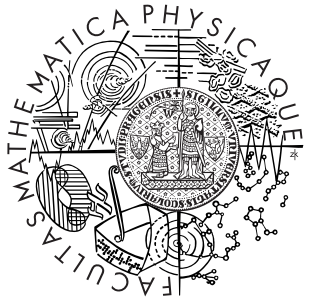
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Jakub Kinšt

DEECo Cloudlets Exploratory Study

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University

Supervisor of the master thesis: Doc. RNDr. Tomáš Bureš, PhD.

Study program: Informatics

Specialization: Software Systems

Prague 2015

I would like to thank my supervisor, Tomáš Bureš, for numerous helpful advices and valuable comments regarding the design and implementation. Also, I would like to express my gratitude to my partner, Eva Kufnerová, and the rest of my family for endless support during my work on this master thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague date signature

Název práce: TODO: name in czech
Autor: Bc. Jakub Kinšt
Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: Doc. RNDr. Tomáš Bureš, PhD.

Abstrakt: TODO: abstract
Klíčová slova: TODO: keywords

Title: DEECo Cloudlets Exploratory Study
Author: Bc. Jakub Kinšt
Department / Institute: Department of Distributed and Dependable Systems
Supervisor of the master thesis: Doc. RNDr. Tomáš Bureš, PhD., Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University

Abstract: TODO: abstract
Keywords: *TODO: keywords*

Contents

Chapter 1

Introduction

1.1 Motivation

With growing usage of mobile devices such as smartphones, tablets and wearables, users tend to use these devices to do things they used to do on desktop computers or laptops. Unlike desktop computers or laptops, these devices are usually not connected to a power source when their users are using them. The computation power of these devices is growing much more rapidly than the capacities of used batteries. With hardware being more and more efficient, the battery life of such devices is relatively stable for past few years, but still, it is very limited. This means that when a developer is designing a mobile application, it is essential to always think of mobile-specific resources that the application uses and design the application to be as efficient as possible.

When developing applications performing resource-intensive computation tasks such as image recognizing, sound recognizing or even communicating with a remote API, there is one technique that can be particularly useful to save at least some of the resources - *offloading*. Offloading resource-intense tasks to another machine can sometimes significantly reduce resource usage when used wisely.

Mobile devices also usually form a very dynamic large-scale environment. As their users are leaving their homes, entering workplaces, using public WiFi networks during transportation, in coffee shops or restaurants, their smartphones or tablets are entering and leaving different networks very often. This dynamic environment tend to be very hard to control. With nodes appearing and disappearing on regular basis, it is necessary to design a system without a dedicated central authority to control such environment.

Recently introduced Ensemble-Based Component Systems (EBCS) can provide an optimal solution for this kind of problem since they are specifically tailored for dynamic, large-scale and decentralized environment. It is worth

examining if EBCS can be used as a control layer in the offloading scenario.

1.2 Thesis goals

The primary goal of this master thesis is to explore possibilities of using the DEECo component model to achieve cloudlet offloading of parts of a mobile application. This means executing parts of application computation on a different machine within a local network.

The most important goal is to design and implement a reference architecture which would bring offloading capabilities to a regular mobile application for the Android platform. This architecture should use the principles proposed in the DEECo component model to create a control layer for the offloading mechanism. This goal also includes choosing as universal way as possible to break the application into separate components that are prepared to be offloaded - run on a different machine. Also a mechanism for deciding which offloadable component will run on which machine in the network is to be designed.

In order to implement the reference architecture, few challenges have to be addressed first. Particularly it is necessary to port the existing Java implementation of DEECo component model to work on the Android platform to be able to run its runtime on mobile devices. The existing library does not support communication between different nodes in "real-life" computer networks. Therefore the next step is to develop an extension to the library providing support for a communication within a local network.

Lastly, a demonstration application is to be developed to bring the whole idea to the "real world" and show the offloading mechanism in action. The demo application should provide a reasonable functionality that can benefit from being offloaded off the mobile device itself and potentially save some of its battery life or another limited resource. The developed application should be evaluated in terms of amount of saved resources while offloading is enabled. The application (together with a set of instruction) should serve as an example of the process of adding offloadable capabilities to any other Android application as easily as possible.

Chapter 2

Background

2.1 DEECo component model

2.1.1 Ensemble-based component systems

With recent increasing possibilities provided by the evolution in the field of mobile devices and their improving connectivity, new ways of addressing social and environmental challenges (ambient assisted living, smart city infrastructures, emergency coordination, environmental monitoring) are emerging. Solutions are achieved by building large-scale Resilient Distributed Systems (RDS) that respond to and influence activities in the real world. "As RDS have to cope with very dynamic and open-ended environments, they exhibit a high degree of adaptivity and autonomicity."[23]

The dynamic and autonomic nature of RDS causes issues with scalability when using traditional component architectures and models. Ensemble-based component systems (EBCS) were introduced to overcome this problem in [23]. The key feature of EBCS is the different composition of components. Instead of explicit component architecture, components are implicitly formed into so-called *ensembles* based on declared predicates which makes the composition of the components very dynamic and adaptive.

Finally, the DEECo (Distributed Emergent Ensembles of Components) component model was proposed "to refine the principles of EBCS into a systematic approach for building software for RDS"[23]. DEECo is an instance of EBCS which comes with a framework for building applications benefiting from EBCS features described earlier.

2.1.2 Running example

For further explaining of DEECo principles and rules in this section, a running example is introduced to help the reader to understand. An example provided in [23] is very useful for highlighting the key features of EBCSs. It is based on the electrical vehicle navigation case study.

The case study involves e-vehicles and their navigation around a city. Each driver of one of the e-vehicles has his own plan of stops (POIs) in the form of a calendar. The vehicles can only park in special parking places equipped with a charger clustered in parking lot/charging stations (PLCS). They are also able to constantly monitor their position, energy consumption and battery level. The vehicles can communicate with each other as well as with the parking/charging stations to be able to plan/reserve a parking place. The objective of the case study is to coordinate journey planning with constrains coming from the parking/charging strategy.

The important factor is that no central coordination authority is assumed in this case study. It involves highly dynamic environment of multiple nodes. The architecture of those nodes changes constantly in time based on the current position of e-vehicles. These are exactly the challenges targeted by EBCS.

2.1.3 Basic principles

DEECo is based on two main concepts - *component* and *ensemble*. According to [23], a component is an independent, self-sustained unit of development, deployment and computation. An ensemble, on the other hand, is a dynamic binding mechanism linking a set of components together and providing a communication channel between them. Actually, one of the most important principle of DEECo is that ensembles provide the only way of communication between components. On top of components and ensembles, DEECo defines a runtime, which provides necessary management services for both. The execution of the components is fully isolated and uses only component's belief - a partial view on the whole system of other components which is automatically cloned by the runtime to make it available locally.

2.1.3.1 Component

A single component consists of *knowledge* and *processes*. Knowledge reflects the state of the component and it is organized in a hierarchical data structure mapping knowledge identifiers to values. It is exposed through an implicit set of interfaces which represent a partial views of the knowledge.

As far as the processes are concerned, they are basically tasks, which are able

```
1  interface AvailabilityAggregator:
2      calendar, availabilities
3
4  interface AvailabilityAwareParkingLot:
5      position, availability
6
7  component Vehicle features AvailabilityAggregator:
8      knowledge:
9          batteryLevel = 90%,
10         position = GPS(...),
11         calendar = [ POI(WORKPLACE, 9AM-1PM), POI(MALL, 2PM-3PM)
               , ... ],
12         availabilities = [ ],
13         plan = {
14             route = ROUTE(...),
15             isFeasible = TRUE
16         }
17     process computePlan:
18         in plan.isFeasible, in availabilities, in calendar,
               inout plan.route
19         function:
20             if (!plan.isFeasible)
21                 plan.route <- Planner.computePlan(calendar,
                       availabilities)
22         scheduling: triggered( changed(plan.isFeasible) OR
               changed(availabilities) )
23     process checkPlanFeasibility:
24         in plan.route, in batteryLevel, in position, out plan.
               isFeasible
25         function:
26             plan.isFeasible <- Planner.isFeasible(plan.route,
                   batteryLevel, position)
27         scheduling: periodic( 5000ms )
28
29 component PLCS features AvailabilityAwareParkingLot:
30     knowledge:
31         position = GPS(...),
32         availability = ...
33     process observeAvailability:
34         out availability
35         function:
36             availability <- Sensors.getCurrentAvailability()
37         scheduling: periodic( 2000ms )
```

Figure 2.1: *Example of DEECo component definitions in a DSL (Source: [23])*

```
1 ensemble UpdateAvailabilityInformation:
2     coordinator: AvailabilityAggregator
3     member: AvailabilityAwareParkingLot
4     membership:
5         if poi in coordinator.calendar:
6             distance(member.position, poi.position) <= TRESHOLD
                  && isAvailable(poi, member.availability)
7     knowledge exchange:
8         coordinator.availabilities <- { (m.id, m.availability) |
              m in members }
9     scheduling: periodic( 5000ms )
```

Figure 2.2: *Example of DEECo ensemble definition in a DSL (Source: [23])*

to manipulate the component's knowledge. They are represented by a function which has a set of input and output knowledge fields. The function is called by the DEECo runtime framework and the process of reading the input knowledge, running the function and writing the output knowledge is atomic. A component should never communicate with other components directly via processes. It should only read and/or manipulate knowledge of its own. The process can be either triggered by a certain event, or it can by performed periodically by the runtime. See Figure 2.1 for an example of component definitions.

2.1.3.2 Ensemble

An ensemble serves as a binding mechanism between multiple components. It provides a way of communication between them. One of the components involved in an ensemble has the role of *coordinator* and others are just *members*. The involvement of components in an ensemble is defined by ensemble's *membership* function. The membership function consists of the definition of the *coordinator interface*, the *member interface* and the *membership predicate* which is evaluated by the DEECo runtime to determine which two components represent a coordinator-member pair.

The communication between the components within an ensemble is provided through the *knowledge exchange* function. Actually, the function provides interaction between the coordinator and all other members only, so all communication must go through the coordinator. Communication is realized by the ability to read and/or write a knowledge of both the coordinator and the member within the coordinator and member interfaces. The knowledge exchange operation can be either triggered by a certain event, or it can by performed periodically by the runtime. See Figure 2.2 for an example of an ensemble definition.

2.1.4 jDEECo library

To be able to actually use the DEECo component model for development of RDS, a framework called jDEECo[26] has been developed. It is an implementation of the model, which provides a way to deploy and run real DEECo-based applications written in Java language. A jDEECo component has a form of a Java class marked by the `@Component` annotation. its knowledge is represented by public non-static fields and the processes are defined by public static methods marked by the `@Process` annotation. its input and output knowledge fields are marked by the `@In`, `@Out` or `@InOut` annotations.

Similarly, a jDEECo ensemble is defined by annotating a standard Java class with the `@Ensemble` annotation. The membership and knowledge exchange functions are defined by public static methods marked by `@Membership` or `@KnowledgeExchange` respectively. its input and output knowledge fields should be marked in the same manner as the component's process functions. An example of jDEECo component definition is provided in Figure 2.3.

jDEECo runtime framework provides functionality for registering components and ensemble definitions both before and during runtime. The runtime is very customizable and easily extendable thanks to extensive granularity of the design.

2.2 Concept of Cloudlets

A *cloudlet* is an emerging architectural element arising from the combination of mobile computing and cloud computing. [6, 38]It is a middle layer of recently proposed 3-layer hierarchy, which consists of a mobile device, cloudlet and cloud as we know it. The goal of the cloudlet proposal is to "bring the cloud closer" to the mobile device. According to [6], a cloudlet features four key attributes:

- **soft state only** - A cloudlet should not have any hard state, but it can contain a cached state from the cloud for instance. The lack of a hard state makes cloudlets self-managing and easy to deploy.
- **powerful, well-connected and safe** - A cloudlet should feature sufficient computation power compared to mobile devices. It should have an unlimited power source (connected to a power outlet) and very good and stable connectivity to the cloud.
- **close at hand** - It should be close to the mobile device in terms of latency and high bandwidth. Usually this means it should be connected on the same local network via Wi-Fi for instance.

```
1  @Component
2  public class Vehicle {
3      public List<CalendarEvent> calendar;
4      public Plan plan;
5      public EnergyLevel batteryLevel;
6      public Map<ID, Availability> availabilities;
7      public Position position;
8
9      public Vehicle() {
10         // initialize the initial knowledge structure reflected
               by the class fields
11     }
12
13     @Process
14     public static void computePlan(
15         @In("plan.isFeasible") @Triggered Boolean isPlanFeasible
               ,
16         @In("availabilities") @Triggered Map<...>availabilities,
17         @In("calendar") List<CalendarEvent> calendar,
18         @InOut("plan.route") Route plannedRoute
19     ) {
20         // re-compute the vehicle's plan if its infeasible
21     }
22
23     @Process
24     @PeriodicScheduling(5000)
25     public static void checkPlanFeasibility(
26         @In("plan.route") Route plannedRoute,
27         @In("batteryLevel") EnergyLevel batteryLevel,
28         @In("position") Position position,
29         @Out("plan.isFeasible") OutWrapper<Boolean>
               isPlanFeasible
30     ) {
31         // determine feasibility of the plan
32     }
33     ...
34 }
```

Figure 2.3: *Example of jDEECo component (Source: [23])*

- **builds on standard cloud technology** - It should be similar to classic cloud infrastructures.

