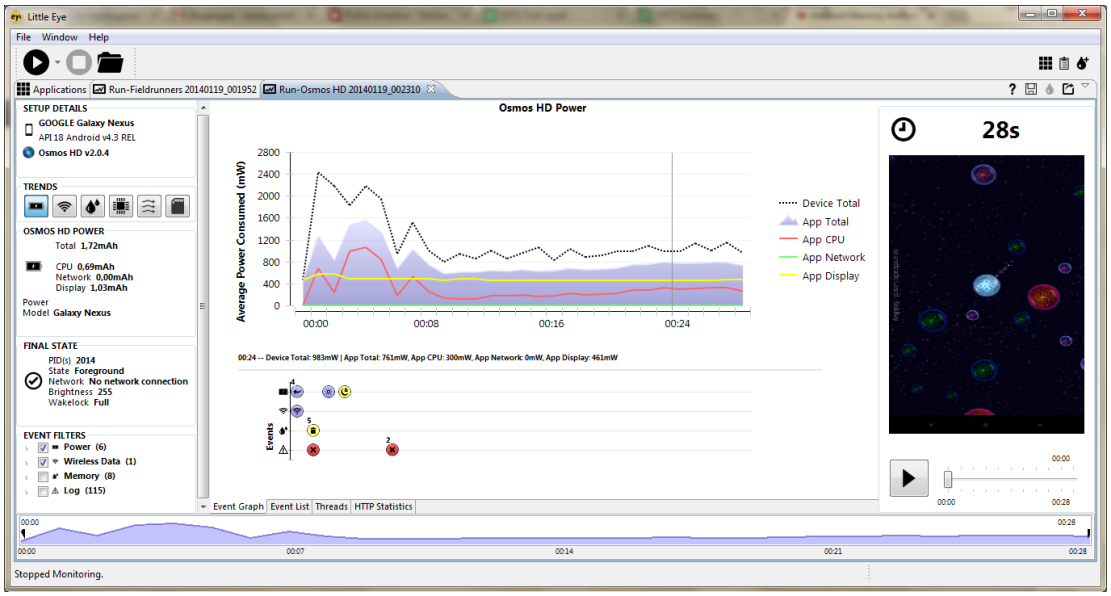

Figure 3. Little Eye UI.

The central part of a window is occupied by the power consumption diagram which is updated in real time (or any other component usage diagram if you choose is in Trends on the left). Under the Trends, the current numbers are displayed. You can see the events in the bubbles under the diagram. If the cursor is pointed at the event bubble, you can see the explanatory message about what has happened. On the right, the screen is depicted. You can go back to any point of time and see what was going on on your screen. This whole view can be saved and opened again later if you need it.

Little Eye Labs did their tests with the use of an actual power monitor [37]. The power consumed by an app is a function of the resources the app is using, like the amount of CPU being consumed, number of bytes being sent over WiFi, and so on. Little Eye has carefully built up power models that can calculate the milli-watts of power consumed. Currently, the default model device is the Nexus One.

Little Eye was available for free 3-month trial. After the project joined Facebook at the beginning of January, 2014, it is only available for the existing customers until June 30, 2014 when a new version is likely introduced.

2.3.2. Android Debug Bridge

Android Debug Bridge (adb) [38] is a command line tool which allows to get the system data such as current voltage, current, workload time, battery used, etc. needed for our research from a connected Android-powered device. adb can be found in Android SDK Platform-tools.

When you got the figures for current voltage (V), battery used (%), battery capacity (mAh), and workload duration (h), you are able to calculate the power using the following formula:

$$Power = \frac{Voltage * BatteryUsed * BatteryCapacity}{WorkloadDuration}$$

adb requires three components for its work: a client running on the development machine, a server running in background on the development machine and a daemon running in background on the emulator or device. The server listens on the local TCP port to the clients' commands. It establishes connection to the running emulators and/or devices. Where an adb daemon is found, the server sets up a connection to that port. As soon as the connection is set up, data from these devices or emulators can be retrieved.

2.3.3. PowerTutor

PowerTutor [39] measures the energy consumption by an application and system component (CPU, network, display, GPS) (Figure 4). It can help the developers evaluate the power-efficiency of their applications as well as the users can see what drains their battery. The results are claimed to be within 5% of actual values. PowerTutor's power model was built on HTC G1, HTC G2 and Nexus one with direct measurements. As for the reports, the application stores the history and generates text-files with detailed information.

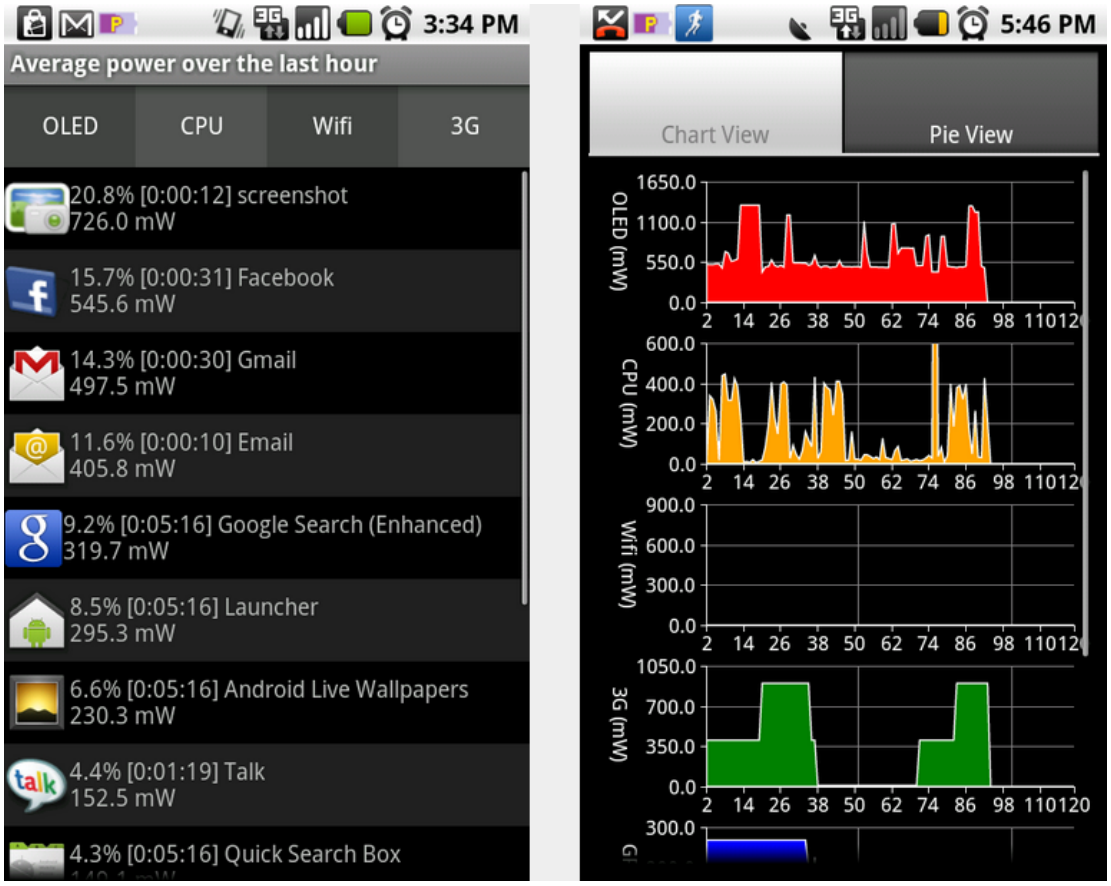

Figure 4. PowerTutor UI.

2.3.4. Andromedar

Andromedar power measurement application was programmed as a diploma project by Marcel Schröder in Carl von Ossietzky Universität, Oldenburg, Germany [40].

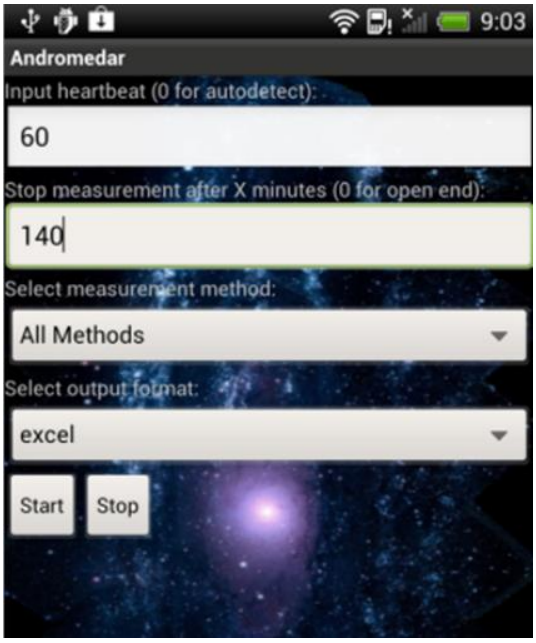

Figure 5. Andromedar UI.

Figure 5 depicts its main functionality. The first field is used to set the interval in which the device power measurements must be done. The second field fixes the duration – how low the measurements must run. As the third step, a user can choose a measurement method if it is clear which one is better in a current situation: based on Android BatteryManager API, vendor's power profile from the device's system or any other power profile from an external XML-file. By default, all methods are included. Fourthly, an export format for measurement

results can be selected. When the measurements finish, a data file is stored in the memory.

2.3.5. Yocto-Amp

It is also possible to measure energy with some hardware tools, for example, Yocto-Amp [41].

This digital ammeter (Figure 6) makes it possible to measure current automatically and per component with high precision. You just connect your mobile device to Yocto-Amp with USB. It also provides a good API to operate the data. In addition, the device can be connected directly to an Ethernet or WiFi network.

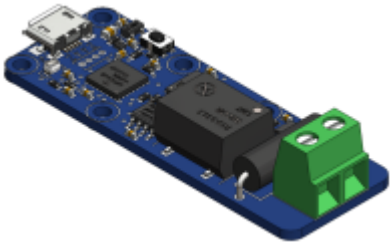

Figure 6. Yocto-Amp.

The modules provides immediate reading on USB, and can also store measures on the device internal flash for later retrieval when connected again by USB.

2.3.6. Which power measurement tool to choose?

Each tool has its own pros and cons. In this work, Little Eye is used.

First of all, it has a rich functionality and all the needed statistics. Moreover, it has a clear UI and you can run an app on a device and observe the real time results on your desktop. It gives a great opportunity to see when exactly the app eats most of power and how it behavior alters. In addition, Little Eye provides a very convenient and vivid form of presenting results.

So, Little Eye is the most convenient tool of all revised, which suits our aims perfectly.

ADB gives just row data that needs to be analyzed programmatically or manually. Besides, it does measurements for the whole device, not per application, that is why, in case it is used, we lose precision and have to do additional job to get results for an app.

PowerTutor and Andromedar run entirely on a device. Consequently, a user cannot interact with a measured app and investigate the resource consumption behavior change simultaneously (that was a significant advantage of Little Eye). What is more, it is easier to extract and save graphic representation with Little Eye that PowerTutor, as it saves only text data. Andromedar has only excel reports as well and measures the power usage for the device, not a single application.

Hardware tools like Yocto-Amp give true results, not based on power profiles or somebody’s power models. But it requires a lot of (in context of this work) distracting work, such as having the ammeter itself, writing some code or having a rooted device. For our purposes, precision is not crucially important: the measures need to be compared. That is why, the possible great opportunity to use hardware can be other tasks, like writing your own power profile.

2.4 Other tools

2.4.1. Vert.x

Vert.x is an application platform for the JVM. It is used in this work to facilitate the implementation of HTTP-communication between a client and a server. Vert.x allows us to write an HTTP server simply, handle requests and send responses and not to refrain from using Gradle [42]. Vert.x is free and open source, licensed under the Apache Software License 2.0.