The concept of cloudlets was designed for mobile application computation offloading scenario.[38] According to this paper, cloudlets are the technology bringing a new type of mobile applications which are resource-intense but latency-sensitive at the same time. These applications are expected to emerge in the near future.

2.3 Computation offloading

Computation offloading generally means delegating certain computing tasks to another node - usually a cloud, cluster or grid. The goal of computation offloading is either to save system resources used on a device or to access higher computation performance unavailable when performing the task on its own.

In the world of smartphones, tablets or wearables, developers are facing new challenges and problems such as limited battery life and lower performance. When developing a mobile application for such devices, transferring resource-intensive tasks to the cloud can significantly reduce battery usage and bring better user experience to the user.[34]

2.3.1 Use cases

Computation offloading is starting to be widely used strategy in certain areas of mobile applications. For instance, each major mobile platform now has its own "voice assistant" (Android: *Google Now*, iOS: *Siri*, Windows Phone: *Cortana*) which is able to listen for whatever question the user may possibly have and respond instantly with an answer.[37] Let's focus on a voice search for a contact name present on the mobile device for instance. This process consists of two main steps: (i) speech recognition which converts a recording of human voice to a search query and (ii) actual search for the contact based on its name. If the process of voice recognition was performed locally on the mobile device itself, the response would most definitely be longer than instant. That is because voice recognition is highly non-trivial task and mobile devices usually don't feature such computation power and storage to provide response in a reasonable time. Here is what actually happens (after simplification): (i) the mobile application preprocesses the voice recording provided by the user, (ii) sends the preprocessed recording to the cloud and (iii) receives a string representation of the search query which was probably computed at a datacenter with enormous computation power.[37] The search query can then be used for a simple local search task.

However, computation offloading is definitely not a silver bullet. Since network communication used to reach the offloading target is considered resource-intensive as well, one has to find an optimal balance when designing application featuring computation offloading. It is necessary to carefully inspect if the nature of a task is suitable for offloading. Generally, most resources can be spared by offloading long-running tasks or by batching (if possible) multiple operations and offloading it at once. A very good examples of such long-running tasks are those involving some kind of recognition - voice recognition, image recognition, song-recognition, augmented reality, video recognition, biometric scanning and many more.[27, 36, 33, 34, 37] Some literature ([36]) even explored offloading of network communication with application's backend API by smart batching and other optimizations. Even this kind of usage was proven to be able to save device's battery life when used wisely.

2.4 Android platform fundamentals

Android[5] is an operating system by Google for mobile devices such as smartphones, tablet computers, smartwatches and other. It is based on the Linux kernel and the apps are written primarily in Java using the Android SDK[3].

Android does not use Java Virtual Machine as one would expect, but instead, it uses *Dalvik* with just-in-time (JIT) compilation as a virtual machine to run the applications. Since Android 5.0 Lollipop, *Dalvik* was replaced by *ART*, which introduced ahead-of-time (AOT) compilation. The compilation is performed at the time of the installation of an application.

Also the APIs of both platforms differ. Most of the standard API is the same, but many differences are present. For instance the classes related to *AWT* or *Swing* are obviously not present, since Android uses its own user interface implementation. Also vendor packages like `com.sun.*` are unavailable. On the other hand, the Android API includes other classes specific to the platform.

These differences are usually sources of issues when developing a library (or any code at all), which should work on both Java SE and Android. Therefore the developer should have these differences in mind at all times during development.

2.4.1 Application development

2.4.1.1 AndroidManifest.xml file

Every Android application must contain a special XML file which describes the application for the operating system. It contains information about application package - universal application identifier, its name, icon, permission requests,

hardware requirements and more. It also contains definitions of main application building blocks such as *Activities*, *Services*, *BroadcastReceivers* and others.

An example of *AndroidManifest.xml* file can bee observed in Figure 2.4.

2.4.1.2 Permissions

On Android platform, any operation that could be potentially considered as dangerous for the user must be declared in the *AndroidManifest.xml* file statically. These operations include access to the Internet, reading or writing to the external storage, accessing the contact list, sending a text message, placing a call and many others.

When installing an application, all requested permissions are presented to the user for review and acceptation. Since then, the application is granted all of the permissions and is able to perform desired operations falling under those permissions. See the example of the *AndroidManifest.xml* file in Figure 2.4 for examples of requesting permissions.

2.4.1.3 Activities

An activity is probably the most important building block of every single Android application. It is a visual component represented by a "window" (usually full-screen if not specified otherwise) and it is dedicated to display the user interface (UI). One or more activities can be tagged as launchable from the list of applications on the device (see the example of the *AndroidManifest.xml* file above) and others can be started manually using *Intents*.

An activity is represented by a Java class extending `Activity`, which describes its functionality. The structure (layout) of the user interface is defined in separate reusable XML files using proprietary markup.

Android applications are started differently than standard Java applications. There is no `main()` method which is run automatically. Instead, when the activity is started, special callback methods are called according to standard activity lifecycle. These methods are meant to be overridden to add functionality.

2.4.1.4 Services

Services are non-visual components providing a way to run arbitrary code in the background without any user interface. Their lifecycle is similar to activity lifecycle displayed in figure2.5. A service can be started via *Intents* but it can be also set up to be invoked regularly by an *AlarmManager* or in reaction to certain event by *BroadcastReceivers* for instance.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/
       android"
3    xmlns:tools="http://schemas.android.com/tools"
4    package="cz.kinst.jakub.example.app">
5
6    <uses-permission android:name="android.permission.INTERNET"/>
7    <uses-permission android:name="android.permission.GET_ACCOUNTS
         "/>
8    <uses-permission android:name="android.permission.
         READ_EXTERNAL_STORAGE"/>
9    <uses-permission android:name="android.permission.
         WRITE_EXTERNAL_STORAGE"/>
10
11   <uses-feature android:name="android.hardware.camera"
         android:required="true"/>
12
13   <application android:name=".ExampleApplication"
14     android:icon="@drawable/ic_launcher"
15     android:label="@string/app_name"
16     android:theme="@style/AsparagusTheme">
17
18     <activity android:name=".ui.activities.MainActivity"
19       android:icon="@drawable/ic_launcher"
20       android:logo="@drawable/logo_font"
21       android:label="@string/app_name">
22       <!-- This activity is the default activity listed in the
             app drawer on the device. It provides an entry point
             for the user. -->
23       <intent-filter>
24         <action android:name="android.intent.action.MAIN"/>
25         <category android:name="android.intent.category.LAUNCHER
               "/>
26       </intent-filter>
27     </activity>
28     <activity android:name=".ui.activities.SettingsActivity"
29       android:icon="@drawable/ic_asparagus"
30       android:label="@string/settings">
31     </activity>
32     <receiver android:name=".sync.GcmBroadcastReceiver"
33       android:permission="com.google.android.c2dm.permission.
           SEND">
34       <intent-filter>
35         <action android:name="com.google.android.c2dm.intent.
             RECEIVE"/>
36         <category android:name="cz.kinst.jakub.example.app"/>
37       </intent-filter>
38     </receiver>
39     <service android:name=".sync.SyncAdapterService"
40       android:exported="true">
41       <intent-filter>
42         <action android:name="android.content.SyncAdapter"/>
43       </intent-filter>
44       <meta-data android:name="android.content.SyncAdapter"
45         android:resource="@xml/sync_adapter"/>
46     </service>
47   </application>
48 </manifest>
```

Figure 2.4: *Example of the AndroidManifest.xml file*

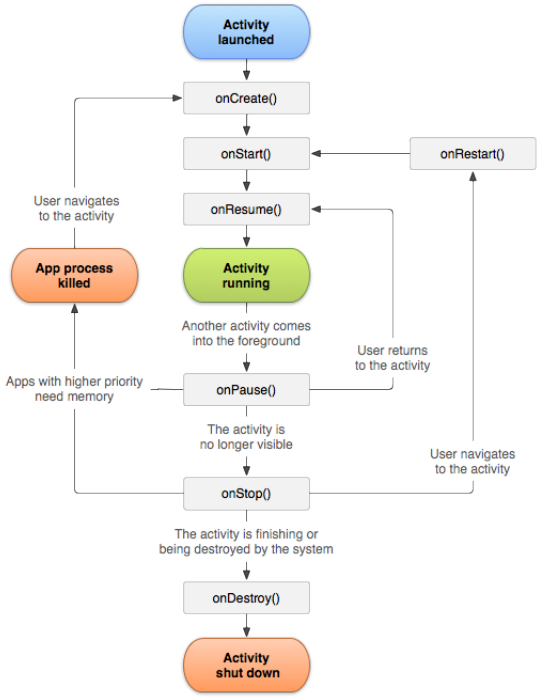


Figure 2.5: *Android activity lifecycle diagram. (Source:* `http://developer.android.com/reference/android/app/Activity.html` *)*

2.4.1.5 Development, building and running applications

The recommended development environment for developing Android applications is currently the *Android Studio*, which is based on *IntelliJ IDEA* from JetBrains. The apps should be built using *Gradle[4]* as a build system. Gradle is relatively new and its Android support is still in development.

Android Studio together with *Gradle* takes care of building and packaging the application. The output is an *.apk* file containing the entire application. This file can be transferred to the device and installed simply by launching it. The developer can also run or debug the application directly on connected device from the IDE using the *ADB* tool provided by the SDK.

2.5 Gradle build system

Gradle uses Groovy-based domain specific language (DSL) for the project build configuration instead of traditional forms like XML used in Apache Ant or Maven. It is also designed for use in a multi-project environment so it fits our needs perfectly.

Sub-projects are called modules in Gradle. Each module is configured via a single *build.gradle* file residing in the module's root directory. The configuration starts with the declaration of used plugin, determining the form of the module. In our case, the form can be one of following:

Java application: `apply plugin: 'application'`

Java library: `apply plugin: 'java'`

Android application: `apply plugin: 'com.android.application'`

Android library: `apply plugin: 'com.android.library'`

Depending on the plugin used, a plugin-specific configuration may follow. Another important part of the configuration file is the explicit declaration of module's dependencies. A dependency can be either one of the sibling modules, a JAR file containing a compiled Java code, or simply a valid identification of a package, which is automatically downloaded from any Maven repository (such as Maven Central Repository, jCenter, or a local repository).

The developed framework as well as the demonstration applications were developed in the *Android Studio*, which is based on *IntelliJ IDEA*. It features built-in support for Gradle build system and is able to build and run both Android Application modules and Java modules.

Chapter 3

Analysis

The main goal of this thesis is to design and implement a reference architecture for offloading mobile applications with DEECo as the control layer. In this section, this goal is analyzed and potential issues and challenges are identified. Also, since some important design choices have to be made throughout the design and development, possible solutions are discussed.

3.1 Towards offloading controlled by DEECo

For designing a reference architecture for computation offloading driven by DEECo, an idea proposed in [21] was used as a rough base. This position paper introduces a possibility of using EBCS as a control layer for computation offloading of mobile applications. Particularly, it focuses on the management of ad-hoc cloud (cloudlet) systems in such dynamic environment without any need for central authority and with focus on scalability and robustness of the solution. Since the proposed architecture covers only part of the problematics we are focused on, we are left with many open design issues, which are to be resolved or dealt with. Also, the implementation of the architecture will bring some challenges as well.

The paper provides a running example which contains a user with a tablet computer traveling in a train or a bus, who wants to do productive work during the transportation. When the user entered the train/bus, the tablet automatically registers there is an offload server available connected to the same Wi-Fi network. In order to save battery, the tablet offloads most resource-intensive tasks to that server. When the train/bus reaches its destination, the server informs the tablet that it will soon be unavailable so tasks will start to move back to local execution. However, the tablet then registers another offload server. This time, it is provided by the train/bus terminal authority. Again, the tablet can move some of its tasks to this node.

Authors of the paper then generalize the idea assuming a mobile device **M** and two stationary devices **S** and **T** (offload servers). **M** runs application **A** which can be divided to **Af** and **Ab** - a frontend responsible for user interaction and backend responsible for computing resource-intensive tasks.

After the generalization, the scenario from the running example can be summarized to following:

1. **M** discovers **S**
2. **M** assesses that it is a better alternative to offload **Ab** to **S** in regards to battery usage
3. **S** notifies **M** that it will be soon unavailable
4. **M** discovers **T**
5. **M** assesses that it is a better alternative to offload **Ab** to **T** in regards to battery usage

The challenge is "in predicting which deployment scenario will—in the context of ad-hoc cloud—deliver the expected user experience".[21] The paper also assumes that each offloadable part will have certain performance model, which provides a rough estimate of user experience for each deployment plan. Parts of the application can then be deployed dynamically based on the predicted user experience when running on different nodes.

3.1.1 Proposed adaptation architecture

Architecture proposed in the paper is designed to form a separate control layer mirroring the architecture of the adapted application in order to separate concerns as much as possible. The architecture consists of a set of components and a set of ensembles linking the components together.

Planner component Each adapted application is represented by one Planner component. This component is responsible for deciding a deployment plan based on provided alternatives for the deployment. It basically means comparing all possible scenarios of backend deployment and choosing which backend component will run on which device. The input of this operation is NFP-related data (*NFPData*) containing measured estimate of potential performance of particular backend deployment.

Monitor component Each offloadable part of the application (backend application component in our case) is mirrored by a set of Monitor components

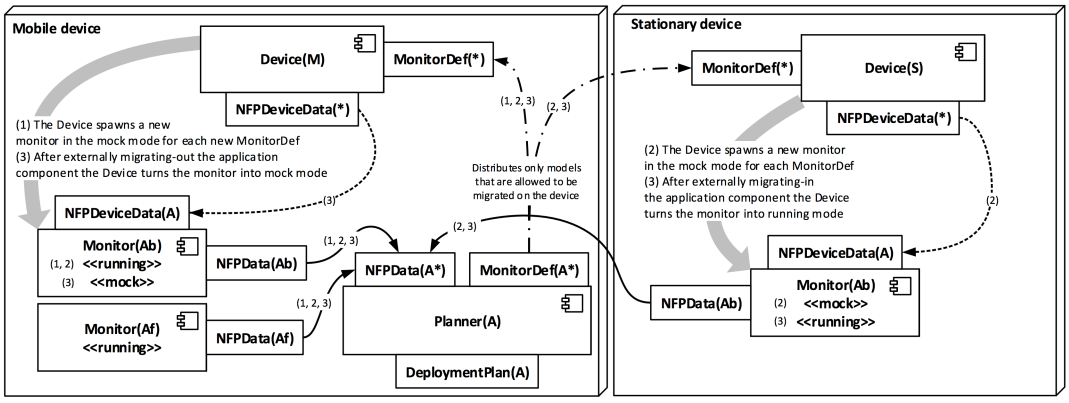


Figure 3.1: *Adaptation architecture of the running example: phases 1 (M isolated), 2 (S discovered), and 3 (Ab migrated to S). Phases 1,2,3 are in the figure denoted by (1), (2), (3). (Source: [21])*

representing its deployment alternatives on different devices. It is responsible for calculating *NFPData* for that particular alternative. The Monitor is either in *running mode*, which means that the corresponding application component is currently running on the node this Monitor belongs to or in *mock mode* - the application component is running on different node. Based on the current mode, the NFPData is obtained by measuring of running component or estimated by provided performance measurement model based on the machine-specific performance data (*NFPDeviceData*).

Device component Device component mirrors each computation node and it is responsible for spawning new Monitor components for each application component based on *MonitorDef*s provided by the Planner. It's also dedicated to obtaining machine-specific performance data (*NFPData*).

Planner-Device ensemble To inform Device components about application components available for offloading, it must be involved in an ensemble together with the Planner. Planner is the coordinator of the ensemble and monitor definitions (application component definitions) are being pushed to Device components. This is the point where the deployment can be constrained - it can be defined which application components are allowed to be deployed on which nodes.

Planner-Monitor ensemble This ensemble's purpose is to periodically gather *NFPData* from all available Monitors and push them to the Planner so it has all possible alternatives available for planning the deployment strategy.

Device-Monitor ensemble Each device pushes its *NFPDeviceData* to its Monitors in mock mode through this ensemble to let them estimate potential performance of the alternative they correspond to.

The number of nodes and Monitors can change very dynamically, since the devices are appearing and disappearing constantly. This is a great opportunity to use the emergent component ensembles, because they were introduced to address exactly this type of situations. Thanks to exploiting the features of EBCS, the adaptation architecture is scalable and robust. The architecture is also very flexible - the deployment can by arbitrarily constrained, the *NFPData* can be obtained by a measurement or by a performance model. According to [21], it is also scalable for extensions: "For instance the Planner itself can be subject to migration in case the application does not have any frontend. Additionally, when understanding the Planner as an entity controlling the NFPs of the application, it is possible to foresee the existence of multiple Planners per application, thus hierarchically decomposing the adaptation."

3.2 Open issues and possible solutions

3.2.1 Suitable DEECo implementation

The first challenge lies in obtaining a proper DEECo implementation. As stated before, a Java implementation called jDEECo has been developed. Since the goal is to implement a solution for Android mobile platform, the Java implementation is potentially a good fit.

The first issue is that, as discussed above, not all standard Java classes are available on the Android platform and it will most likely be necessary to perform certain changes to the framework to make it compatible with Android.

Other big challenge is introduced by the fact that jDEECo has practically never been used in a real-life application. It was tested in many simulations, but the framework lacks any support for network communication between nodes whatsoever. Therefore, an extension providing such capability has to be developed for the jDEECo library.

3.2.2 Offloadable application components

To enable parts of an application to be deployed on different machines, one must design an universal, compact and scalable solution for splitting an application into offloadable components. Such components should be encapsulated and provide a common interface for interaction over a wireless network.

Regarding how the components should be deployed on the nodes, there are probably two main options to choose from. Either the code is installed on all possible nodes in advance statically, or some kind of source code/binary migration has to be used. Latter would enable dynamic component deployment to new nodes without manual deployment in advance.

3.2.3 Deployment strategies

As described in the adaptation architecture, when deciding current deployment plan for all the components, many strategies are available for consideration. Deployment strategy generally consists of two steps: (i) obtaining *NFPData* (an estimate of the user experience) for deployment of particular component on certain node and (ii) deciding, based on provided set of *NFPData* for each component and each node, which application component will run on which node in the new deployment plan. Both steps can be customized in many ways.

In case the component is currently deployed on a different node, one way of calculating *NFPData* is using provided performance model involving device-specific data - for example a function of currently available memory, CPU usage, battery level, connectivity quality, or any other metric available - to obtain a rough estimate of user experience provided by corresponding alternative. In some cases it may be better to actually measure the execution of the component with a small static sample input. For instance, when implementing a text-recognition component, the performance can be calculated by measuring the execution time of recognizing a short word like “apple”. However, not all use cases provide this option. On the other hand, when the component is currently deployed on the corresponding node, one can simply measure the execution based on some metric. In our text-recognizing scenario, we can for example measure how long in average does it take to recognize one word. The other option is not to take the current mode into account and simply use the performance metric or small sample execution in both cases.