2.4.2 OpenShift

OpenShift is an open hybrid cloud application platform (PaaS) which allows to host your applications. It is used in this work to host the server-side applications. While creating the application, it is possible to choose from a number of frameworks and databases available as well as a Do It Yourself cartridge to customize it for not pre-installed tools such as Vert.x.

2.4.3. HTTP Communication

The server and client sides communicate via the hypertext transfer protocol (HTTP). In our case, information is written to JSON (JavaScript Object Notation) strings and put into the body of HTTP requests and responses. For serializing and deserializing data, gson Java library is used which makes transformations in one line of code (methods toJson() and fromJson()). Thereby a REST architecture is built.

2.4.4. JCodec

Making a new video from separate images or changing an existing one is not a trivial task. JCodec library [43] makes it simpler. It is an open source Java implementation of video and audio codecs and formats. In September 2013, the Android version was released. In this work, the library is used for getting frames from a video (decoding) and encoding a new mp4 file.

As an alternative, we tried Xuggler by Xuggle [44], but this library is very big – 44 times bigger than the whole application – and we refused to use it.

2.4.5. Patterns

Two design patterns are used in this work: proxy pattern and command pattern.

Proxy pattern [45] [46] is applied to create a placeholder for the class and control the access to the class. Initially, there were a number of classes (ClassName) in the project. In order to vary the implementation of these methods depending on the connection presence, new classes were created (RemoteClassName). If checking connection returns positive result, we instantiate an instance of the RemoteClassName which in its turn does not have the full implementation of the method, but calls it on a real remote class in the cloud. Thus, a remote class is only created on demand.

Command pattern [47] [48] allows to encapsulate a request as an object and diversify the method call. For instance, two classes have the

method yourMethod(), and corresponding to some condition (in our case, Internet connection presence), the variable obj is initialized with the instance of one of these classes. When obj.yourMethod() is run, the command specific for this class will be executed.

The patterns are used in order to make the application architecture clear and the code - reusable.

3. Approach

This section addresses some strategies to apply in order to implement our approach.

First of all, you need to find suitable blocks of code to move to the cloud. It is not the aim of this paper to define all the power-hungry features to be extracted, but the number of them will be enumerated. Some of them were initially obvious, others have come to light during our work.

Secondly, it will be discussed how to organize client-server communication to exchange data. Two ways are going to be described. Both of them were used in our experiments.

After that, we will touch the test cases to measure the energy consumption of the applications running solely on a mobile device - “locally” - and while applying our approach – partly “remotely”.

3.1. What to look for

It looks sensible to find:

- the code containing complicated calculations which load the CPU significantly and may take a lot of time;
- ideally, methods with a few lightweight parameters and returned valued and lots of sophisticated functionality inside. It appears to bring the highest efficiency. As we will see later, you will not benefit from transferring a huge deal of data over the network sometimes;
- already existing AsyncTasks. All long running operations, on the one hand, should be implemented as asynchronous tasks, on the other hand, it is beneficial to run them remotely. HTTP communication should be done as an AsyncTask in order not to block the app, as the connection is unpredictable.

These are the fragments that are the most convenient to extract or can bring the largest benefit.

3.2. How to port the app to the cloud

This section is devoted to a very important issue – porting an application to the cloud. A general scheme (see Diagram 1) is going to be described: what has to be done to run the parts of the mobile app’s functionality remotely.

The first actions on the diagram: identify the code blocks, adapt the code, setup client-server communication and port the code – refer directly to the approach. Finally, the power consumed should be measured to verify the effectiveness of the approach.

First of all, the code to transfer to the cloud should be found. The way to use here is static program analysis: parsing the source code of

the application with an automated tool or just reading to find the most power-hungry blocks (described in the previous section, see “What to look for”).

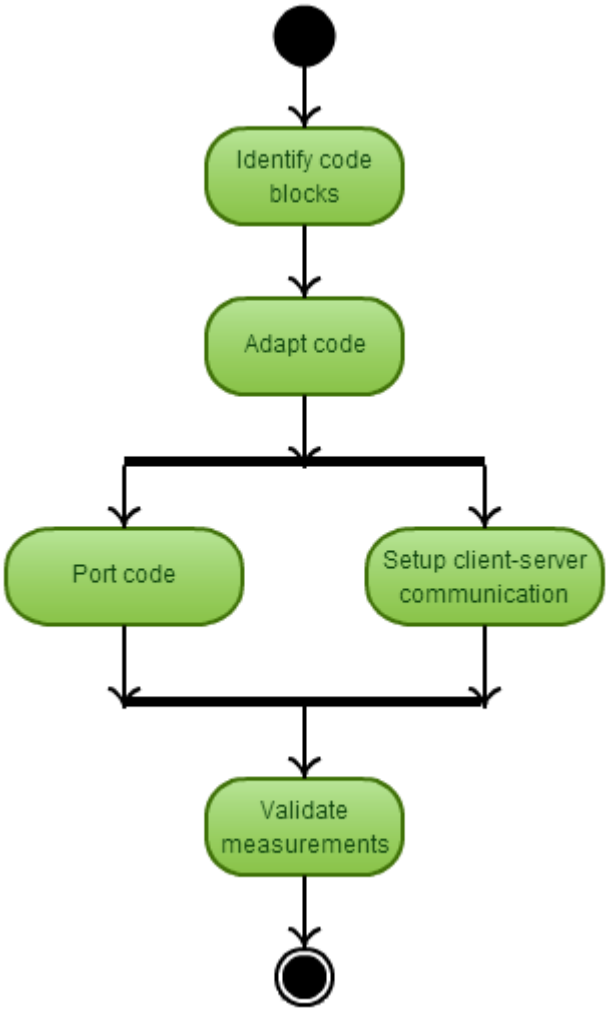

Diagram 1. Porting application to the cloud.

After that, make sure the chosen functionality is fit for running remotely. That means:

- it is a distinct method;
- it returns a result value;
- it does not use Android-specific resources, such as UI or system services;
- it does not modify the external state of the application inside without returning the modified values;
- most likely, it does not use platform-specific classes, stubs or native code.

It can be useful to extract the chosen methods so as they will look like on a remote server. Program new classes which implement the required functionality, substitute the initial methods with new ones and run the application on a device (locally). The “new” code will be the same or almost the same as it was before – those methods that you have chosen to port. At this point, if everything works as expected, the half of the work is done.

The last two approach activities are executed in parallel: it is necessary to implement client-server data exchange and move the chosen functionality to the server-side application.

Two ways to transfer data are studied in this work: one is based on serializing values into JSON strings and another uses sending files. In both cases, HTTP communication is applied. For details, see the next section, “Client-server HTTP communication”.

The client-side application has to contain all the methods that you are going to call from the mobile app. If you have nevertheless included the methods that use mobile platform specific code (native code, platform dependent libraries, etc.), make sure you are able to run it on the server. In addition, the server application needs a class(es) to parse the HTTP-responses, obtain transferred data, call the appropriate method, and send back results.

At the end, verify that the transition has improved energy consumption. For this purpose, chose a suitable tool and conduct a number of power measurements against various conditions and restrictions. Pay attention to the connection speed, the amount of data to be transferred via the net, time a user needs to execute the task, and possible performance change.

3.3. Client-server HTTP communication

HTTP communication between a client and a server is the main integral part of our approach. In this section, two ways which we have used are explained.

One way to implement the connection between the Android application and the remote server can be the following. A method under consideration – that is, which we are going to run remotely – is usually called from another more complicated method (the code here and later is not the real apps' code, but the pseudocode, that resembles our applications and helps to understand the core features of the approach):

```
1. class SomeClass {
2.     …
3.     someMethod() {
4.         …
5.         ClassName obj = new ClassName();
```

```
6.          obj.yourMethod();
7.          …
8.      }
9.      …
10. }
```

```
1.  class ClassName {
2.      …
3.      yourMethod() {
4.          …
5.      }
6.  …
7.  }
```

Here the proxy and command patterns are used: there are usually two classes – ClassName and RemoteClassName (which would rather extend a common class and implement a common interface):

```
1.  interface CommonInterface {
2.      …
3.      yourMethod();
4.      …
5.  }
```

```
1.  class CommonClass {
2.      … // parent class for ClassName and
            RemoteClassName
3.  }
```

```
1.  class ClassName extends CommonClass implements
            CommonInterface {
2.      …
3.      yourMethod() {
4.          … // “local” implementation
5.      }
6.      …
```

```
7. }
```

```
1. class RemoteClassName extends CommonClass
        implements CommonInterface {
2.     …
3.     yourMethod() {
4.         … // “remote” implementation
5.     }
6.     …
7. }
```

If there is no Internet connection, the instance of ClassName is created and the app executes as it did before modification, but if the connection exists, RemoteClassName class is used (Figure 7). SomeClass changes:

```
1. class SomeClass {
2.     …
3.     someMethod() {
4.         …
5.         CommonClass obj;
6.         if (connected) {
7.             obj = new RemoteClassName();
8.         } else {
9.             obj = new ClassName();
10.       }
11.       …
12.   }
13. ...
14. }
```

Connection presence is checked with our ConnectionUtil class with the help of Android ConnectivityManager.

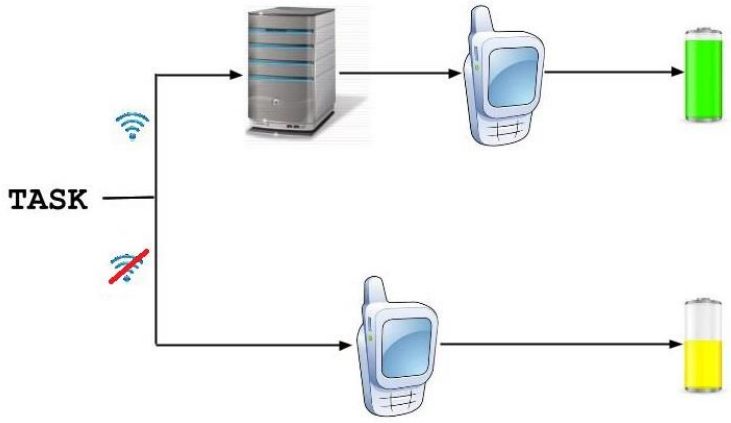

Figure 7. Decide where to execute a task.

Certainly, there are methods that cannot be extracted to the server. Some reasons for that will be defined later in this thesis. Initially, we have a pool which includes methods of an app. For each method, we decide whether it can be executed remotely. If it can, we send its name, name of a class (if needed), and input parameters to the cloud where they are handled and a result is sent back. The result is the same as if it was produced by this method on a mobile device.

Diagram 2 illustrates this scheme.

In the called method of the “remote” class, class name, method name and parameters are packaged into the object of PostRequestParams (another our auxiliary class written especially for this purpose).