Later, when deciding which deployment alternative is the best for certain application component, a function choosing the best *NFPData* value from provided set has to be designed. This is the second entry point for potential customizations.

The reference architecture in the form of implemented framework should provide an easy way for customizing deployment strategies suitable for each use case.

3.2.4 Stateful components

Many use cases for computation offloading involve some kind of a soft state (perhaps even a hard state). A voice recognition component may feature a history of queries improving further recognition, a text recognition component may have some kind of basic user preferences, components may use a caching mechanism to improve their performance and many more.

Such feature definitely adds certain amount of complexity to the solution, because in this kind of dynamic environment, nodes appear and disappear on regular basis. Often they disappear suddenly and without any previous notice. (Wi-Fi out of range, Wi-Fi disabled by the user, etc.) The solution should therefore include a mechanism addressing this type of situation by seamlessly transferring the state between the nodes. The mechanism should be as robust as possible (given circumstances).

With stateful components implemented, another design decision is to be made. In a situation, where multiple mobile devices are offloading the same component to the same node, a separate state have to be maintained for each. The question that arises is if there should be a separate backend component for each device or only a single component holding a set of states for all clients involved.

3.2.5 Communication between layers

Mentioned in the jDEECo library description, the process methods of DEECo components are executed in a static context by the jDEECo runtime. Therefore no reference to the component itself or its dependencies is available inside these methods. However, the components need to have some channel for communication with a "world outside DEECo" - the components at the application layer need to be notified when DEECo control layer decides to change the deployment strategy for instance. Moreover, jDEECo component processes run in a background thread, while any changes to the application UI must be performed in the main thread.

3.2.6 Framework API

One of the goals is making the developed framework easy-to-use for transforming a regular Android application to an offloadable application where offloading is managed by DEECo. This involves designing a straightforward interface for smooth transition which may be quite challenging to do. The API should interfere with existing application's codebase as little as possible but sufficiently customizable at the same time.

3.2.7 Java and Android together

The reference architecture implementation should serve as a framework to make standard Android apps offloadable. The developed framework has to be compatible with Android platform as well as the desktop (Java SE/EE) since the purpose of the work is to enable parts of Android applications to be executed on more powerful nodes (Servers, Desktops). This requirement can bring certain challenges with compatibility that have to be considered during the development. The code must be either compatible with both platforms or split into two branches for either platform with as much common code as possible.

3.2.8 Multiple offloadable applications on a single device

Imagine a scenario, where multiple installed applications on an Android device are to be made offloadable. Should each application have its own instance of DEECo runtime? Perhaps there should be one central DEECo runtime installed on the device, which all offloadable applications would use. Similar issue arises, when designing a communication between application components. Offloadable parts of the applications need to offer an interface accessible via network. Different architecture approaches are available, let's discuss them shortly:

Each applications with its own offloading runtime In this scenario, every single application with offloadable ability would carry its own offloading (and jDEECo) runtime. The problem of this approach is that there may be a conflict of multiple runtimes in terms of networking. Each Android application run in a separate process, hence each would have to use a different network port for inter-device communication to avoid conflicts.

One common offloading runtime A different approach involves only a single instance of offloading (and jDEECo) runtime, which is used by all offloadable applications installed on the particular device. This approach is obviously beneficial in terms of efficiency and resistance to networking conflicts. On the other hand, this option introduces other significant disadvantages: (i) one instance of the runtime creates single point of failure for multiple independent applications, (ii) the user would be forced to manually install a separate application, which may be confusing for some, and finally (iii) since each application runs in different process on Android, an inter-application communication and dynamic loading of Java code to different application would have to be used. Inter-application communication is possible via `BroadcastReceiver`[1], yet not very convenient, reliable and easy to use. A more complex challenge is dynamic code loading. This can be theoretically achieved by using frameworks such as *OSGi[2]*, but on

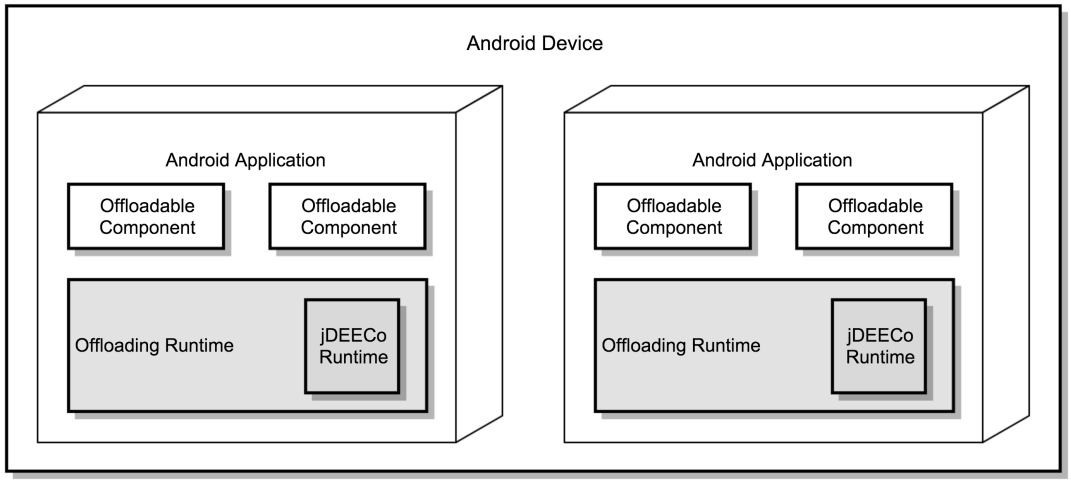


Figure 3.2: *Each applications with its own offloading runtime*

Android platform, OSGi capabilities are still very limited and the implementation would bring much complexity to the architecture.

Each alternative has its advantages and disadvantages but only one of the approaches has to be chosen.

3.2.9 Presenting a reasonable demonstration application

Wise choosing of a demonstration scenario for proposed reference architecture is undoubtedly challenging as well. The demonstration application should represent a use case that is useful for the user and suitable for computation offloading at the same time. It should present most of the architecture's features and possibly serve as a reference application or an example of making a regular Android application offloadable.

Use cases presented in Section 2.3 may be a possible fit for demonstrating offloading capabilities, since they have a good potential in terms of saved resources while being offloaded.

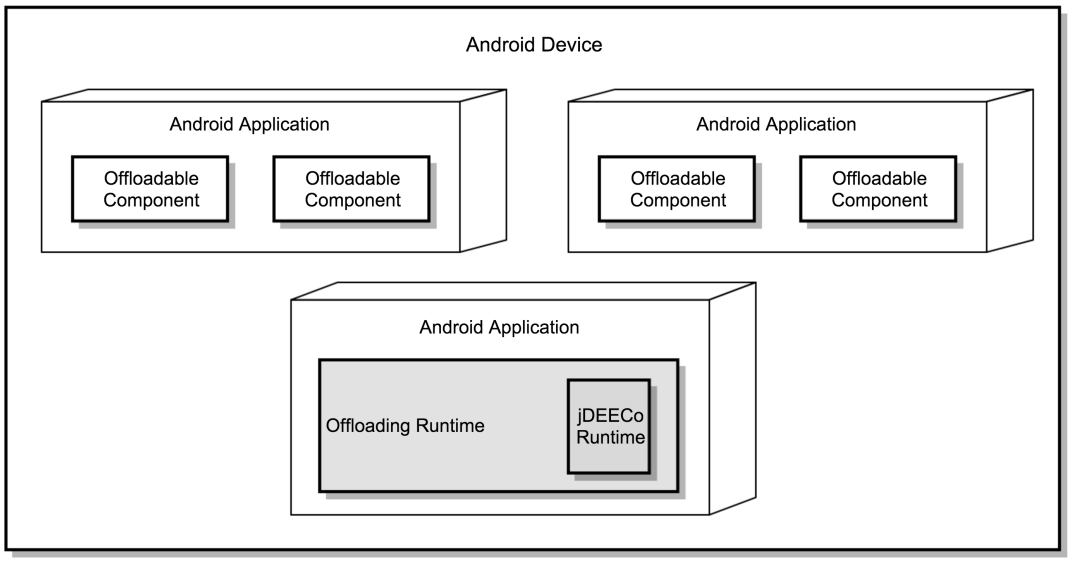


Figure 3.3: *Each applications with its own offloading runtime*

Chapter 4

Reference Architecture

In this Chapter, a solution is presented in form of a reference architecture for computation offloading managed by DEECo model. Firstly, a conceptual design of the reference is presented and later, in the next Chapter, implementation-related details are described. Furthermore, two demonstration applications presented in Chapter 6 can be considered as part of the reference architecture as well.

The architecture itself design comprises of a number of important and notable concepts discussed in subsequent Sections.

4.1 DEECo inter-device communication

To allow deploying DEECo components to different nodes in a network and form ensembles across the nodes, some sort of network communication has to be incorporated into the DEECo framework. As mentioned earlier, jDEECo framework does not provide any real network communication implementation yet. Thankfully, the framework is well designed and it is easily extensible.

Let's remind ourselves on how the DEECo runtime actually works. To be able to maintain encapsulation and autonomicity of the components, the runtime is basically designed to periodically clone the knowledge of all available components so that it is available locally and so that the membership functions and knowledge exchanges can be evaluated locally relying only on the local copy (a belief) of the remote knowledge. This is the key feature that makes the system so robust and dynamic.

Therefore, we don't really seek a way to let components communicate over a network. We rather need to implement a physical layer for cloning component's knowledge to all other nodes in the network that are involved. The jDEECo implementation features high level of granularity, so the extension can be incor-

porated in a very straightforward way.

Regarding the actual network interface, the nature of the goal we are trying to achieve (cloning the knowledge to all other nodes) points us one particular direction - broadcasting. Additionally, the knowledge cloning mechanism was not designed to be reliable in terms of communication. This all together suggests to use standard UDP broadcasting as the network interface for knowledge cloning.

Conceptually, the inter-device DEECo system will work like this: (i) each involved node (device) will run its own jDEECo runtime with its own components but identical ensemble definitions, (ii) the existing knowledge cloning manager together with incorporated physical networking layer will take care of knowledge cloning between those nodes and (iii) the components and ensembles will be evaluated separately at each of the nodes, but everywhere with the same outcome thanks to the knowledge cloning mechanism.

4.2 Offloadable app components

When making parts of an application offloadable (able to be executed on another machine) it is necessary to break the application to a set of encapsulated components with a common interface providing interaction via network. In the reference architecture, HTTP server is used as the interface. In fact, each offloadable component is represented by a RESTful[29] resource, which is able to define any number of methods accessible via standard HTTP methods (GET, POST, PUT, DELETE, etc.). A simple HTTP server serves these resources to make them available through a defined URL.

The developed framework defines a standard way of creating such components and registering them. It also provides an interface to interact with components deployed on a different node (knowing its IP address) in a very straightforward way, which is almost transparent to the developer's point of view.

To fulfill the requirement specified in Subsection 3.2.4, each offloadable component can hold a set of custom state representations (serialized, saved in a state repository) for each client interacting with it. The server is able to identify the client when it is calling one of the component's methods and pick the right state data from the state repository.

Because implementing dynamic source code loading through a network goes far beyond the goals of this thesis. The components are supposed to be deployed to all required nodes manually in advance. This limitation can in fact become a feature in some use cases. For instance when deployed on different platforms, each platform can feature different implementation performing identical operation, which can be more suitable for the particular platform and offer better results. See demonstration application in Chapter 6 for an example of

such scenario.

Furthermore, in order to optimize deployment of offloadable application components, each of them must implement two important methods. It needs to define a way of measuring potential performance of the corresponding deployment alternative - a method which is executed on all possible nodes returning an estimate of the quality of user experience (*NFPData*). Secondly, a function, which determines the best alternative given a set of *NFPData* values has to be defined. Both methods are executed by the offloading runtime automatically.

4.3 DEECo control layer

Now that it is sorted out how to offload an application component to another machine and interact with it knowing its IP address, the next step is to design a control layer, which decides which IP address should the client mobile application use for the actual execution. In other words, which node in the network to use for the computation offloading at a specific moment.

The control layer is represented by a runtime executed together with the target application on the mobile device as well as on other stationary or mobile devices. This runtime encapsulates DEECo runtime and uses adaptation architecture introduced in Section 3.1 as the base idea.

The architecture is designed in a way that DEECo management components are mirroring components on the application level. The communication between the layers is restricted so that only corresponding components can talk to each other.

4.3.1 Application-level components

The architecture defines a set of components on the application level. Note that these are not DEECo components. These are components in a traditional general meaning.

Backend This is an explicitly defined component described in the previous Section. There may be more than one Backend component within a single application. The components should be encapsulated enough, because they are the subjects of the offloading mechanism.

Frontend A special one of a kind component responsible for interaction with the user interface of the target application. It works as the link between the DEECo control layer and the rest of the application which should stay shielded from the DEECo management logic. The Frontend keeps track of which Backend component is offloaded to which device and informs the

user interface of any change in the deployment scenario provided by the DEECo control layer.

BackendStateData A component instantiated for each of the offloadable Backend components. It is dedicated to persisting the state data for the particular offloadable component and it is also responsible for making current state data available at the node to which the component is being offloaded at the moment. This means it takes care of caching/mirroring state data to the backend components in ACTIVE state.

4.3.2 DEECo-level components

Each device equipped with the runtime deploys its **Device** component and the client device (mobile device with the target application installed) deploys one **Planner** component for the target application. Both of them feature more or less the same functionality as described earlier in the adaptation architecture. The rest of the control layer is formed by Monitor components. Monitors are designed to mirror application components. There are three types of Monitors present in the architecture:

BackendMonitor A generic type of Monitor mirroring each offloadable application component (Backend). It is deployed by the corresponding Device component on each device for each offloadable component. The BackendMonitor is responsible for periodic gathering of the *NFPData* - the value representing potential performance of the particular alternative (component X running on device Y). The *NFPData* is then transferred to the appropriate Planner component for evaluation and determining the best alternative for deployment. The BackendMonitor is also tracking the corresponding offloadable component's state - it is either ACTIVE - it represents the best alternative for deploying the offloadable component, or NOT ACTIVE - it is not currently used for execution.

FrontendMonitor A Monitor dedicated to mirroring the Frontend component from the application level. It provides the Frontend with the data necessary for its functioning - current deployment plan, which is later passed to the application UI.

StateDataMonitor A generic Monitor deployed for each BackendStateData component linking it to the DEECo control layer. It is also responsible for instructing BackendStateData to pull the latest state data from the currently active Backend component to keep the local state data up to date.

4.3.3 DEECo-level ensembles

The data flow between the components is achieved by their involvement in several dynamic ensembles defined within the architecture:

PlannerToDevice This is the ensemble responsible for distributing the Backend Monitor definitions (*MonitorDefs*) from the Planner component representing the app to all available Device components (nodes). The Device component then deploys appropriate BackendMonitors to start offering offloading alternatives to the Planner.

NFPDataCollecting An ensemble dedicated to periodic collecting *NFPData* values from all BackendMonitors to the Planner. The knowledge exchange takes the latest *NFPData* value from the Monitor's knowledge and stores it into a map holding all *NFPData* values for different Backends on different Devices inside Planner's knowledge.

BackendStateDistributing As the Planner component is periodically constructing new deployment plan, this ensemble, based on the new plan, informs all Monitors of their current state. The Monitors corresponding to alternatives chosen for deployment are set as ACTIVE and all others become NOT ACTIVE.

ActiveBackendMonitorToFrontend BackendMonitors in ACTIVE mode are in this ensemble together with the FrontendMonitor. The knowledge exchange function sets the IP address of currently active alternative in the FrontendMonitor's knowledge so that it can notify the application user interface via the Frontend application component.

ActiveBackendMonitorToStateData Similar to ActiveBackendMonitorToFrontend, this ensemble informs matching StateDataMonitor component of currently active backend.

4.3.4 Control cycle

As all building blocks of the architecture have been introduced, the next step is to describe the architecture's lifecycle. We can say that the lifecycle basically consists of single cycle, which is repeated again and again bringing the needed functionality.

Before the cycle itself, an initialization has to be performed. It involves deploying static DEECo components. Planner component is deployed only on the device with the application user interface installed. It is initialized with the list of offloadable Backend definitions. Device component is deployed on all nodes

running the runtime an identified by the node's IP address. FrontendMonitor and StateDataMonitor (for each Backend definition) components are also deployed in advance.

Now, let's go through the control cycle step by step:

1. First of all, the offloadable Backend definitions in form of *MonitorDefs* are transferred from the Planner component to Device component via the knowledge exchange of the PlannerToDevice ensemble. The Device automatically instructs the runtime to deploy a BackendMonitor for each new *MonitorDef* received.

2. The BackendMonitors start to regularly obtain the *NFPData* and pass it to the Planner component through the NFPDataCollectingEnsemble.

3. The Planner, after receiving all *NFPData*, calculates the best alternative (IP address) for each Backend based on the data available at the moment. The BackendStateDistributing ensemble is then used to let the BackendMonitors know, which was determined to be in the ACTIVE and which in the NOT ACTIVE mode. The mode is persisted in Monitor's knowledge.

4. Once the Monitors are aware of their current mode, the ActiveBackendMonitorToFrontend and ActiveBackendMonitorToStateData ensembles couples ones in the ACTIVE state with the FrontendMonitor (or StateDataMonitor respectively) to let it know, which node is being used at the moment.

5. The FrontendMonitor component then uses a inter-layer communication to pass the new deployment plan to the application user interface via special Frontend component on the application level.

4.3.5 Timing constants

For better understanding, the control cycle was intentionally described as if the steps were actually executed in that order one after another. In fact, the DEECo control layer works differently. All those process methods inside components and knowledge exchange methods inside ensembles are actually performed periodically and individually. This means that certain parts of the system may not have the latest data available at all times, but eventually, everything gets always synced after all. For instance, right after step 3, it may happen that two or more BackendMonitor components may "think" they are in the ACTIVE mode and propagate this observation to the FrontendMonitor. But eventually, based on the rules, only one stays in the ACTIVE mode. This behavior is not wrong,

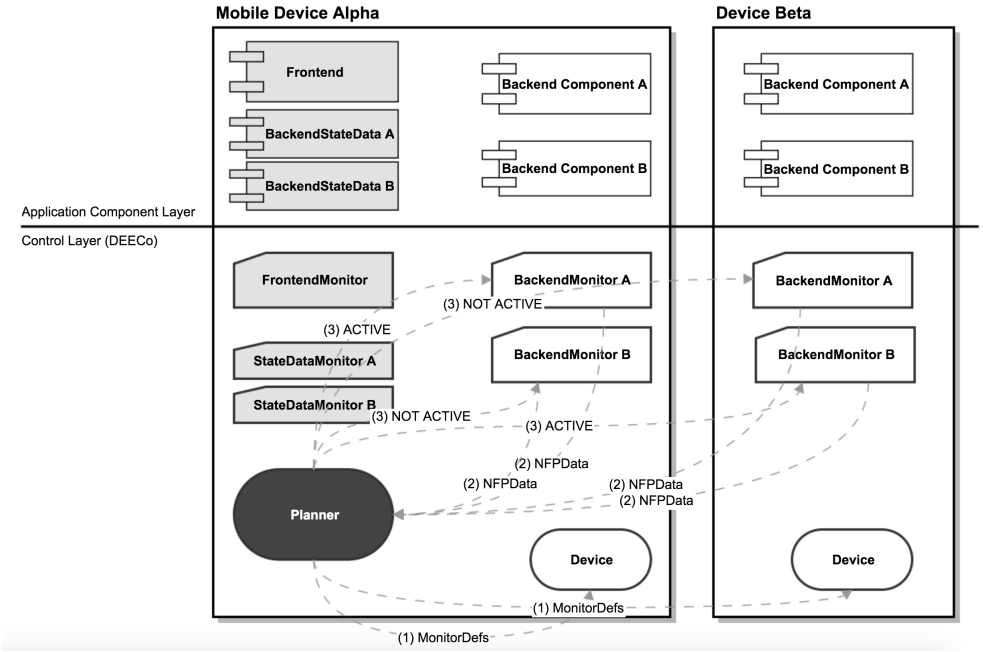


Figure 4.1: *The first three steps of the control cycle (dashed line: ensemble knowledge exchange; solid line: inter-layer communication)*

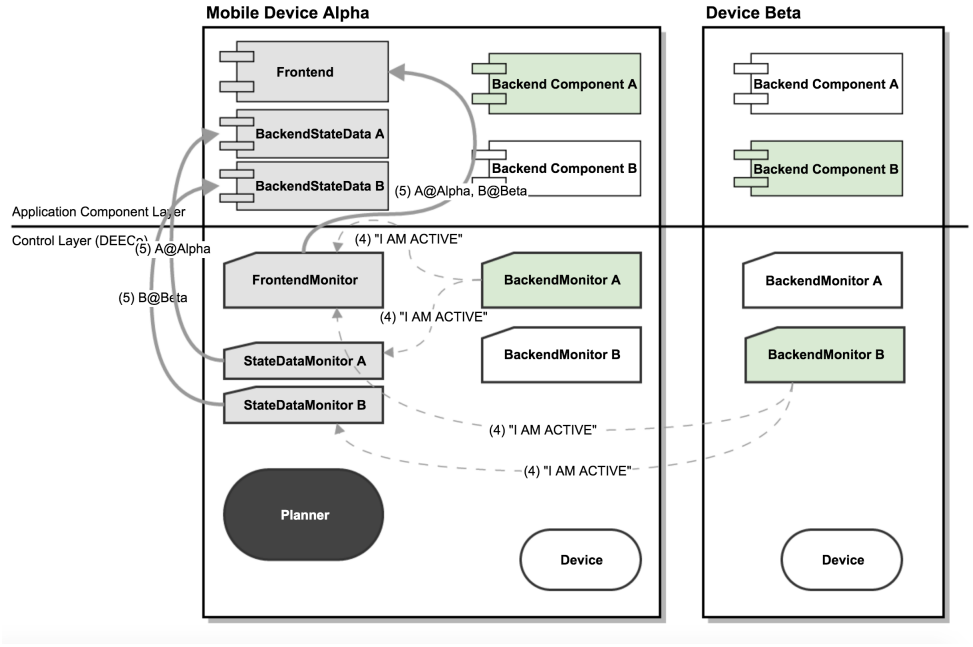


Figure 4.2: *Steps 4 and 5 of the control cycle (dashed line: ensemble knowledge exchange; solid line: inter-layer communication)*

since the DEECo model is designed to work similar to this itself (belief based behavior).

Therefore an important challenge is the setting of particular periods. They can represent a significant factor in the resulting overall performance. The steps of the control loop described above still needs to be executed one after each other to ensure the needed data flow. And since the operations are executed periodically, the length of the intervals directly affects the total duration of the control cycle. Bringing this to our scenario, the periods determine the total time needed for a mobile device to register a new node offering a better alternative for Backend deployment, and transfer the execution accordingly.

On the other hand, to achieve optimal results, the timing constants should not be too small, since excessive execution of the process and knowledge exchange methods can bring a big overhead.

Chapter 5

Implementing the Reference Architecture

5.1 Structure of the solution

The reference architecture was implemented in a form of framework, which is supposed to be used to bring offloading capabilities to an ordinary Android application. The entire implementation is realized as a Java project consisting of a set of sub-projects (modules).

jdeeco-java-library A module containing solely the jDEECo library binaries with its dependencies (modified to run on Android platform) in form of JAR files.

jdeeco-udp-broadcast-library A Java extension library for the jDEECo framework featuring knowledge cloning over UDP broadcast.

restlet-android A fork of the Restlet[16] library, which was slightly modified to work on the Android platform properly.

jdeeco-offloading-java-library A Java library module containing the offloading framework itself.

jdeeco-offloading-android-library An Android library module providing only the Android-specific parts of the framework implementation.

5.2 jDEECo knowledge cloning over UDP broadcast

The first implementation challenge was developing a network communication channel for jDEECo based on UDP broadcast. UDP broadcast works in a way that the packets are sent to a special logical IP address at which all connected nodes are able to receive them. Unlike multicast, all nodes receive the packets and the sender can not specify the set of recipients. The broadcast address can be calculated from a device's IP address and the subnet mask.

The implementation of UDP broadcast networking differs on Android and standard Java platform since the API for accessing the network interface of the device is different. Therefore the implementation of some methods such as obtaining the broadcast address or obtaining the device's assigned IP address itself has been split. In fact the Java implementation is compilable on the Android platform, but it simply does not return proper values. Therefore there is a Java library and an Android extension library which needs to be packed only when deploying to the Android platform.

Once the implementation of methods for sending and receiving packets over the network is done, it's time to incorporate it as a physical layer to the jDEECo knowledge cloning mechanism. The broadcast data flow is a perfect fit in this situation since knowledge cloning is supposed to take the local components' knowledge data and clone it to every available node in the network (or nearby) so that each jDEECo runtime has access to the latest data possible.

The incorporation of UDP broadcast communication to the jDEECo framework consisted mostly of adding a brand new implementation of the *NetworkInterface* interface, subclassing the *KnowledgeDataManager* and creating new runtime builder putting all the new pieces together and returning modified runtime instance. The process involved some necessary generalization, since the existing *KnowledgeDataManager* and *NetworkInterface* were used for simulating a different network scenario involving embedded devices.

5.3 Offloadable backend components

In the previous Chapter we decided that the offloadable backend components should provide their interface through a REST interface over the HTTP protocol. To achieve this, a number of implementation alternatives are available to choose from. Many Java libraries provide more or less an easy way to deploy HTTP server and respond to incoming requests.

Since the components are supposed to be deployed on both mobile devices running Android and stationary device running standard Java environment, the

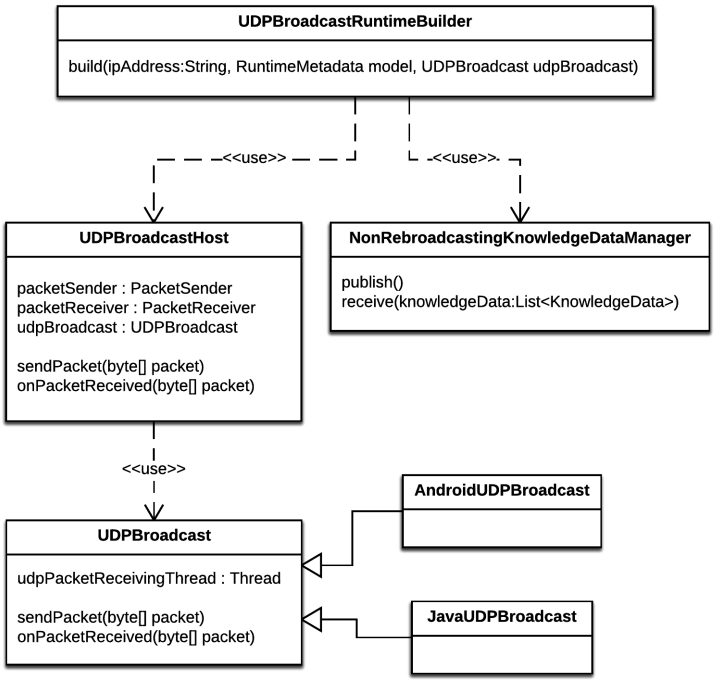


Figure 5.1: *Class diagram covering UDP broadcast extension for the jDEECo library*

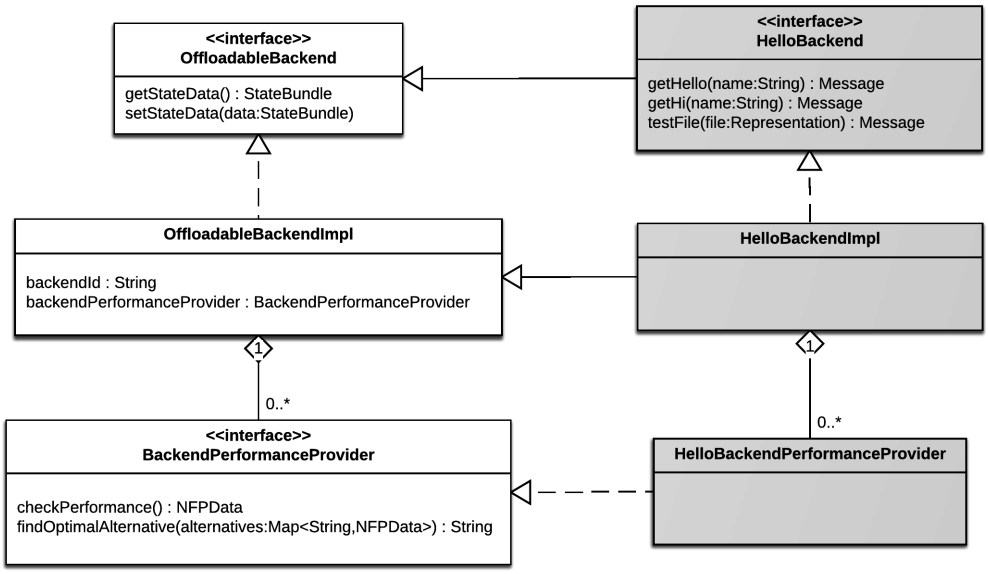


Figure 5.2: *Class diagram covering backend component deployment*

library must fully support both platforms. This requirement puts some of the alternatives out of picture. Out of the rest of them, one library stands out in terms of provided functionality, ease of use and simple componentization. It is the *Restlet* library[16]. Other alternatives worth consideration were *Jersey[11]* or *NanoHttpd*[13]. The former was not used since its Android support turned out to be very problematic[10] and the latter was, although very lightweight, found to be too low-level and it would require a significant amount of coding to achieve the functionality provided by *Restlet*.

The component system is designed so that each component consists of a Java interface declaration describing its external interface (methods used to interact with the component) and a Java class implementing those methods.

The interface must always extend abstract *OffloadableBackend* interface defining "service methods" common for all offloadable backends. Furthermore, each interface method must be annotated with one of the annotations (*@Get, @Post, @Put, @Delete*) provided by the Restlet library to mark the method accessible via the HTTP interface. The annotation must contain a unique string attribute defining resulting URL path of the HTTP endpoint.

The backend's functionality itself is residing in a Java class implementing the backend interface and extending the *OffloadableBackendImpl* class. The superclass contains the implementation of common "service methods" and required functionality to make the methods available via HTTP interface coming from the Restlet's *ServerResource* class, which the superclass extends. When instantiating a backend implementation, the superclass constructor must be called with two important parameters - a unique URL path used for serving the backend and an implementation of the *BackendPerformanceProvider* interface. As the name suggests, the implementation is supposed to implement methods used by the offloading runtime to benchmark the backend's performance on current machine. It consists of two methods - one returns the *NFPData* and the other is responsible for selecting the best alternative (device IP address) out of a set of pairs of device IP address and *NFPData* value. For clarification, the *NFPData* class is abstract. The actual form of performance metric result is completely arbitrary. See Figure 5.3 for an example.

The *OffloadableBackendImpl* superclass also provides a functionality to access current state data from within the exposed methods as well as special methods exposed via HTTP to allow the offloading runtime to pull/push the entire state data bundle from/to the backend as described earlier.

One of the benefits of using HTTP as the transport protocol is the ability to easily transport binary files as well as text data. Transferring binary files comes in handy especially in use cases targeted by this thesis - image recognition, sound recognition, image analysis, etc. The library's basic support for file transferring was even enhanced to support multiple file transport together with text data using the Multipart MIME Content-Type[17].

5.3.1 Deploying the components

When an application contains two or more offloadable backend components, only a single HTTP server is deployed serving all backends on different routes instead of having a separate server for each component. Although latter solution features a better encapsulation, selected alternative brings less complexity (no issues with port collisions for instance) and it is certainly more efficient since each server runs on a separate background thread.

The Restlet library provides an interface to register a properly annotated class as a REST resource and make it's annotated methods accessible via HTTP interface. With the right configuration and setup, it takes care of server deploying, handling incoming HTTP requests and propagating them into the right method of the right resource class. It also handles most of the serialization of methods' arguments and return types.

The framework wraps the Restlet library within the *OffloadingManager* class