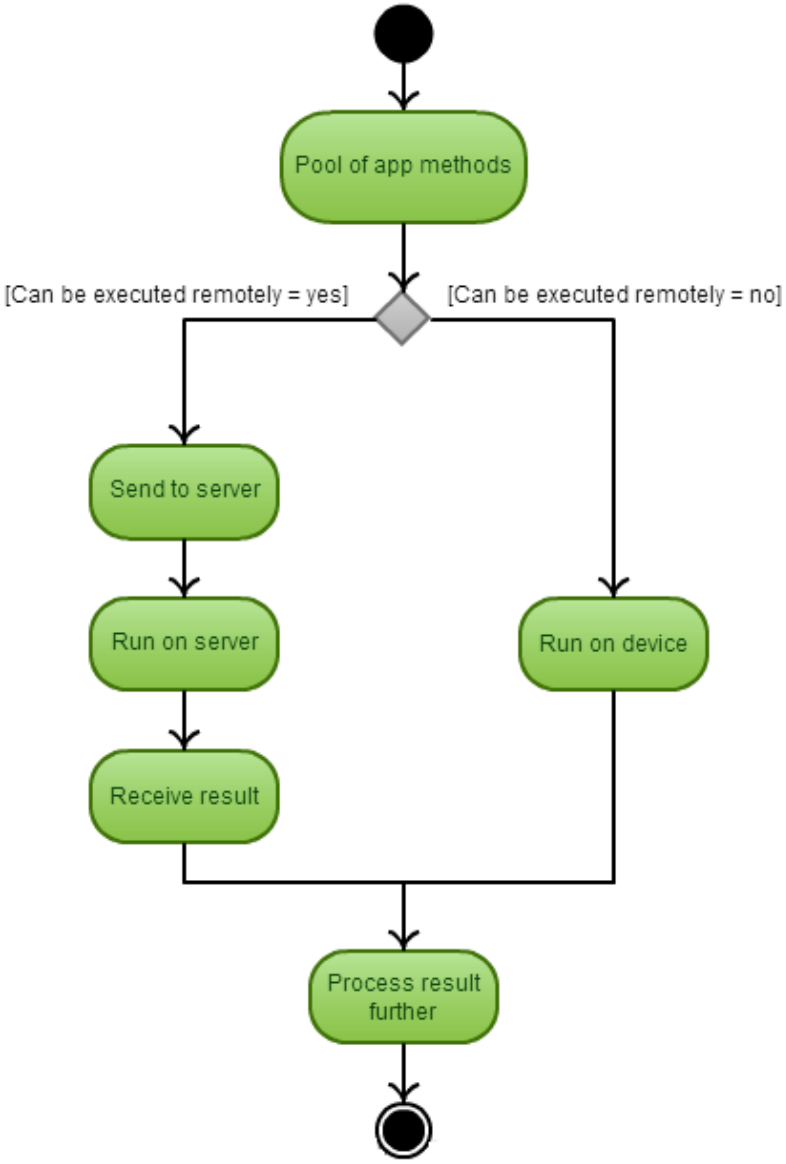

Diagram 2. Run a method remotely or locally.

```
1. class RemoteClassName extends CommonClass
       implements CommonInterface {
2.     ...
3.     yourMethod() {
4.         ...
5.         PostRequestParams params = new
            PostRequestParams(className, methodName,
            otherValues);
6.         receivedResult = new
            PostRequestTask().execute(params);
```

```
7.          result = gson.fromGson(receivedResult,
             ResultClass.class);
8.          ...
9.      }
10. ...
11. }
```

After that, PostRequestTask().execute(params) (which is an AsyncTask) is called. PostRequestTask writes params to JSON string (Gson's toJson()), establishes connection to the remote server and executes post request (org.apache.http.client.methods.HttpPost). Method with passed parameters is run in the cloud. SomeTask class gets a result back in the form of JSON from the response body (org.apache.http.HttpResponse). Finally, the method returns the result to where it was called from where it is read from JSON into ResultClass or PostRequestResult (if the JSON string contains several classes or just independent values).

```
1.      class PostRequestTask extends
AsyncTask<PostRequestParams, ProgressType,
ResultType> {
2.          doInBackground(PostRequestParams... params) {
3.              ...
4.              establishConnection();
5.              postRequest.setEntity(params.toJson());
6.              response =
client.execute(postRequest).getEntity();
7.              result = toString(response);
8.              return result;
9.          }
10.    ...
11.    }
```

Classes PostRequestParams and PostRequestResult are created for two reasons: firstly, AsyncTask takes only one value type for parameters and one for result, and secondly, it is simpler to work with one class at once and to prevent mistakes. These classes consist of the required fields to set all the values to be sent, setters and getters, plus the fields for class and method names for PostRequestParams.

On the server side, the incoming request (org.vertx.java.core.http.HttpServerRequest) is caught. JSON from request body is parsed, PostRequestParams object is derived which allows to get the required method with the passed parameters. The server application contains the versions of the Android classes with fields, methods and inner classes which are supposed to be run remotely. According to the extracted method name, the instance of the corresponding class is instantiated and needed method is called. He result is written to JSON and sent back to the Android application in HTTP response body (org.vertx.java.core.http.HttpServerResponse).

```
1.  class RequestHandler {
2.      …
3.      contentType = request.headers().get(“Content-
            Type”);
4.      switch(contentType) {
5.          case “application/json”:
6.          …
7.          PostRequestParams params =
            gson.fromJson(requestBodyString,
            PostRequestParams.class);
8.          className = params.getClassName(); // if
            needed
9.          methodName = params.getMethodName();
10.                 switch(mothodName) {
10.                         case…:
11.                                     …
12.                                 result =
            gson.toJson(resultValues);
13.                                     …
14.                         }
15.                     …
16.             }
17.             response.headers().set(“Content-Type”,
            “application/json”);;
18.             response.end(result);
19.           case “application/octet-stream”:
20.                         …
21.         …
```

```
22.         }
```

The method described above is implemented in first two examples: FiniteElementCalculator and AsciiCam Image.

The second approach is to simply exchange files when it is feasible. For instance, it can be used to send music or video, as we do it in AsciiCam Video.

In this case, we do not need an intermediate class and implement RemoteClassName as an AsycnTask child class. It will write the file into a request body, execute the request and save result file when it comes back.

```
1.  class SomeClass {
2.      …
3.      if (connected) {
4.          obj = new RemoteClassName();
5.          obj.execute(filePath);
6.      } else {
7.          obj = new ClassName();
8.          obj.yourMethod();
9.      }
10.    …
11. }
```

```
1.  class RemoteClassName extends AsyncTask<String,
            ProgressType, ResultType> {
2.      doInBackground(String... filePath) {
3.          …
4.          httpPost.setEntity(new FileEntity(file,
             “application/octet-stream”));
5.          response = request.execute(httpPost);
6.          getResultFileFromResponse(response);
7.          …
8.      }
9.  }
```

The server again gets the request, determines its content type, reads the file from the request body, processes it as it is required by the task and sends the output file back.

```
1.  class RequestHandler {
2.      ...
3.      contentType = request.headers().get(“Content-Type”);
4.      switch(contentType) {
5.          case “application/json”:
6.              ...
7.          case “application/octet-stream”:
8.              inFile = getIncomingFile();
9.              outFile = processFile(inFile);
10.                 response.headers().set(“Content-Type”,
                    “application/octet-stream”);
11.                 response.write(outFile);
12.                 response.end();
13.         ...
14.     }
15. ...
16. }
```

Thus, two way to exchange data are sketchy illustrated at Figure 8.

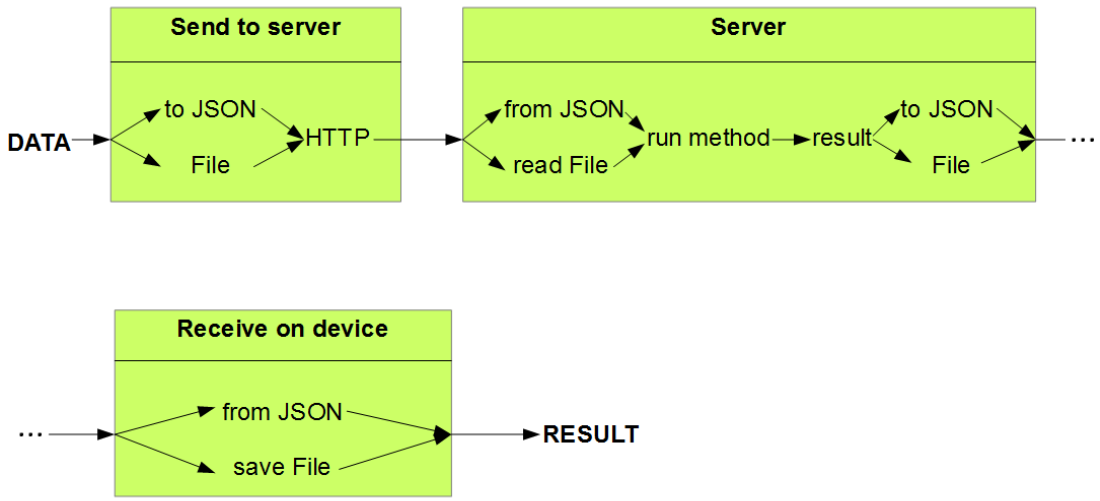

Figure 8. Sending input data and receiving method result.

3.4. Test cases for power measurement

So, we have three applications which can process data locally (on a device) and remotely (in the cloud). They are going to be tested against:

- various volume of data to be handled;
- various Internet connection speed.

Thus, test cases imply that each application will be run in three modes: on a device, using the cloud and fast Internet connection, and using the cloud and slow Internet connection.

In addition, they use diverse ways to transport information which also can be compared.

4. Test applications

Three hypotheses of the work are:

- the energy needed for data transmission from device to a server and receiving back a result can be less than the energy consumed for executing the same functionality at a device;
- there can exist some “heavy-loaded” code blocks which can be extracted without hurting an application and executed remotely;
- performance can change to the worse as well as to the better.

To verify these hypotheses, three applications were chosen: FiniteElementCalculator, AsciiCam Image, AsciiCam Video. This section describes their functionality, architecture, transition to the cloud, problems which were managed, and power measurement tests. It shows how to implement our approach and what results to wait for.

Test applications are just the tools to reach our objectives. The most interesting for a reader parts of this chapter should be those about how we transformed the app to run it partly in the cloud, what app's features are important for our approach and how efficient it can be.

4.1. FiniteElementCalculator

The FiniteElementClaculator Android application was written by us on the basis of open source Java-code [49] for a desktop application in order to conduct the power measurements within this work. It owns a number of features described in previous sections: it performs complex mathematical calculations based on significant but not very large amount of data, loads CPU greatly and uses asynchronous tasks. All these make it an ideal app for our experiments. The data to manipulate is small enough to benefit to the fullest from bringing the functionality to the cloud, but also large enough to be compared to the amount of data that can be used in real useful applications. FiniteElementClaculator itself can be hardly called useful, except for our purposes.

4.1.1. Application architecture

The FiniteElementCalculator code implements the finite element method [50] for solving the problem of solid plasticity. The app's logic is following: 1. pass a matrix of points represented as an array of arrays (contains integer and floating point values); 2. solve a system of differential equations; 3. return a result as a two-dimensional array (floating point values). Input values are read from files on a device. There are 5 cases representing different files to read from. The files vary in data volume.

Thus, the application completes some sophisticated numerical computations.

The algorithm must have used a significant amount of data to transfer to the cloud, so that it was comparable to the volume of data being received and returned by the real applications. We could send only a number of a case and obtain all the input values from files on a server, but in this case we will gain even more benefit as a device will not waist resources on reading files and transferring large information via WiFi.

The application has only two activities (Figure 9). The first is used by a user to input the number of case and confirm it with a button and the second one displays the time required for a task to be completed.