```
1  interface HelloBackend extends OffloadableBackend {
2
3      @Post("?hello")
4      public Message getHello(String name);
5
6      @Post("?hi")
7      public Message getHi(String name);
8  }
9
10 public class HelloBackendImpl extends OffloadableBackendImpl
       implements HelloBackend {
11
12     public HelloBackendImpl(String path, final Context context)
           {
13         super(path, new BackendPerformanceProvider() {
14             @Override
15             public NFPData checkPerformance() {
16                 return new SingleValueNFPData(Device.
                       getPerformance(context));
17             }
18
19             @Override
20             public String findOptimalAlternative(Map<String,
                   NFPData> alternatives) {
21                 return SingleValueNFPFata.getMax(alternatives);
22             }
23         });
24     }
25
26     @Override
27     public Message getHello(String name) {
28         int count = getStateData().getInt("hello_counter", 0) +
               1;
29         getStateData().putInt("hello_counter", count);
30         return new Message("Hello no. " + count + " to " + name
               + " from " + android.os.Build.MODEL + "!", new Date()
               .getTime());
31     }
32
33     @Override
34     public Message getHi(String name) {
35         return new Message("Hi to " + name + " from " + android.
               os.Build.MODEL + "!", new Date().getTime());
36     }
37 }
```

Figure 5.3: *Example of an offloadable application component*

which provides most of the functionality. The key endpoint is an instance of Restlet's *Component* class which represents the server and an instance of *Router* class which takes care of URL routing. *OffloadingManager* class also features a completely transparent way of interacting with a backend deployed on another machine. All the developer needs to provide is the name of the interface (backend component) and an IP address of the target node. Since that moment, the framework provides an object implementing desired interface wrapping all of the networking logic inside and the developer can use the object almost as if it was deployed locally.

5.4 DEECo control layer

Implementing the DEECo control layer of the offloading framework involved mainly translating the design requirements to the language of the jDEECo framework and designing data structures for storing and transporting the knowledge data.

Each component and ensemble definition described in the previous Chapter is represented by a Java class annotated with proper jDEECo annotation (*@Component* or *@Ensemble*). The *DEECoManager* class is a management class wrapping the jDEECo runtime instance. It provides methods for controlling the lifecycle of the runtime (initialize, start, pause) which are essential especially when integrating with the lifecycle of an Android activity or service. New jDEECo component instances and ensemble definitions are also registered through this wrapper. They can be both registered ahead of the runtime initialization as well as later when the runtime is already running. Latter is used when deploying BackendMonitor components from the Device component dynamically.

The *DEECoManager* instance is then wrapped by the *OffloadingManager* class, making it - together with the ability to deploy offloadable components - a single management class used by the application itself to enable offloading capabilities. The *OffloadingManager* class uses the Singleton design pattern so its single instance is available statically through the entire application.

When initializing the offloading runtime, the special management application components mentioned earlier are also instantiated. Particularly, a single instance of the *Frontend* class is created, which acts as a middleman between the control layer and the user interface of the application. It provides a set of useful callback registration methods providing a way to listen for deployment plan changes and other interesting events. Additionally, an instance of *BackendStateData* class is created for each deployed backend component within the *OffloadingManager*. Each is capable of storing the state data of an arbitrary type for the backend component and provides methods for pushing and pulling the

data to/from backend components deployed on different nodes. These methods are called by the DEECo control layer to achieve a robust state data caching mechanism.

5.5 Notable implementation challenges and solutions

During the process of implementing the reference architecture, a number of problems and complications stepped into the way and must have been faced. In this Section a notable subset of those issues are discussed and final solutions are presented.

5.5.1 Handling sudden node disconnection

One of the most interesting problems faced during the development of the DEECo control layer was handling a situation when a device gets disconnected from the system. There are two primary reasons for node's disconnection.

- The device can leave the network physically. It can get out of Wi-Fi signal range, the network cable can be disconnected, or part of the network can simply encounter a malfunction. The user may also disconnect the device from the network manually or the device may switch to a different network automatically based on the signal strength for instance.
- The offloading runtime (including the DEECo runtime) may be stopped at any point. It can happen by exiting the application, killing the service executing the runtime, system crash, etc.

Both of the reasons provide the same outcome - the backend application components residing on the device are no longer reachable from other nodes. Therefore it doesn't provide a suitable offloading alternative to the mobile application.

The knowledge of an existing BackendMonitor component representing the alternative is distributed through the network by knowledge cloning mechanism. Once the BackendMonitor instance deployed on a certain device stops working as a result of reasons described above, the knowledge of the component is never removed from other nodes. The jDEECo framework doesn't provide any way of checking if a component is still alive or it is just a remaining knowledge of a component whose process functions are no longer being executed.

To overcome this serious problem, a live-checking mechanism was incorporated into the DEECo control layer. Each of the DEECo components feature an additional process with a single purpose. It is executed every second and all

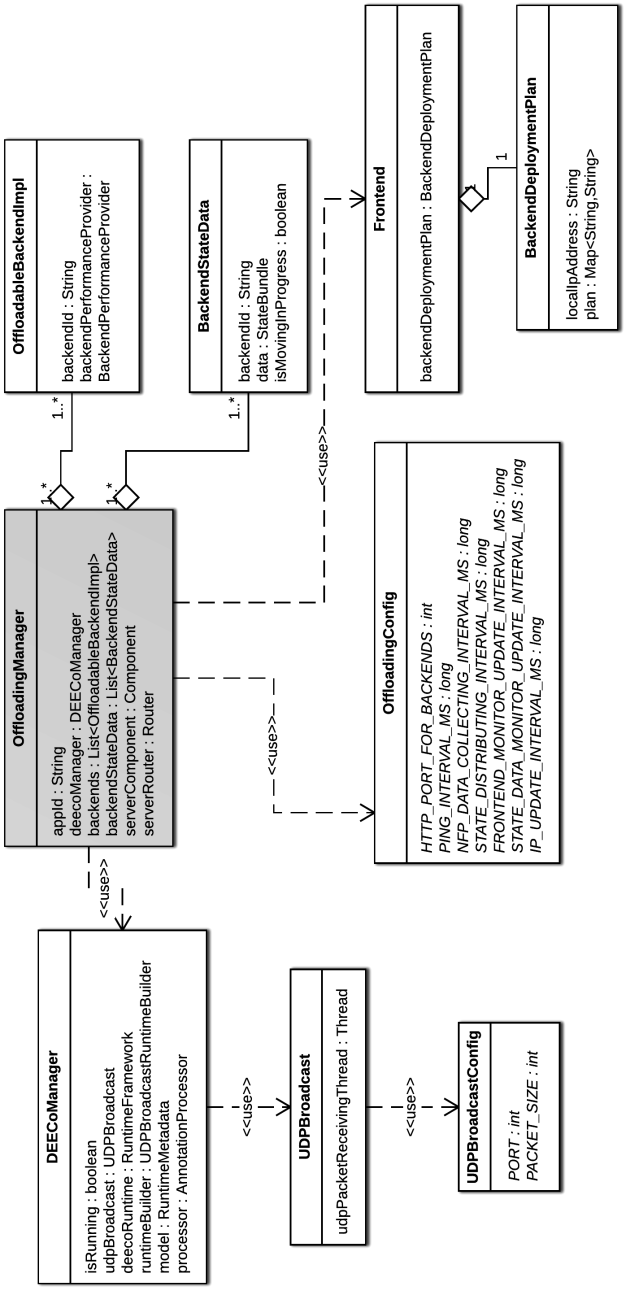


Figure 5.4: *Class diagram covering the core of the offloading framework, describing important relationships between main entities*

it does is writing current timestamp to the component's knowledge. The other part of the live-check mechanism is checking the last timestamp value of both components during the membership function of all ensembles. If the component's last timestamp is older than a predefined threshold, it is considered dead and the membership function returns false. Thanks to this process, only active components are involved in the ensembles.

Additionally, since the Planner component gathers the *NFPData* values from all BackendMonitors and stores them inside its knowledge, eventually some of the stored values will belong to alternatives no longer available. To avoid this, The *NFPData* values are timestamped as well and the timestamps are once more checked against a predefined threshold.

As previously mentioned, the DEECo model works in a way that the ensemble's membership function is executed on each device separately expecting the same results thanks to the knowledge cloning. Hence, while comparing the component's timestamp against current timestamp, each of the compared timestamps may have been created on different machine. This results in a required assumption of synchronized time among involved machines well enough, so that the possible offset is not higher than the threshold value used. Since most of the devices automatically synchronize current time with time servers, this requirement is generally met.

The live-checking mechanism is actually a feature, which could be included in the jDEECo framework itself. The DEECo runtime should check the component's activity automatically on its own because the developer should be shielded from this kind of internal logic. According to the plans for the jDEECo development, this is a feature which is currently being considered for future implementation.

5.5.2 Transitioning between networks

Another issue arising from the dynamic nature of the environment of mobile devices is handling of transitions between different networks. As described, the Device component is instantiated with the device's IP address. Unfortunately, the IP address can change as a result of a number of situations:

- Switching from one (Wi-Fi) network to another
- Transitioning between a cellular network to a Wi-Fi network
- Leaving a Wi-Fi network and connecting to a cellular network
- Leasing a new IP address while connected

For these reasons, components with a reference to the device's IP address were modified to periodically check the network interface API for current IP address in case it was changed. If that's the case, the IP address is overwritten in the knowledge and subsequent interactions use it as current device identification.

5.5.3 Inter-layer communication

Since DEECo component's process methods are executed statically, the components can not use any reference to other objects at runtime. The instantiation process of the component class may seem like the runtime works with the components' instances, but it is actually only used as a straightforward method of initializing component's knowledge when being deployed.

Hence, a static communication channel had to be used for a communication between the components in the DEECo control layer and the application components. For this purpose, the *Event Bus* design pattern seemed like a good fit. It was designed as a communication channel between components (within the same process) without a need for the components to be aware of each other. Any component can subscribe to a certain type of events, while any other component is able to publish the kind of event at any time. The components only need to have an access to the same instance of the event bus, which can be a application-wide static singleton.

There is a number of event bus implementations out there with different features available - *Google Guava EventBus*[7], *Greenrobot EventBus*[8], *Otto[14]* and more. While others provide more features and/or better performance, the Google Guava implementation was chosen for our purpose because of its full support for both Android and Java platforms. For an example of the usage, see Figure 5.5.

5.5.4 Multi-application environment

The problem of multiple applications using the offloading framework running side by side on a single device presented in Subsection 3.2.8 was solved by choosing the option of packing the offloading runtime within each of the applications. The framework API provides a way to customize network ports used for the DEECo communication and serving the backends over HTTP. When different ports are used by different applications, no conflicts will occur while running at the same time.