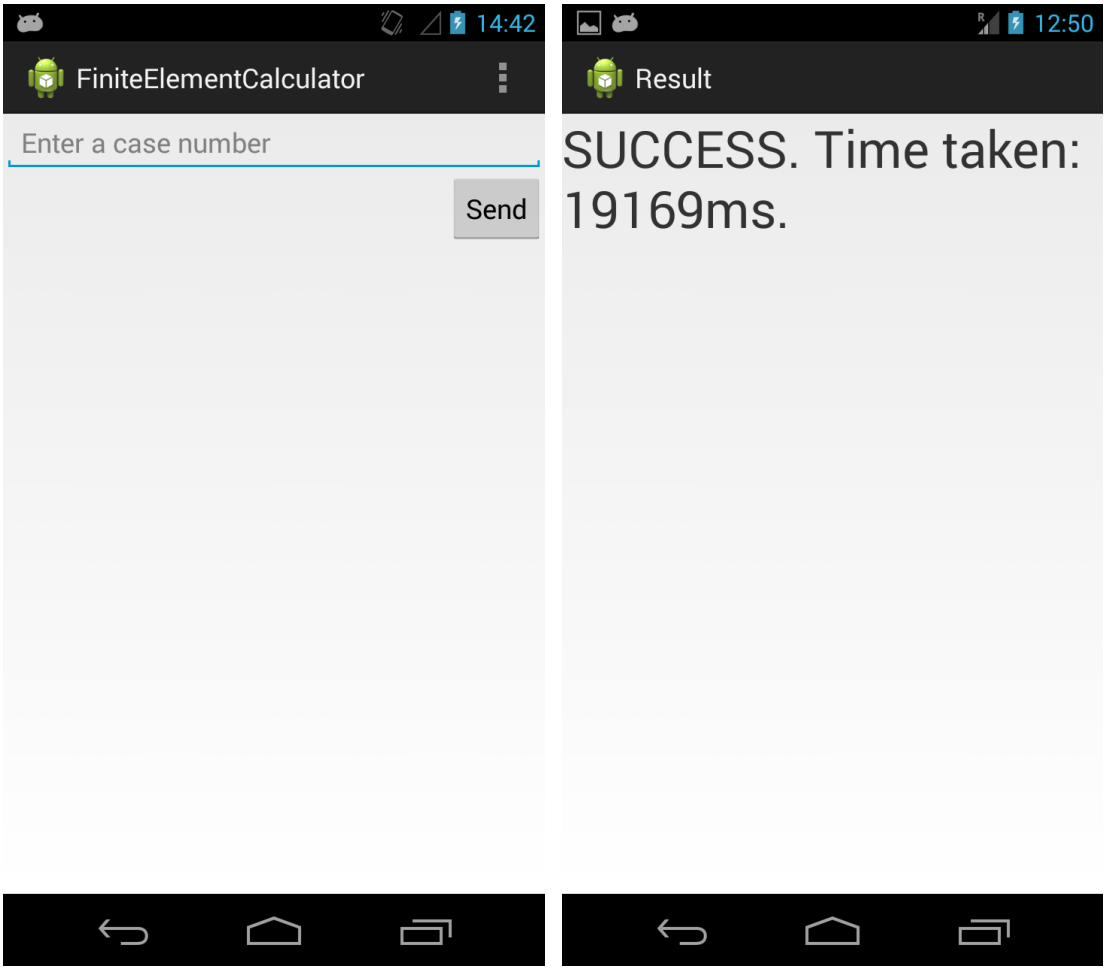

Figure 9. FiniteElementCalculator UI.

4.1.2. Porting to the cloud

Porting functionality to the cloud is not very complicated in this case. Actually, we have the same code on both sites, but there are only some classes (not all) with particular fields and methods on the server – those which are called remotely.

The device-cloud interaction is implemented as explained above in the “Client-server HTTP communication” part of the “Approach” chapter. For this purpose, HttpUtils and AsyncTask classes were added.

Some minor refactoring of initial source code was made in order to make it feasible to send and receive values via the net and use them in AsyncTask methods.

4.1.3. Power measurements

Test cases presuppose that the application is run in three modes: on a device, using the cloud and fast Internet connection, and using the cloud and slow Internet connection. Battery drain, CPU load and time spent are measured.

Firstly, only one case is used. This means, the application reads user input, takes values from a file and executes computations once.

When the Internet connection exists, the app sends and receives data from the server again only one time. The test finishes.

After that, the app gets 3 or 5 user's input cases (one by one) and runs calculations 3 or 5 times, correspondingly. This means, it will switch from a device to a server and back several times. This policy resembles a real application to the greater extent than the previous one in that way that real applications are likely to work on a device and server in turn.

The results of these test cases can be observed in the table below. The volume of outcoming and incoming data is deliberately not summarized in the second and third cases to differentiate between calls to the server.

The percentage of battery drain is a forecast of consumption for an hour which is based on the amount of energy eaten during the test.

	Local			Fast connection (D 6-12 Mbps/ U 7,7-12)			Slow connection (D 0.9-2.4 Mbps/ U 0,6-0,8)			
	Battery, %	CPU, %	Time, s	Battery, %	CPU, %	Time, s	Battery, %	CPU, %	Time, s	Data volume (out/in), Mb
FiniteElementCalculator										
1 case										3.2 / 103.34
Average	29.3	31.8	32.3	17.35	3.83	15.3	16.6	2.9	20.3	
Standard deviation	1.26	2.02	1.53	0.7	1.2	7.5	0.15	0.48	4.04	
3 cases										3.2 + 17.92 + 8.16 / 103.68 + 72.24 + 122.12
Average	30.3	36.3	123.3	16.6	3.13	52.3	16.7	2.3	72.3	
Standard deviation	0.41	0.48	4.62	0.39	0.1	3.79	0.25	0.31	9.7	
5 cases										1.63 + 3.2 + 3.2 + 17.92 + 8.16 / 111.79 + 103.34 + 103.68 + 72.24 + 122.12
Average	30	36.4	181.7	16.9	3.2	78.7	16.7	1.87	136.3	
Standard deviation	0.27	0.19	4.62	0.3	0.36	10.4	0.12	0.04	3.79	

Table 1. Power measurements for FiniteElementCalculator.

According to the test results presented in the table, we can save a lot of device's battery power if a part of functionality is delegated to the remote server. It appears that the CPU load decreases dramatically.

The following diagrams illustrate the energy consumption during the first case.

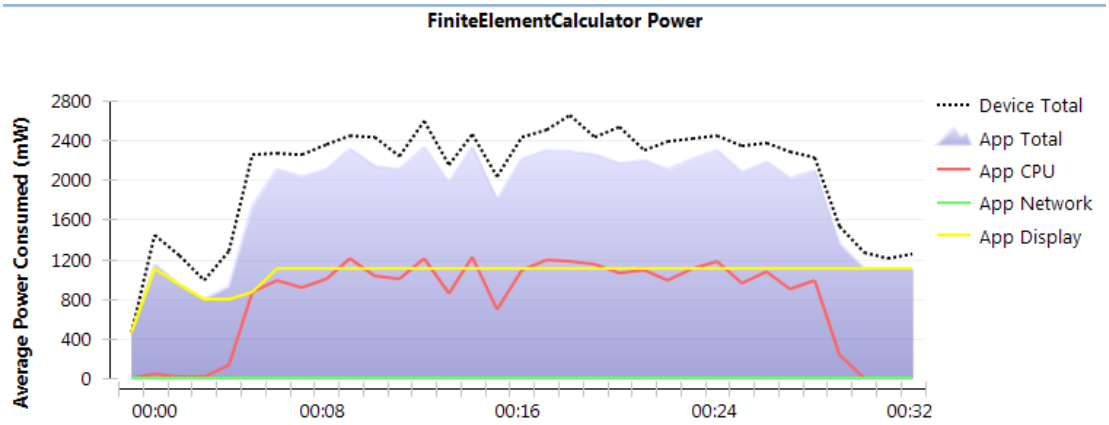

Figure 11. FiniteElementCalculator power consumption while running on a mobile device.

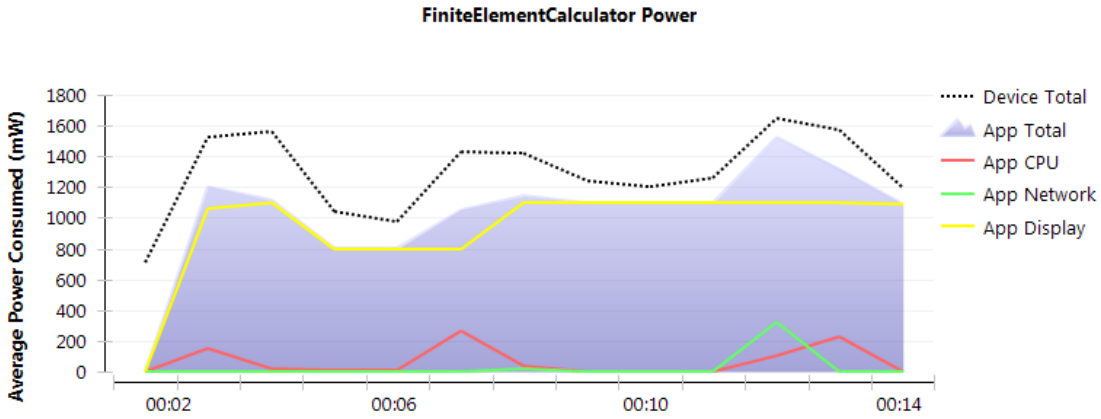

Figure 10. FiniteElementCalculator power consumption while using the cloud.

The power consumption decrease is gained by CPU loading relief. The figures below show how the behavior changes.

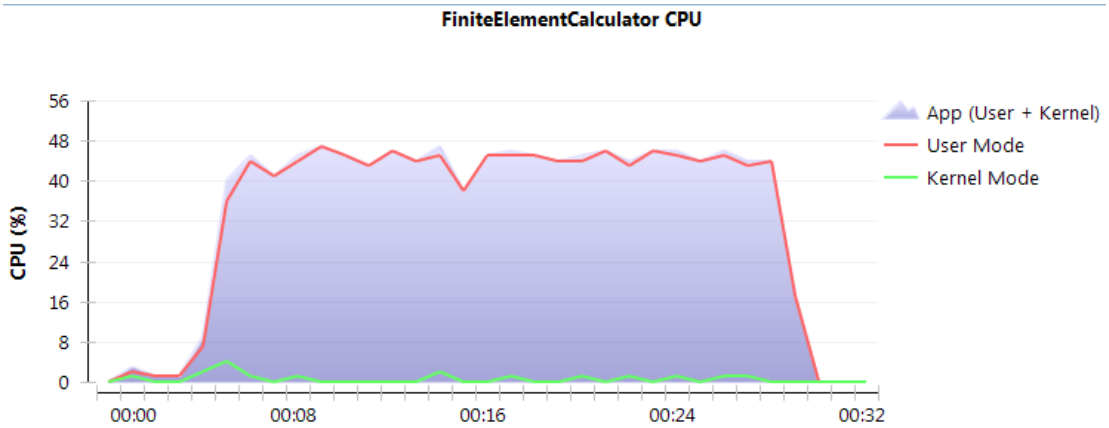

Figure 12. FiniteElementCalculator CPU load while running on a mobile device.

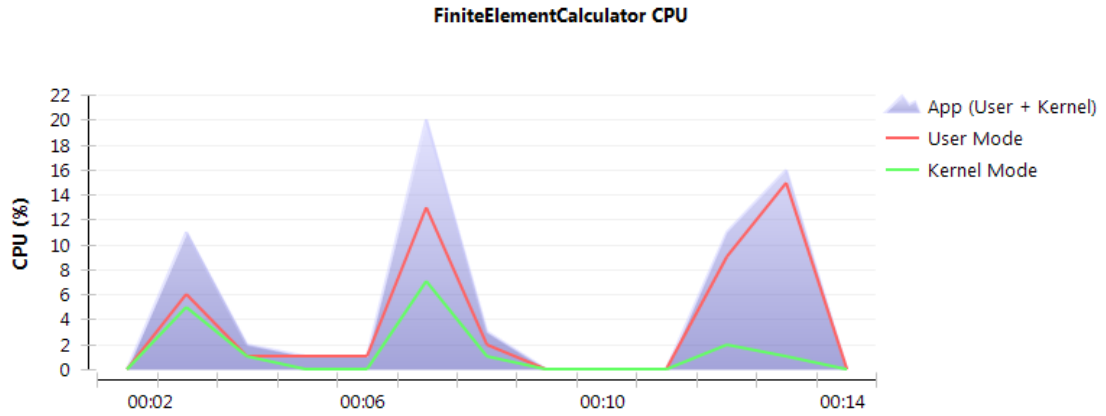

Figure 13. FiniteElementCalculator CPU load while using the cloud.