```
1  class ComponentA {
2
3      public ComponentA(){
4          EventBus.getInstance().register(this);
5      }
6
7      @Subscribe
8      public void onEvent(UpdateUIEvent e){
9          ...
10     }
11
12     ...
13 }
14
15 class ComponentB{
16
17     ...
18
19     public void updateUI(){
20         EventBus.getInstance().post(new UpdateUIEvent());
21     }
22 }
```

Figure 5.5: *Example of the Google Guava EventBus usage*

5.6 Compatibility

The minimal requirements are often affected by used libraries and/or frameworks.

Android at least version 4.0 (API level 14), optimized for and compiled against version 5.1 (API level 22)

Java at least version 7, compiled against version 7

The offloading framework was tested on following mobile devices:

- *Google Nexus 5 (LG D281)* running stock Android 5.0, 5.0.1, 5.0.2 and 5.1
- *Google Nexus 7 (2013)* running stock Android 5.0.2 and 5.1
- *Motorola Moto G* (2013) running Android 4.4 and 5.0.2
- *Xiaomi Redmi 1S* running MIUI v5 based on Android 4.3
- *LG Optimus One (P500)* running Android 4.0.3
- Android Emulator running stock Android 4.0.3, 4.4 and 5.0.2

All of the devices were compatible with the framework and the demonstration applications worked properly.

The device used tested as the stationary device was *Apple MacBook Pro (2013, 13" Retina)* running 64-bit Java version 1.8.0_25.

The framework was tested on multiple Wi-Fi networks at home, public places and enterprise. The network must not have UDP broadcast communication manually disabled, which was the case when testing the framework at a university building using the *eduroam* network.

Chapter 6

Demonstration Applications

As a part of the solution, two Android applications were developed to demonstrate features and usability of the offloading framework. The demonstration applications are the last piece of the reference architecture. The applications serve as an example showing the process of transforming an arbitrary Android application so that it has offloading functionality driven by the DEECo model.

Both applications were tested on all of the devices listed in previous Chapter. The minimal requirements remain the same as well. For building and running instructions, see the README files in corresponding modules.

6.1 Development Demo application

The first application was created rather for testing purposes during the development. Unlike the other application described in the next Section, this application doesn't serve any specific usable purpose. It solely demonstrates the capabilities of the framework and provides a way to simulate many possible scenarios of the offloading mechanism.

The application features a single offloadable application component providing a set of exposed methods. The methods are designed to test a transfer of serializable data structures as well as binary files. The key feature of this application allowing the user to test various scenarios is that the performance of given deployment alternative is provided by the user through a simple seekbar (slider). The application can be installed on multiple mobile devices and the offloadable backend component can be deployed to a computer equipped with the Java runtime. Each device can set its performance through the user interface and eventually, all of them will offload the backend component to the best-performing device - either one of the mobile devices or one of the stationary devices, where only the backend component is deployed. By adjusting the

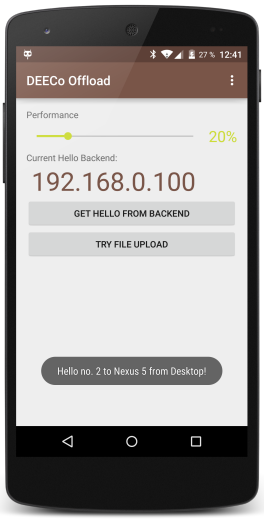


Figure 6.1: *The Development Demo application user interface*

performance of the different devices, the offloading framework's response can be tested for various possible situations.

The application always displays the IP address of the device, which is currently used to offload the computation. The user can communicate with the backend at any time by clicking one of the on-screen buttons.

One of the buttons sends a custom serialized *Message* object carrying a name of the client device and the backend component returns a greeting containing the name of the device it was executed on. The backend component also exercises the state data features by counting the greetings it has sent to the particular client. When the deployment changes and the backend component "moves" to another node, the greetings counter keeps its value thanks to the state data caching mechanism. The message received from the backend is displayed as a flash message on the bottom of the screen.

This demonstration application consists of two modules included in the main project - **jdeeco-offloading-android-demo** containing the Android application itself and **jdeeco-offloading-android-demo-java-backend** containing a command-line Java application running the offloading runtime and serving the backend component for other devices.

6.2 Offloadable OCR application

The second demonstration application was developed to show offloading capabilities on an application with a real use and purpose. It is a text recognition application (OCR) able to recognize a text either from a picture taken right inside the application by the device's camera or from an arbitrary image selected from device's storage. The text recognition use case was selected based on the discussion in Chapter 2 - it is a use case with a large potential of saving resources while using computation offloading.

The application uses the Camera API provided by the Android platform to display a live camera preview on the screen. The preview is able to respond to a screen orientation change and adjust the preview properly. The user can either shoot a photo right from the application or select any photo from gallery. Once the photo has been taken or selected, the application starts recognizing any texts found in the picture. When the process is done, user is presented with the result containing the recognized text string and the duration of the recognition process.

The recognition is performed by an open source OCR engine called *Tesseract*[20]. According to the authors, it is "probably the most accurate open source OCR engine available". It provides a great support for all major platforms out there - Windows, Linux, Mac OS X, Android and iOS. The authors recommend to use the engine in cooperation with the *Leptonica*[12] image processing engine. Currently, there is a large number of libraries wrapping Tesseract and Leptonica engines to make it available through many programming languages on various platforms.

To improve the text recognition process, Tesseract always needs a trained data file to perform the recognition. It needs to be present in the *tessdata* directory passed as an argument to the engine initialization method. Trained data files can be downloaded for many languages from the main Tesseract repository.

The demonstration application is able to perform OCR process on its own using the *tess-two*[18] library, which brings the Tesseract engine to the Android platform. A compiled tess-two Android library can be found in the **tess-two** module. The trained data file for English language is included in the application's assets and it gets copied to the device's external storage once installed.

Since the goal is to make the application offloadable, the OCR mechanism is wrapped inside an offloadable backend component. The potential OCR performance of a device is determined by performing the actual recognition on a small sample input (An image, 101x56px large, containing a word "The") and measuring the time needed for the process. Later, the alter-

Figure 6.2: *A sample input used for measuring potential OCR performance*

Figure 6.3: *The Offloadable OCR demonstration application in action*

native, which can perform the operation fastest, is selected as the best one.

To deploy the OCR backend component to a stationary device, a Java command-line application was created containing the offloading runtime with almost the same backend component. Since the tess-two library can be used on the Android platform only, another Tesseract wrapper library was chosen for the Java platform - *Tess4J*. It is a Java JNA wrapper expecting Tesseract and Ghostscript[1] software installed on the machine. Even if one of the libraries was compatible with both platforms, the usage of two different implementations can be beneficial because each can be tailored for the particular platform exploiting its maximum potential. The output of the recognition process is the same on both platforms because the underlying layer is the same Tesseract engine only compiled for the specific platform.

When backend is deployed on a machine residing in the same network as the mobile device and the performance of the OCR process is better, the application seamlessly switches currently used backend to the one deployed on the stationary machine and the actual text recognition is performed there. The application only sends the picture and after the process is done, it receives a string representation of the result. The user interface always displays the IP address of the device hosting currently used backend. A comparison of resulting performance is discussed in the following Chapter.

[1] An interpreter for the PostScript language and for PDF

In this demonstration application, the offloading runtime resides in the main activity, meaning that it is active only when the application is opened. However the runtime can be integrated within an Android service just as easily. This alternative provides a way to offload background tasks, but it must be used with care, since continuous execution of the runtime can become a resource intensive operation itself. Additionally, when placing the runtime inside a background service, the application can provide offloading alternative to other devices in background.

The Offloadable OCR application can be found in the **jdeeco-offloading-android-demo-offloadableocr** module and the Java command-line application containing the backend resides in the **jdeeco-offloading-android-demo-offloadableocr-java-backend** module. The compiled library of the tess-two library is available in the **tess-two** module.

Chapter 7

Evaluation

In order to determine whether the application can benefit from using computation offloading controlled by the DEECo model, an evaluation process of some kind has to be performed. In fact, performance the Offloadable OCR application described in previous Chapter was used as the subject of the performance evaluation. An evaluation screen has been added with functions dedicated to benchmark the offloading framework and, by extension, the reference architecture.

7.1 Method

The framework was benchmarked by comparing the performance of the application with the framework enabled and OCR tasks offloaded to a stationary device, and with the framework disabled and OCR tasks executed locally. The *EvaluationActivity* screen provides a way to set up a number of test cases and start performing OCR operations one by one with offloading framework enabled or disabled. The recognition is performed on a sample image shipped with the Tesseract engine. It is a 139kB JPEG image with resolution 1024x800px displayed in Figure 7.1. Once the desired amount of test cases is performed, a report file is generated and saved to the device's external storage. The report file is in JSON format and contains following parts:

- Durations of individual OCR operations (test cases)
- Identification of the host executing the OCR operations
- Total duration of all test cases together
- Battery level percentage decrease during the entire process

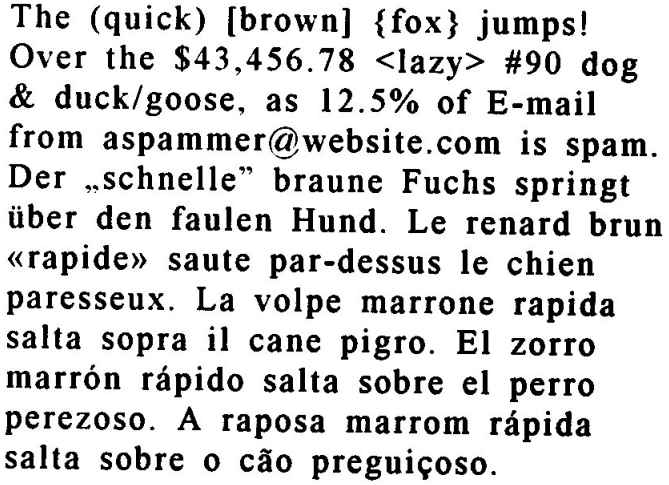


Figure 7.1: *The sample image used for OCR offloading evaluation*

The data from the report files can be compared against each other resulting in response time and battery comparison of both alternatives. Additionally, a CPU utilization profiler shipped with the Android SDK is used to track CPU usage during execution.

7.2 Setup

The evaluation process was performed on two mobile devices and one stationary device. Here are their brief specifications (Source: [9]):

- ***Google Nexus 5 (LG D281)*** running stock Android 5.0.2

CPU:	Quad-core 2.3 GHz Krait 400
GPU:	Adreno 330
Memory:	2 GB RAM
WLAN:	Wi-Fi 802.11 a/b/g/n/ac
Battery:	Li-Po 2300 mAh battery

- ***Xiaomi Redmi 1S*** running Android 5.1

CPU: Quad-core 1.6 GHz Cortex-A7
GPU: Adreno 305
Memory: 1 GB RAM
WLAN: Wi-Fi 802.11 b/g/n
Battery: Li-Ion 2000 mAh battery

- ***MacBook Pro (Retina, 13-inch, Late 2013)*** running Mac OS X Yosemite 10.10.2

CPU: 2,4 GHz Intel Core i5
GPU: Intel Iris 1536 MB
Memory: 8 GB RAM 1600 MHz DDR3
WLAN: Wi-Fi 802.11 a/b/g/n/ac

The wireless network used while performing evaluation supports Wi-Fi standards 802.11 b/g/n/ac. It works in both 2.4GHz and 5GHz bands.

In case of mobile devices, most of the processes belonging to applications installed were killed before performing the evaluation so they don't interfere with the observed process and bias the results.

7.3 Output

The evaluation process was executed on both mobile devices with and without offloading enabled. Each time, 300 OCR operations were performed sequentially. The reports gathered from evaluations were used to create Table 7.1 . Individual

Offloading disabled		
	Total duration	**Battery level drop**
Google Nexus 5	22 min. 19 sec.	11 %
Xiaomi Redmi 1S	31 min. 4 sec.	12 %
Offloaded to MacBook Pro		
	Total duration	**Battery level drop**
Google Nexus 5	6 min. 7 sec.	2 %
Xiaomi Redmi 1S	6 min. 24 sec.	2 %

Table 7.1: *Output of the evaluation process consisting of 300 test cases performed on each mobile device*

test case durations from a single evaluation report are presented in Figure 7.2.

It plots the lengths of individual OCR operations with and without offloading enabled.

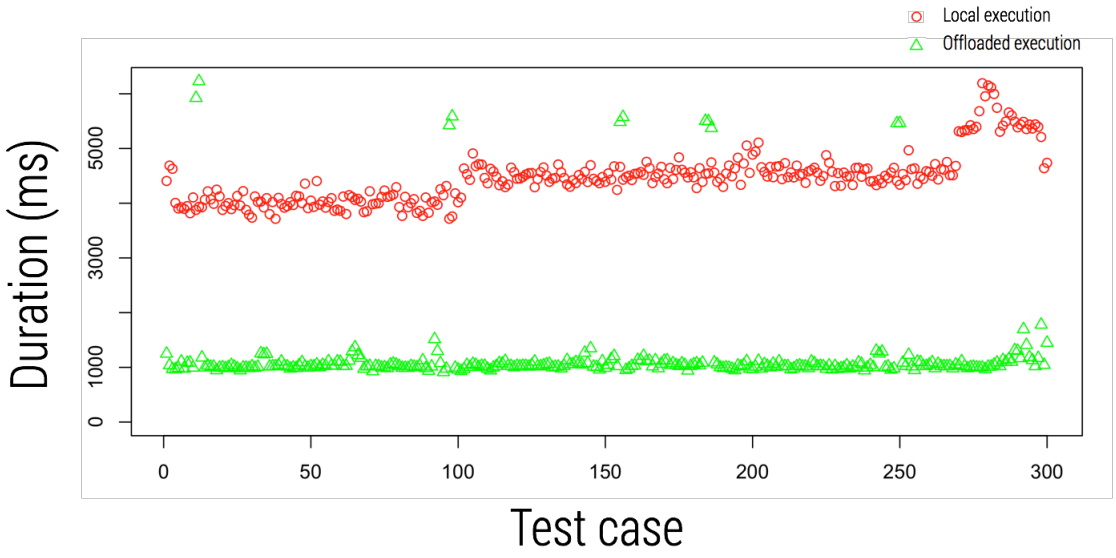


Figure 7.2: *Comparison of test case durations with computation executed locally and offloaded to a stationary device. Measured on Google Nexus 5 as the mobile device and MacBook Pro as the stationary device. A total of 300 test cases were performed.*

Furthermore, CPU profiles were gathered as part of the evaluation process on the Google Nexus 5. Figures 7.3, 7.4, 7.5 and 7.6 display CPU usage of the device while in four different states. The first Figure shows usage when the offloading framework is disabled and OCR operations are currently not running, second was captured during sequential OCR operations were performed without offloading framework enabled (executed locally). Third profile was captured when offloading framework was enabled, without OCR tasks running, and the last profile was taken during sequential OCR operations offloaded to a stationary device (MacBook Pro).

7.4 Observations and conclusion