The two following pie diagrams display how greatly power consumption has altered. Test scenarios are the same in these cases.

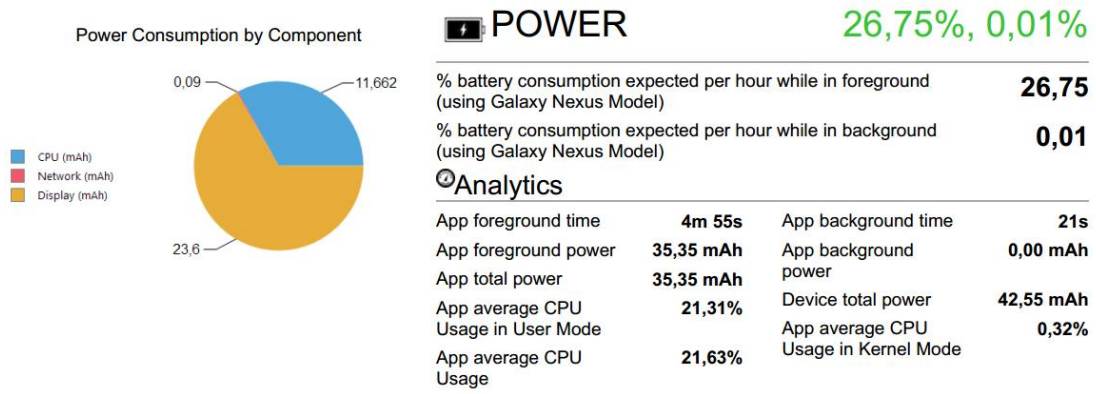

Diagram 3. FiniteElementCalculator: power consumption by component while running on a mobile device.

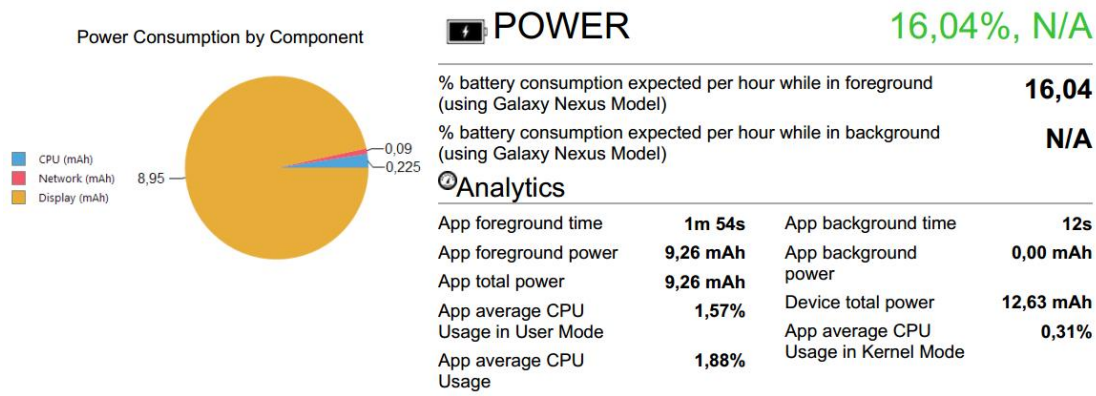

Diagram 4. FiniteElementCalculator: power consumption by component while using the cloud.

Moreover, it takes less time to execute the task remotely. That is why, if we need to perform several operations we will save even more power with a server.

4.1.4. Result

The FiniteElementCalculator application is a good example of how the power consumption on a mobile device can benefit from using the cloud computing.

The app migration to the cloud was easy due to the absence of platform-specific code and existing asynchronous tasks. It was only necessary to implement the HTTP-communication and build the server-side application with needed classes.

The energy consumption was decreased significantly owing to the relatively small amount of data to be packed into JSON and transmitted over the net. In that way, the CPU was not loaded with the heavy calculations, and work with JSON hasn’t required much of its resources.

4.2. AsciiCam Image

AsciiCam application is provided by Dozing Cat Software [51]. AsciiCam converts an image that Android camera is pointed at into an ASCII picture in real time. In addition, a user can make an ASCII version of the images already stored in the gallery. Black-and-white, primary colors and full colors options are available. AsciiCam is completely free and open source [52], contains code from the Android Open Source Project licensed under the Apache License, Version 2.0 [53], copyright by Brian Nenninger [54].

Figure 14 shows the initial picture and an image made from it with AsciiCam.

Figure 14. Original pictures and transformed with AsciiCam.

4.2.1. Application architecture

The main computation method takes camera input data, number of ASCII rows and columns to convert to, and the ASCII characters to use, ordered by brightness. For each ASCII character in the output, it determines the corresponding rectangle of pixels in the input image and computes the average brightness and RGB components if using color. A bitmap is built. Final images are can be saved in PNG and HTML formats. For existing pictures, an ASCII image is built from an existing bitmap. The application also creates thumbnails for showing pictures in the gallery.

4.2.2. Porting to the cloud

Extracting functionality to run locally

First of all, some permissions were added into AndroidManifest.xml file in order to be able to connect to the Internet, get internal network information and read the low-level system log files.

The first part of the experiment was to extract methods which will be later executed remotely and run the application locally, using these extracted methods.

The most attention was paid to the AsciiConverter class which contains the core, most complicated methods: computeResultForRows() and computeResultForBitmap().

computeResultForRows() is the main computation method. It takes camera input data, number of ASCII rows and columns to convert to, and the ASCII characters to use ordered by brightness. For each ASCII character in the output, it determines the corresponding rectangle of pixels in the input image and computes the average brightness and RGB components if using color.

computeResultForBitmap() builds an ASCII image from an existing bitmap and is used to convert the pictures already stored on a device.

All the permutations are done according to the processes described in “Client-server HTTP communication” section of “Approach” chapter. In order to extract the methods, Converter interface was created and implemented by AsciiConverter and RemoteAsciiConverter classes. computeResultForRows() initially returned nothing (void). As we need to get any result from the server in the future so that the image could be built or updated, the method was modified and now returns a custom Converter.Result type value which is used to make a picture. Probably, this will not work remotely, since the image will not then be produced in real time, a long delay is expected.

The method computeResultForCameraData() of the same class uses toPixelCharArray() which is not so complicated but runs a long for-cycle and can also be extracted. The Pixel interface is created and implemented by PixelUtil and RemotePixelUtil. The method is also used in AsciiConverter.computeResultForBitmap(), but in this case the whole

method is already extracted so it does not requires a separate call for remote toPixelCharArray() execution.

In AsciiRenderer class, createBitmap() was extracted (is used in CreateBitmap() and createThumbnailBitmap()) - connection is checked and if it exists, the method is run remotely (RemoteBitmap class).

The whole methods of AsciiRenderer class cannot be extracted as they directly interact with Android components such as Canvas or threads.

For all the modifications above, proxy and command patterns were applied as explained earlier in Patterns paragraph of the Tools chapter. Actually, the RemoteClassName classes are the copies of the initial classes, but contain only the needed components.

Other application methods are not suitable because they are not too complicated, or directly interact with Android resources (R), Canvas, etc., or have side effects (ex. Bitmap.compress() in AsciiImageWriter methods that instantaneously modifies the passed bitmap), or deal with threads (ex. AsciiConverter.computeResultForCameraData()).

At the end, application runs successfully with new classes, no errors, the functionality is fully saved.

Moving to the cloud

In order to port code to the cloud, three things must be implemented: firstly, a server application which will receive, process requests and execute methods, secondly, a PostRequestTask class which will contain all the HTTP connection stuff and result processing, and thirdly, RemoteClassName classes must be rewritten so that they contain only the PostRequestTask execution call but not the whole implementation of the methods.

Again, all the transformations comply with those explained in a “Client-server HTTP communication” part of the “Approach” chapter.

Server application consists of a class that receives requests and writes responses, the classes which were earlier created in Android application in respect to proxy pattern (RemoteClassName classes; now they are just ClassName classes as those in the mobile application) and some other classes that ClassName classes require. The class that processes requests acts as described in a “Client-server HTTP communication” paragraph in “Approach” chapter.

PostRequestTask class in Android application extends AsyncTask. It receives the parameters of PostRequestParams type and produces a String as a result. Parameters contain the class name, the method name which is going to be called, and the parameters to this method. The HTTP client is created and parameters are send to the server in the POST-request body in JSON format. PostRequestTask.execute() is called from the RemoteClassName class' method. After getting response from

the server, PostRequestTask returns its entity as a JSON-string to the method which called it.

As it was mentioned above, RemoteClassName calls PostRequestTask.execute(). Method parameters are packed into PostRequestParams' child class in accordance to this particular method. Besides, at the beginning, it can initialize some variables and even do some operations in case not the whole method is ported to the server. At the end, method gets a string from PostRequestTask and deserializes JSON to the result value.

Lessons learned, or why the mobile application will not work on another platform

Eventually, we have a client-server application which communicates via HTTP. Android application sends some information to the cloud where it is handled and receives back a result.

Everything would be perfect, if not for some crucial details that break the system.

android.graphics.Bitmap is not going to work on the server without Android. It's a platform specific library, and calling its constructor (that is used in computeResultForBitmap()) causes Stub! error (see “Dalvik” section in “Tools” for more information about stubs). This indicates the existence of some interface which is accessible only from Android but not for our server. The problem in this case was solved by tearing up the bitmaps and using the java.awt.image.BufferedImage on the server. It was feasible as these two classes have the similar methods, but the same solution cannot be guaranteed for all other cases since it may happen that you will not find a resembling Java class.

computeResultForRows() - the main computation method that produces result image that we see in real time on the screen - if run in the cloud, seems not to be efficient. HTTP communication succeeds, but there are only 2-3 requests each 15-20 seconds. The result is, at first, we see the image if through the ordinary camera (without any changes) for some seconds and then the screen goes black. That is why we can conclude that probably such a client-server solution is not fully suitable for the apps, where the result is supposed to be displayed instantaneously. And for this reason, we transform the pictures from the gallery later in out test cases, not directly from the camera. In addition, sometimes OutOfMemory error occurs while processing JSON on the device. It is important and can be evidence of data amount restrictions for handling on a device and, consequently, exchanging with the server.

Based on this experience, these are some deductions about which code should not or cannot be extracted:

- that modifies the external state of the application without returning the new value;

- that uses mobile platform-specific resources, such as UI or system services;
- that uses mobile platform-specific classes and stubs;
- that uses native code (or it needs to be recompiled for your server).

As for the native code, applications using Android NDK “will be more complicated, have reduced compatibility, have no access to framework APIs, and be harder to debug” [55]. Using native code does not necessarily result in performance improvement [56] and it may be sensible to refrain from using it.

Acquiring data from the system services such as getting a GPS signal or processing sensor updates is very power hungry [57] [58], but we cannot manage it in the cloud without the device's hardware. However, it is claimed that the new sensors introduces with Android 4.4 KitKat are more reserved concerning energy consumption [59].

4.2.3. Power measurements

As well as the other applications, we test AsciiCam in three modes: on a device, using the cloud and fast Internet connection, and using the cloud and slow Internet connection. Battery drain, CPU load and time spent are measured.

In addition, the tests are divided into three groups depending on the image size. Image size means resolution in this case, since the whole picture is never sent to the cloud, but only an array of its pixels.

Table 2 presents the results. The percentage of battery drain is a forecast of consumption for an hour which is based on the amount of energy eaten during the test.

	Local			Fast connection (D 6-12 Mbps/ U 7,7-12)			Slow connection (D 0.9-2.4 Mbps/ U 0,6-0,8)			
	Battery, %	CPU, %	Time, s	Battery, %	CPU, %	Time, s	Battery, %	CPU, %	Time, s	Data volume (out/in), Mb
AsciiCam Image										
Heavy image										5487.25 / 150.86
Average	22.5	33	32	26.3	45.4	116.8				
Standard deviation	2.54	2.74	5.39	4.18	4.37	62.95				
Medium image							13.69	18.95	800	2887.77 / 68.8
Average	18.4	29.8	26	28.1	41.9	36.7				
Standard deviation	2.94	5.2	5.29	0.51	1.08	3.79				
Light image										430.67 / 116.23
Average	18.4	25.2	17	24	32.3	20.7	14.5	37	230	
Standard deviation	2.94	3.03	1	1.38	2.23	1.53	0.28	1.15	7.55	

Table 2. AsciiCam Image power consumption.

In case of a large image combined with slow connection, the picture was not even received by a server in reasonable time, that is why the test falls out.

The case with medium resolution picture and slow connection also does not seem to be any beneficial as the performance (time) drops dramatically. For this reason, it was carried out only once.

Figure 15 and Figure 16 reveal the changing behavior of the app when running locally and remotely.

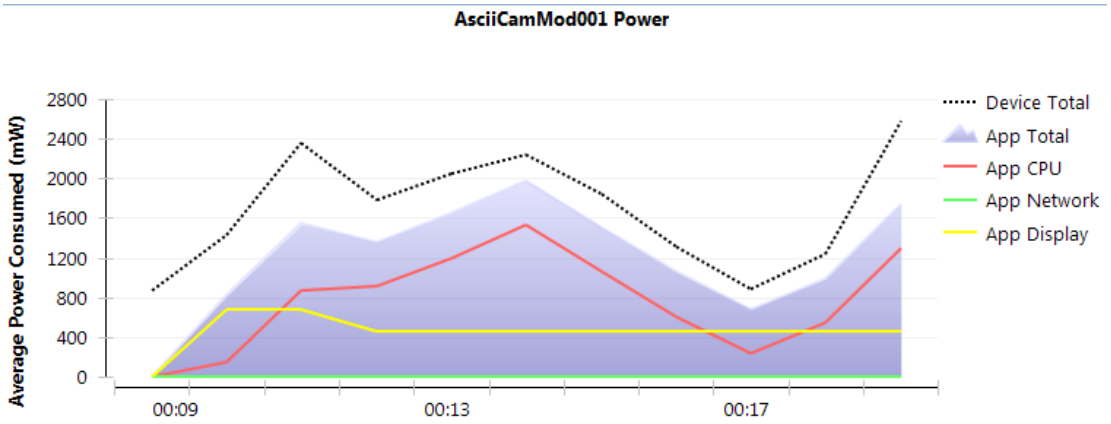

Figure 15. AsciiCam Image power consumption while running on a mobile device.

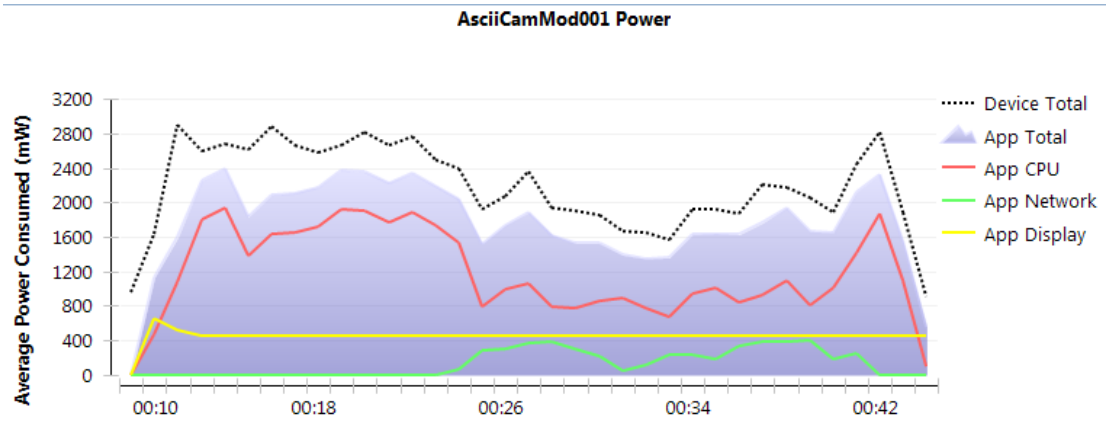

Figure 16. AsciiCam power consumption while using the cloud.

Taking the measurement outcomes into consideration, AsciiCam Image appears to be the application that most of the time will not benefit from the way it is now implemented. We do not say “will not benefit from our approach” for the following reasons:

- success (saving energy) depends on the amount of data that the app need to send and receive. Consequently, if the picture is small we could benefit;
- high battery drain in cases that use the cloud is caused by intense CPU load. The reason is writing and reading JSON with gson. This

means, we should adjust the way to transport data to its size. If there is a lot of what to send, it is useful to divide the data and transfer only those parts which are really processed (probably, some refactoring will be required) or send files (of course, if your method deals with them). Otherwise, it could be reasonable not to use a server for such tasks;

- the app just requires fast Internet connection. Otherwise, the performance (time) goes down significantly. If the connection is very slow, there is no point in performing the task remotely.

Figure 17 and Figure 18 display the CPU load in different modes.

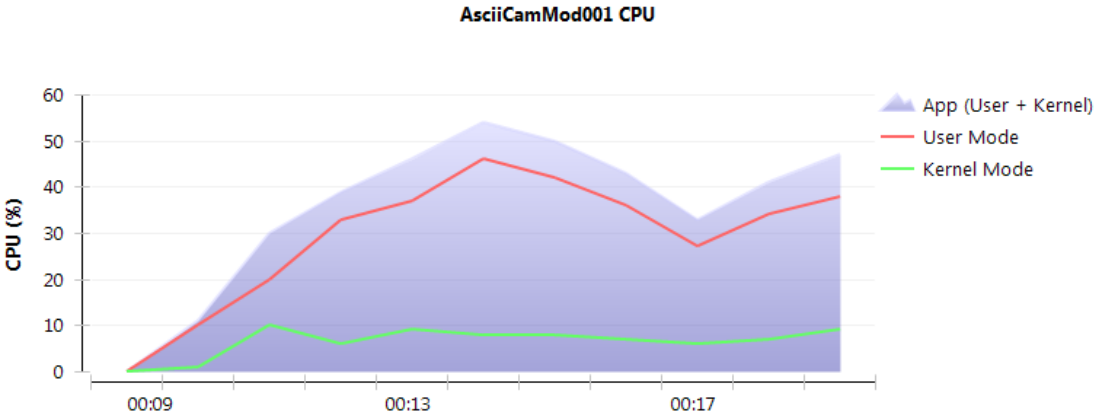

Figure 17. AsciiCam CPU load while running on a mobile device.

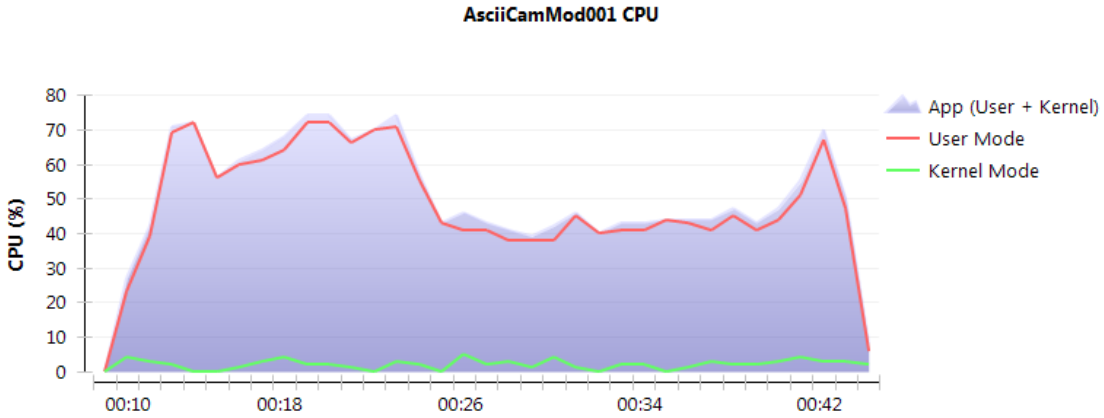

Figure 18. AsciiCam CPU load while using the cloud.

4.2.4. Result

The AsciiCam Image example has revealed some restrictions on the approach.

First of all, working with JSON may load the CPU greatly if the data to be transferred is big, for example, a very large array of values. Thereby, it is not possible to reach the key condition for reducing energy consumption – reducing CPU load. Consequently, the power was not

saved.

Moreover, a developer can face the issues that hamper the app transition to the cloud such as platform-specific libraries. In our case, it was android.graphics and its Bitmap class. The problem was solved by using a resembling Java class which can be managed on the server. But you can probably not find such for your case and then you have to write it by yourself or refrain from porting that piece of code. Another option which suits for native code issues is to find the source code, recompile it for your platform and then use, but it is sensible to consider the worthiness.

Last but not least, pay attention to the connection speed. If it is too slow, the performance may be unacceptably deteriorated. To prevent your application from such an unpleasant experience, provide a timeout for the remote task so that if it is not completed in some seconds run it locally.

4.3. AsciiCam Video

The AsciiCam Video application is a modification of AsciiCam Image written by us and makes the same but with video files, not images. Our goal while constructing this application was to make the device work harder and longer in comparison to AsciiCam Image.

4.3.1. Application architecture

To be able to work with video, it was necessary to give the app the access to the video files in the gallery.

When an mp4 file is picked, it is divided into frames. A frame is processed as a picture – that means, exactly as it was in AsciiCam Image. Direct after the transformation, a picture is written into a new mp4 file, which is going to be the output video.

Since the mobile devices are relatively strongly restricted in recourses, it is important not to keep intermediate values in memory. For example, we could firstly divide the whole video into frames putting them into an array and, after that, work them up. But in this way, we will get OutOfMemory error and no chance to finish the operation.

To decode and encode video, JCodec library is used.

4.3.2. Porting to the cloud

In contrast to the two previous examples, we do not use JSON in AsciiCam Video: a client and a server exchange video files (the second communication way described in a “Client-server HTTP communication” section in “Approach”). The rest on the communication method remains the same.

On the server, the type of the response body is defined and, according to it, a decision about that methods to execute is made.

This time, we faced the same problem as with AsciiCam Image before. AsciiCam works with the instances of Bitmap class on a device and with BufferedImage on the server. Because of that, we need to solve the problem with colors. Furthermore, we have to accommodate the frame's width and height and direction (vertical/horizontal). It was important to get the video of approximately the same size as a resulting video made on a device has.

It is not important for this thesis, how the problems were solved. But it is significant that they occur. So you should be ready that porting your application to the cloud can also appear to be a bit problematic.

4.3.3. Power measurements

As well as the other test applications, AsciiCam Video is tested locally, with slow Internet connection and with fast Internet connection. Video processing takes a lot of time even if the video is very short, which directly depends on the length of the video and its resolution. For this reason – and since we have already tried various data volumes – we use only a short video in our test cases.

The results of the tests are presented in Table 3.

	Local			Fast connection (D 6-12 Mbps/ U 7,7-12)			Slow connection (D 0.9-2.4 Mbps/ U 0,6-0,8)			
	Battery, %	CPU, %	Time, s	Battery, %	CPU, %	Time, s	Battery, %	CPU, %	Time, s	Data volume (out/in), Mb
AsciiCam Video										
Short video										2.73 / 30.26
Average	34.8	47.7	1260	16.6	38.5	224.7	16.3	40.7	809	
Standard deviation	3.2	5.3	129.62	2.75	0.79	53	0.87	1.52	188	

Table 3. AsciiCam Video power measurement.

According to these values, our approach is beneficial for this application. Both battery drain and time demand were improved.

When a device computes the results, its power consumption behavior looks like at Figure 19.

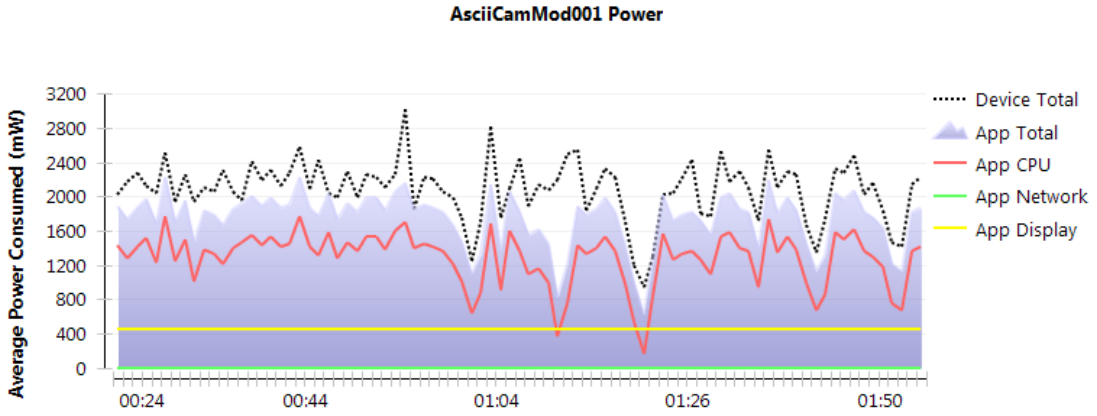

Figure 19. AsciiCam Video power consumption while running on a mobile device.

It is obvious, that the CPU is responsible for the most of energy consumed. When the cloud is used, the CPU is not so active, and WiFi module which also requires energy is not so power-hungry (Figure 20, Diagram 5, Diagram 6).

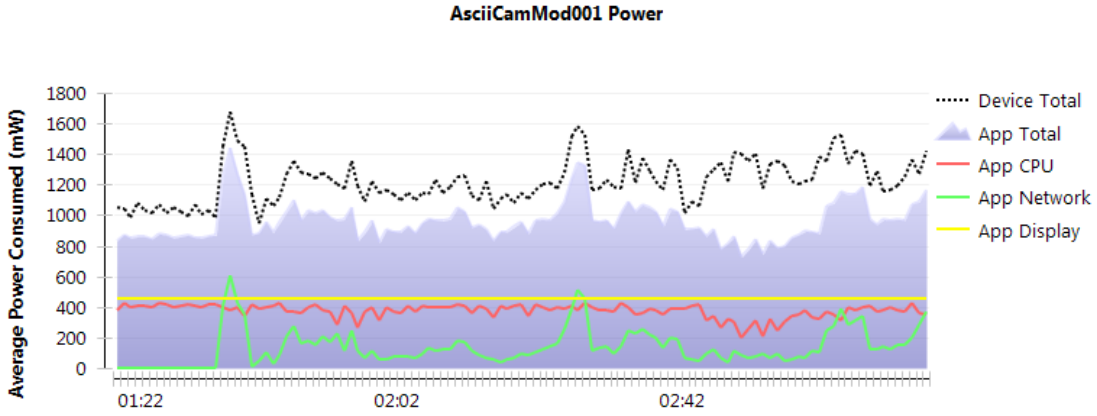

Figure 20. AsciiCam Video power consumption while using the cloud.

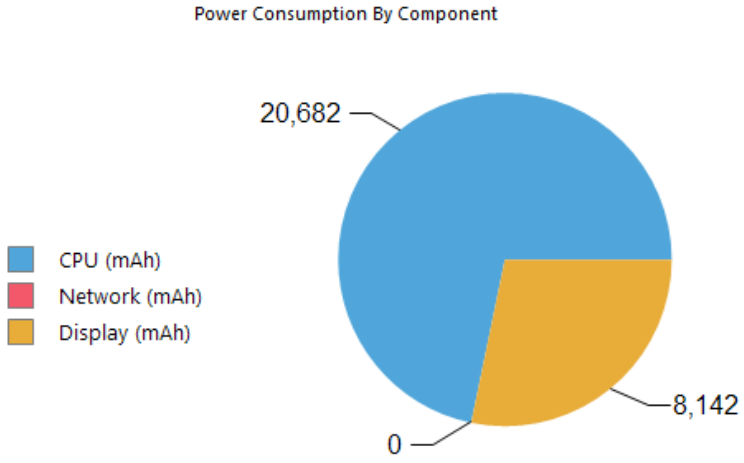

Diagram 5. AsciiCam Video: power consumption by component for 4 minutes 19 seconds while running on a mobile device.

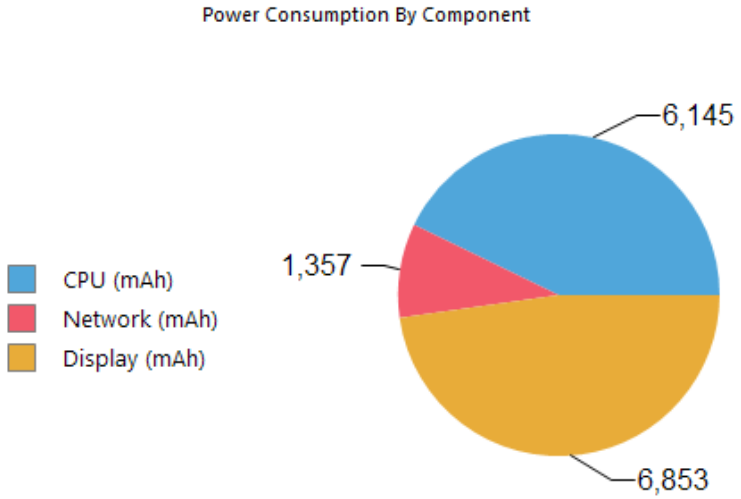

Diagram 6. AsciiCam Video: power consumption by component for 3 minutes 19 seconds while using the cloud.

The Diagram 5 and Diagram 6 are evidence of how the energy consumption changes – decreases – when we use the remote server to execute some parts of our code. Scenarios were the same for these two tests.

4.3.4. Result

In contrast to AsciiCam Image, both energy consumption and performance are improved when processing video files. What is the difference between these cases?

The reason is that writing and reading JSON is not used, but video files are sent to the server as they are. This does not load the CPU significantly, and consequently, although the amount of data is bigger than in the case with pictures, the power is saved.

Another tip for saving energy can be switching off the screen then the operation is in progress. This is not suitable for all the apps, but if a user knows that the task will anyway take long time and they can access the result later (as in our case with AsciiCam Video, where the operation itself is important), turning off the screen will save even more energy.

5. Generalizing approach

This section generalizes the approach to porting the parts of the functionality of a mobile application to the cloud in order to reduce the energy consumption. The steps are enumerated, the tips and challenges are summarized.

To run the mobile app partly remotely the next activities have to be completed:

- identifying suitable code blocks;
- adapting code;
- setting up client-server HTTP communication;
- porting code to the cloud.

Step 1: find suitable blocks of code. These should be the most power-hungry functionality such as:

- the code containing complicated calculations which load the CPU significantly and may take a lot of time;
- methods with a few lightweight parameters and returned valued and lots of sophisticated functionality inside;
- already existing AsyncTasks.

The way to find this code is static analysis. For more information, check chapter 3.1 “What to look for”.

Open issues: How to optimize the search for these code blocks? Is there any way to automate it, maybe develop a special tool or use an existing one effectively?

Step 2: adapting code. Make sure the chosen functionality is fit for running remotely:

- it is a distinct method;
- it returns a result value;
- it does not use Android-specific resources, such as UI or system services;
- it does not modify the external state of the application inside without returning the modified values;
- most likely, it does not use platform-specific classes, stubs or native code.

Try to extract the chosen methods so as they will look like on a remote server and run the app using these “new” methods. After that, what remains is HTTP communication.

Open issues: Probably, it is possible to extract the found code into separate methods automatically, like IDEs are able to extract and insert code. Then how to avoid mistakes and make sure we get what we want?

Step 3: implementing client-server HTTP communication. Two ways of exchanging data can be applied: JSON and files. For detailed information, check section 3.3 “Client-server HTTP communication”.

Open issues: Is it feasible to make the templates for such a communication, auto-generate or adjust them? Are there more efficient approaches to data exchange?

Step 4: porting code to the cloud. It means wring a server-side application using the code you have chosen at step 1. In addition, it has to include a class or classes to receive the HTTP requests, analyze them, call the required method and send back the result. If you need, you have to implement or make run native or some other helper code.

Basing on the experiments of this work, these are some deductions about which code should not or cannot be extracted:

- that modifies the external state of the application;
- that uses Android-specific resources, such as UI or system services;
- that uses platform specific classes and stubs;
- that uses native code (or it needs to be recompiled for your server).

Open issues: How to build templates for helper classes? How to construct a server-side application (partially) automatically? Is it possible and worth writing a framework for such server-side apps that will run the functionality of a mobile device?

You should also be careful with serializing and deserializing big amounts of data to and from JSON that can load the CPU significantly and lead to even more energy drain. When it is possible, send files as they are to the server. This means, you should adjust the way to transport data to its size.

Do not forget about the Internet connection speed. In case it is too slow, you risk to damage performance (e.g. increase time).

In conclusion, the main factor to improve energy consumption is to decrease CPU loading. That is why, find methods to achieve this goal.

6. Conclusion

This section reviews the results and deductions of the work.

At the beginning, in “Introduction” chapter, three hypotheses for the thesis were stated. They are:

- the energy needed for data transmission from device to a server and receiving back a result can be less than the energy consumed for executing the same functionality at a device;
- there can exist some “heavy-loaded” code blocks which can be extracted without hurting an application and executed remotely;
- performance can change to the worse as well as to the better.

Thereby, the key idea of this work is that running the parts of a mobile application functionality in the cloud can save mobile device energy.

Three applications were chosen to verify these hypotheses. They were adjusted to be able to execute the parts of their functionality remotely, according to the approach explained in section 3 “Approach” and summarized in chapter 5 “Generalizing approach”. A number of experiments and measurements were conducted which gave results to verify the statements above.

Applying the approach presented in this work and based on delegating the parts of mobile app’s functionality to the cloud makes it possible to save battery power on a mobile device.

Firstly, *applying approach saves energy* in many cases. This is achieved by reducing the CPU load, as a device now does not have to compute all the results by itself, but this function is delegated to the cloud. The amount of energy required to send data to and receive it from the server is usually less than the power needed to execute these methods on a device itself. But there can be exceptions. For example, if you have very big data to write to JSON strings and read it from them, then you are probably going to increase the device’s CPU use and therefore, energy will not be saved. Thus, you should consider *adjusting the way to transfer data to its amount and type*.

Secondly, each application accomplishes some functionality. It can be represented by *the code which requires a lot of resources*. These blocks of code can be extracted and ported to the cloud to be executed there. This process is not very difficult and can be divided into some particular steps (described in chapters 3 and 5), but a number of challenges can be faced while doing this. Some *code could be closely*

coupled with the mobile platform, so it may occur to be troublesome to run it on another one.

Thirdly, *the performance (time) of the app can change*. In the example applications, it was obvious that the task can be executed both faster and slower. It depends on its complexity, the amount of data to be exchanged between the device and the server, and the speed of internet connection. That is why you should test your application in order to decide whether the performance variations are acceptable for a user. It may be sensible to provide a timeout to run a task locally if there is no answer from the cloud.

To evolve the approach, the following issues can be can be solved:

- is it possible to automate:
  - the detection of heavy-loaded code blocks?
  - server-side application composition?
- is it feasible and beneficial to run the same platform in the cloud?
  - for example, use Android both on a device and on a server;
- how to provide lacking classes (those used on a mobile platform, but inaccessible from a server) with minimal costs and efforts?
- are there other ways to exchange data? Do they save more power?

In such a way, the approach described in this work lets improve energy consumption of mobile devices and add more complicated functionality to mobile applications.

7. References

[1] S. Taylor, "The New Mobile World Order: Perspectives on the Future of the Mobile Industry," September 2012. [Online]. Available: http://www.cisco.com/web/about/ac79/docs/sp/New-Mobile-World-Order.pdf. [Accessed 13 November 2013].

[2] S. Taylor, A. Young, N. Kumar and J. Macaulay, "Mobile Consumers Reach for the Clouds," July 2011. [Online]. Available: http://www.cisco.com/web/about/ac79/docs/sp/Mobile-Cloud_Consumer_IBSG.pdf. [Accessed 13 November 2013].

[3] J. Hruska, "The death of CPU scaling: From one core to many — and why we're still stuck," 1 February 2012. [Online]. Available: http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck/1. [Accessed 13 November 2013].

[4] A. Abdelmotalib and Z. Wu, "Power Consumption in Smartphones (Hardware Behaviourism)," May 2012. [Online]. Available: http://ijcsi.org/papers/IJCSI-9-3-3-161-164.pdf. [Accessed 13 November 2013].

[5] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," 2010. [Online]. Available: https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Miettinen.pdf.. [Accessed 13 November 2013].

[6] S. Taylor, A. Young and A. Noronha, "What Do Consumers Want from Wi-Fi? Insights from Cisco IBSG Consumer Research," May 2012. [Online]. Available: http://www.cisco.com/web/about/ac79/docs/sp/SP_Wi-Fi_Consumers.pdf. [Accessed 13 November 2013].

[7] J. Hruska, "The future of CPU scaling: Exploring options on the cutting edge," 28 February 2012. [Online]. Available: http://www.extremetech.com/extreme/120353-the-future-of-cpu-scaling-exploring-options-on-the-cutting-edge/1. [Accessed 13 November 2013].

[8] A. AppStorm, "Weekly Poll: How Often Do You Charge Your Phone?," [Online]. Available: http://android.appstorm.net/general/weekly-poll/weekly-poll-how-often-do-you-charge-your-phone/. [Accessed 26 December 2013].

[9] A. Shye, B. Scholbrock and G. Memik, "Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures," 2009. [Online]. Available: http://users.eecs.northwestern.edu/~bas667/papers/micro09-jamlogger.pdf. [Accessed 11 December 2013].

[10] T. H. Bui, "Android* Power Measurement Techniques," 6 January 2012. [Online]. Available: http://software.intel.com/en-us/articles/android-power-measurement-techniques. [Accessed 13 November 2013].

[11] S. Taylor, A. Young, N. Kumar and J. Macaulay, "A "Marriage Made in Heaven": Mobile Devices Meet the Mobile Cloud," October 2011. [Online]. Available: http://www.cisco.com/web/about/ac79/docs/sp/Mobile_Cloud_Device.pdf. [Accessed 13 November 2013].

[12] M. Hamdaqa and L. Tahvildari, "Cloud Computing Uncovered: A Research Landscape," 2012. [Online]. Available: http://www.stargroup.uwaterloo.ca/~mhamdaqa/publications/Cloud_Computing_Uncovered.pdf. [Accessed 13 November 2013].

[13] Google Inc., "Android, the world's most popular mobile platform," [Online]. Available: http://developer.android.com/about/index.html. [Accessed 13 November 2013].

[14] Apple Inc., "Siri," [Online]. Available: http://www.apple.com/ios/siri/. [Accessed 15 November 2013].

[15] A. Nusca, "How Apple's Siri really works," 3 November 2011. [Online]. Available: http://www.zdnet.com/blog/btl/how-apples-siri-really-works/62461. [Accessed 15 November 2013].

[16] D. Rawat, "iPhone 4S - How Does Siri Work?," 27 February 2012. [Online]. Available: http://www.webteacher.ws/2012/02/27/iphone-4s-how-does-siri-work/. [Accessed 15 November 2013].

[17] B. Johnson, "How Siri Works," January 2013. [Online]. Available: http://electronics.howstuffworks.com/gadgets/high-tech-gadgets/siri.htm. [Accessed 15 November 2013].

[18] W. Gerhardt, R. Medcalf, S. Taylor and A. Toouli, "Profiting from the Rise of Wi-Fi New, Innovative Business Models for Service Providers," March 2012. [Online]. Available: http://www.cisco.com/web/about/ac79/docs/sp/SP_Wi-Fi_PoV.pdf. [Accessed 13 November 2013].

[19] Oracle Java, [Online]. Available: http://www.java.com/en/.

[20] Red Hat, "OpenShift," [Online]. Available: https://www.openshift.com/.

[21] Vert.x. [Online]. Available: http://vertx.io/.

[22] Google-gson. [Online]. Available: https://code.google.com/p/google-gson/.

[23] D. Bornstein, "Google I/O 2008 - Dalvik Virtual Machine Internals," [Online]. Available: http://www.youtube.com/watch?v=ptjedOZEXPM. [Accessed 26 December 2013].

[24] M. Gargenta, "Learning Android," O'Reilly, 2011, p. 2.

[25] D. Bornstein, "Dalvik JIT," [Online]. Available: http://android-developers.blogspot.de/2010/05/dalvik-jit.html. [Accessed 25 December 2013].

[26] Google Inc., "Get the Android SDK," [Online]. Available: http://developer.android.com/sdk/index.html.

[27] Google Inc., "Application fundamentals," [Online]. Available: http://developer.android.com/guide/components/fundamentals.html. [Accessed 13 November 2013].

[28] Google Inc., "App structure," [Online]. Available: http://developer.android.com/design/patterns/app-structure.html. [Accessed 28 November 2013].

[29] Google Inc., "Activities," [Online]. Available: http://developer.android.com/guide/components/activities.html. [Accessed 28 November 2013].

[30] Google Inc., "Processes and Threads," [Online]. Available: http://developer.android.com/guide/components/processes-and-threads.html. [Accessed 29 November 2013].

[31] Google Inc., "AsyncTask," [Online]. Available: http://developer.android.com/reference/android/os/AsyncTask.html. [Accessed 29 November 2013].

[32] M. Gargenta, "Learning Android," O'Reilly, 2011, p. 11.

[33] The Android Open Source Project, "Dalvik Porting Guide," [Online]. Available: https://android.googlesource.com/platform/dalvik/+/master/docs/porting-guide.html. [Accessed 25 December 2013].

[34] Android-scripting. [Online]. Available: https://code.google.com/p/android-scripting/. [Accessed 25 December 2013].

[35] C. Neukirchen, "Programming for Android with Scala," 14 April 2009. [Online]. Available: http://chneukirchen.org/blog/archive/2009/04/programming-for-android-with-scala.html. [Accessed 26 Deccember 2013].

[36] Little Eye Labs, "Little Eye," [Online]. Available: http://www.littleeye.co.

[37] Little Eye Labs, "How Little Eye Measures Power Consumption," [Online]. Available: http://www.littleeye.co/blog/2013/07/30/how-little-eye-measures-power-consumption/. [Accessed 22 January 2014].

[38] Google Inc., "Android Debug Bridge," [Online]. Available: http://developer.android.com/tools/help/adb.html.

[39] PowerTutor, [Online]. Available: http://ziyang.eecs.umich.edu/projects/powertutor/.

[40] M. Schröder, "Erfassung des Energieverbrauchs von Android Apps," Oldenburg, Germany, 2013.

[41] Yoctopuce, "Yocto-Amp," [Online]. Available: http://www.yoctopuce.com/EN/products/usb-sensors/yocto-amp. [Accessed 7 February 2013].

[42] Gradleware Inc., "Gradle," [Online]. Available: http://www.gradle.org/.

[43] JCodec, [Online]. Available: http://jcodec.org/.

[44] Xuggler, [Online]. Available: http://www.xuggle.com/xuggler.

[45] E. Gamma, R. Helm, E. R. Johnson and J. Vlissides, "Design Patterns. Elements of Reusable Object-Oriented Software," Pearson Education, 1994, pp. 233-245.

[46] SourceMaking, "Proxy Design Pattern," [Online]. Available: http://sourcemaking.com/design_patterns/proxy. [Accessed 5 December 2013].

[47] E. Gamma, R. Helm, E. R. Johnson and J. Vlissides, "Design Patterns. Elements of Reusable Object-Oriented Software," Pearson Education, 1994, pp. 263-273.

[48] SourceMaking, "Command Design Pattern," [Online]. Available: http://sourcemaking.com/design_patterns/command. [Accessed 5 December 2013].

[49] I. Ivanov, A. Saksonov, E. Malutin and F. Atyakshin, "finite-element-technique," [Online]. Available: https://code.google.com/p/finite-element-technique/. [Accessed 9 December 2013].

[50] "Finite element method," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Finite_element_method. [Accessed 16 January 2014].

[51] "Dozing Cat Software [Accessed 27 November 2013," [Online]. Available: http://www.dozingcatsoftware.com/. [Accessed 27 November 2013].

[52] Google Play, "AsciiCam," [Online]. Available: https://play.google.com/store/apps/details?id=com.dozingcatsoftware.asciicam. [Accessed 27 November 2013].

[53] "Apache License, Version 2.0," [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0.html. [Accessed 27 November 2013].

[54] Nenninger, Brian, bnenning@gmail.com, [Online]. [Accessed 27 November 2013].

[55] D. Turner, "Introducing Android 1.5 NDK, Release 1," [Online]. Available: http://android-developers.blogspot.de/2009/06/introducing-android-15-ndk-release-1.html. [Accessed 7 December 2013].

[56] Google Inc., "Android NDK," [Online]. Available: http://developer.android.com/tools/sdk/ndk/index.html. [Accessed 08 December 2013].

[57] M. Gargenta, "Learning Android," O'Reilly, 2011, p. 190.

[58] "SensorManager," [Online]. Available: http://developer.android.com/reference/android/hardware/SensorManager.html. [Accessed 12 January 2014].

[59] Android Developers, "Android 4.4 KitKat and Updated Developer Tools," [Online]. Available: http://android-developers.blogspot.de/2013/10/android-44-kitkat-and-updated-developer.html. [Accessed 13 January 2014].

[60] M. Gargenta, "Learning Android," O'Reilly, 2011, p. 10.

Appendix

CD

A CD provided with this master's thesis contains the following data:

- PDF version of the thesis;
- FiniteElementCalculator source code;
- AsciiCam source code;
- screenshots, diagrams, tables.