The outputs presented in previous Section clearly demonstrate that a mobile device user can benefit when an application features computation offloading controlled by the DEECo model. Even when the OCR tasks were offloaded to a laptop with average specifications, they were performed usually up to 5

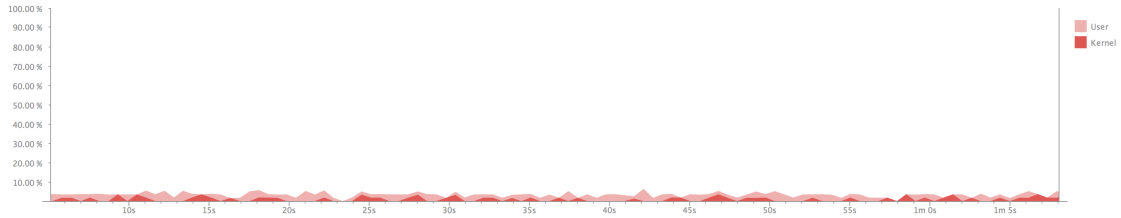


Figure 7.3: *CPU utilization profile of the application without offloading framework in a standby mode (Google Nexus 5)*

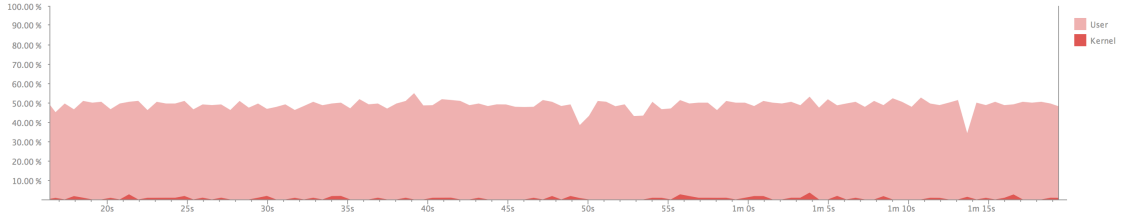


Figure 7.4: *CPU utilization profile of the application without offloading framework while performing OCR tasks sequentially on its own (Google Nexus 5)*

times faster consuming 6 times less battery capacity. Furthermore, from the plot displaying individual durations of OCR operations (Figure 7.2), we can see that the difference between offloaded execution and local execution is significant and relatively stable.

One important observation comes from the CPU profiles - the offloading runtime (including jDEECo runtime) keeps the CPU busy when in a standby mode (not performing any OCR tasks). This is the price coming from the nature of DEECo component model, whose runtime continuously evaluates membership functions, knowledge exchange functions, component's processes and more. On the other hand, when comparing Figure 7.5 and Figure 7.6, almost no difference in terms of CPU usage can be spotted when the OCR tasks execution begins.

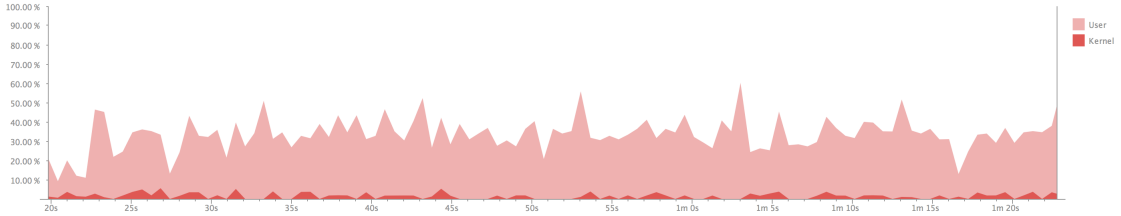


Figure 7.5: *CPU utilization profile of the application with offloading runtime enabled in a standby mode (Google Nexus 5 & MacBook Pro)*

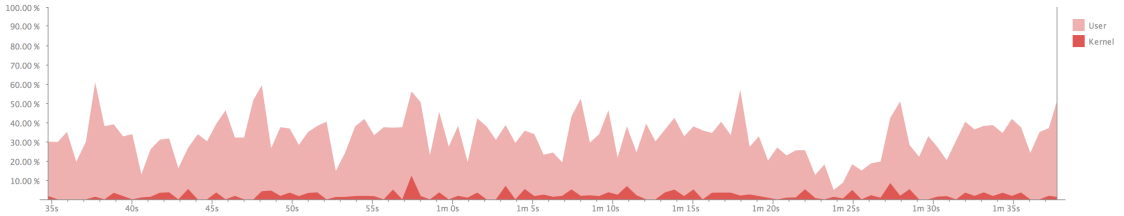


Figure 7.6: *CPU utilization profile of the application with offloading runtime enabled while sequentially performing OCR tasks offloaded to the stationary device (Google Nexus 5 & MacBook Pro)*

Chapter 8

Making an Android Application Offloadable

Generally, the Development Demo application (**jdeeco-offloading-android-demo** module) can be used as an application skeleton when developing a new application with offloading capabilities or as a simple example when making an existing application offloadable. The module declares **jdeeco-offloading-android-library** as its dependency, all other necessary modules and libraries are linked transitively. Alternatively, the application can add compiled JAR files as its dependency, but all transitive dependencies must be added manually by copying generated JAR and AAR files available in the *release* directory. Additionally, libraries defined in **jdeeco-offloading-java-library** module's *build.gradle* file must be also added manually to the application's dependencies.

Note that it is crucial that the target application must declare all permissions listed in the demonstration application's *AndroidManifest.xml* file marked as "Framework related permissions". Without those permissions, the application will not have access to the network interface and the offloading mechanism will not function at all.

The offloading framework can be incorporated into an activity lifecycle as presented in the demo application. Another alternative is to bind the runtime to a service to enable background execution of the framework. Service and activity lifecycles are very similar, hence the incorporation process is very straightforward as well.

As far as the backend deployment for stationary devices running Java is concerned, the process is similar. An inspiration is provided in the **jdeeco-offloading-android-demo-java-backend** module. Its dependencies must be almost the same, except the **jdeeco-offloading-android-library** module, which is required solely for Android applications. The implementation consists of simple offloading framework initialization and deploying the same offloadable back-

end components used in the Android application (actually, the implementation may differ, but the interface and backend ID must stay identical). The runtime is instructed to deploy only backend-related components by setting the *MODE_ONLY_BACKEND* flag when initializing the *OffloadingManager*.

Chapter 9

Related work

Since the field of ensemble-based component systems and DEECo component model are relatively young, there is, to our knowledge, currently no other proposed architecture combining DEECo model or any other EBCS with the notion of cloudlets in form of mobile computation offloading. The DEECo component model is currently still in active development stage and this work represents one of its first real-life applications.

The DEECo control layer of the reference architecture can be seen as a node management system within a cluster. From this point of view, a relatively similar goals are addressed by the Red Hat Enterprise Linux Cluster Suite (RHEL Cluster Suite)[15]. It is a software providing a way to create node clusters with focus on load balancing and high availability. RHEL Cluster Suite takes care of managing the nodes, watching over their current availability and migrating tasks and roles between them to maintain high availability at all times. However, RHEL Cluster is designed for server deployment and it is certainly not suited for mobile deployment at all.

Other related work can be found when breaking the architecture down to two separate concepts: ensemble-based component systems and mobile computation offloading. As far as the EBCS are concerned, one recent work can be considered as related - the HELENA[28] approach. HELENA stands for Handling massively distributed systems with ELaborate ENsemble Architectures. It is an EBCS, which sees ensembles as goal-oriented communication groups of components. The ensemble's functionality is described by roles which can be dynamically adopted by the components. This approach addresses similar problematics as DEECo - managing highly dynamic distributed environments. The difference lies mainly in the focus on components' roles: "In contrast to the other component models Helena is centered around the notion of a role which allows to focus only on those capabilities of a component that is actually needed in a particular collaboration"[28].

Regarding mobile computation offloading managed by other systems, there are many frameworks currently available designed for such purpose. The Cuckoo framework[30] is one of them. It is a practical implementation of a mobile offloading framework brought to the Android platform. It provides a way to develop an offloadable application with a dynamic runtime system, that can dynamically decide whether a part of an application will be executed locally or offloaded to a remote node. Besides Cuckoo, other approaches were presented in the past: MAUI[24], Scavenger[32], ThinkAir[31] and more. However, using EBCS (in form of DEECo) as the management layer brings additional features to computation offloading for situations targeted by this thesis - high dynamicity, robustness, automatic discovery capabilities, etc.

Chapter 10

Conclusion

10.1 Achieved goals

The master thesis explored a possibility of using DEECo component model as the control layer for creating cloudlet-like environment for computation offloading technique allowing mobile devices to save its limited resources such as battery life. The situation was firstly analyzed, potential problems were identified and possible solutions were discussed. Later, a reference architecture for cloudlet offloading was designed and discussed. The architecture was then realized by implementing an offloading framework supporting Android platform for mobile devices and Java platform for stationary devices. Presented issues were addressed and possible alternatives discussed. As a part of the solution, two Android applications were developed to demonstrate and test the framework. One was used when developing and fine-tuning the framework and the other represents one of the use cases suitable for applying computation offloading - an OCR (text recognition) application. It is able to offload the OCR tasks to any other Android device or stationary device running Java. Finally, in the evaluation part, the outcomes of using the offloading framework were presented and the results clearly prove that it can save a significant amount of device's resources (mainly battery life).

10.2 Future work

The presented solution introduces a number of future work opportunities. For example, the scope of this thesis could not cover implementing a code migration mechanism for backend deploying. Implementing dynamic code loading could move the framework to a higher level because new mobile applications could be dynamically offloaded to a machine without manual backend deploying.

Furthermore, backend advanced load-balancing and backend state data caching mechanism are definitely places where future improvement is possible.

Another possible way of extending presented work is to apply the offloading mechanism on an application with large user base to test it in a truly large-scale dynamic environment.

Finally, some issues regarding the jDEECo framework, which were observed during the work (such as missing component live-checking), could be subjects for future work.

Chapter 11

Attachments

Enclosed DVD / Attached ZIP file contains following materials:

- Source code of implemented offloading framework
- JAR files containing compiled framework and its dependencies
- Generated development documentation for the framework (JavaDoc)
- Source code of implemented demonstration applications
- Installable APK files of both demonstration applications
- PDF version of this thesis and all images used inside
- Reports gathered from the evaluation process
- README files containing additional instructions for particular modules

List of Abbreviations

ADB	Android Debug Bridge
AOT	Ahead-of-time
API	Application Programming Interface
APK	Android application package
ART	Android Runtime
DEECo	Dependable Emergent Ensembles of Components
DSL	Domain Specific Language
EBCS	Ensemble-based component systems
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JIT	Just-in-time
MIME	Multipurpose Internet Mail Extensions
NFP	Non-Functional Properties
OCR	Optical Character Recognition
PLCS	Parking Lot/Charging Station
POIs	Points Of Interest
RDS	Resilient Distributed Systems
REST	Representational State Transfer
RHEL	Red Hat Enterprise Linux

SDK	Software Development Kit
UDP	User Datagram Protocol
UI	User Interface
XML	Extensible Markup Language

List of Figures

List of Tables

Bibliography

[1] Android BroadcastReceiver documentation. http://developer.android.com/reference/android/content/BroadcastReceiver.html. Accessed: 2015-02-25.

[2] Android Knopflerfish OSGi. https://www.knopflerfish.org/releases/5.1.0/docs/android_dalvik_tutorial.html. Accessed: 2015-02-25.

[3] Android SDK. http://developer.android.com. Accessed: 2015-02-25.

[4] Android Tools: Gradle Plugin User Guide. http://tools.android.com/tech-docs/new-build-system/user-guide. Accessed: 2015-02-25.

[5] Android™. http://android.com. Accessed: 2015-02-25.

[6] Elijah: Cloudlet-based Mobile Computing. http://elijah.cs.cmu.edu. Accessed: 2015-03-13.

[7] Google Guava EventBus. https://code.google.com/p/guava-libraries/wiki/EventBusExplained. Accessed: 2015-04-01.

[8] Greenrobot EventBus GitHub page. https://github.com/greenrobot/EventBus. Accessed: 2015-04-01.

[9] GSMArena. http://www.gsmarena.com. Accessed: 2015-04-06.

[10] Jersey for Android. https://github.com/giorgio-zamparelli/jersey-android. Accessed: 2015-03-30.

[11] Jersey Framework. https://jersey.java.net. Accessed: 2015-03-30.

[12] Leptonica. http://www.leptonica.com. Accessed: 2015-04-03.

[13] NanoHttpd Github page. https://github.com/NanoHttpd/nanohttpd. Accessed: 2015-03-30.

[14] Otto GitHub page. `https://github.com/square/otto`. Accessed: 2015-04-01.

[15] Red Hat Enterprise Linux Cluster Suite. `http://www.linuxjournal.com/article/9759`. Accessed: 2015-04-14.

[16] Restlet. `http://restlet.com`. Accessed: 2015-03-30.

[17] RFC1341 - The Multipart Content-Type. `http://www.w3.org/Protocols/rfc1341/7_2_Multipart.html`. Accessed: 2015-04-01.

[18] Tess-Two GitHub page. `http://www.leptonica.com`. Accessed: 2015-04-03.

[19] Tess4J homepage. `http://tess4j.sourceforge.net`. Accessed: 2015-04-03.

[20] Tesseract OCR. `https://code.google.com/p/tesseract-ocr`. Accessed: 2015-04-03.

[21] Lubomir Bulej, Tomas Bures, Vojtech Horky, and Jaroslav Keznikl. Adaptive deployment in ad-hoc systems using emergent component ensembles: Vision paper. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 343–346, New York, NY, USA, 2013. ACM.

[22] Gerostathopoulos I. Hnetynka P. Keznikl J. Kit M. Plasil F. Bures, T. Autonomous components in dynamic environments. *Awareness: Self-Awareness in Autonomic Systems Magazine*, pages 249–252, September 2012. `http://www.awareness-mag.eu/pdf/004415/004415.pdf`.

[23] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. Deeco: An ensemble-based component system. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, pages 81–90, New York, NY, USA, 2013. ACM.

[24] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[25] D3S, Charles University in Prague. CDEECo website. `https://github.com/d3scomp/JDEECo`. Accessed: 2015-02-25.

[26] D3S, Charles University in Prague. jDEECo website. `https://github.com/d3scomp/JDEECo`. Accessed: 2015-02-25.

[27] Dejan Kovachev and Ralf Klamma. Framework for computation offloading in mobile cloud computing. *International Journal of Interactive Multimedia and Artificial Intelligence*, 1(7):6–15, 2012.

[28] Rolf Hennicker and Annabelle Klarl. Foundations for ensemble modeling–the helena approach. In *Specification, Algebra, and Software*, pages 359–381. Springer, 2014.

[29] IBM. RESTful Web services: The basics. `https://www.ibm.com/developerworks/webservices/library/ws-restful`. Accessed: 2015-03-23.

[30] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a computation offloading framework for smartphones. In *Mobile Computing, Applications, and Services*, pages 59–79. Springer, 2012.

[31] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[32] Mads Darø Kristensen. Scavenger: Transparent development of efficient cyber foraging applications. In *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 217–226. IEEE, 2010.

[33] K. Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, April 2010.

[34] Zhiyuan Li, Cheng Wang, and Rong Xu. 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 '01, pages 238–246, New York, NY, USA, 2001. ACM.

[35] Andy Piper. Android and OSGi. `http://www.osgi.org/wiki/uploads/CommunityEvent2010/OSGi%20Community%2010%20-%20Piper.pdf`. Accessed: 2015-02-25.

[36] Aki Saarinen, Matti Siekkinen, Yu Xiao, Jukka K. Nurminen, Matti Kemppainen, and Pan Hui. Can offloading save energy for popular apps? In

Proceedings of the Seventh ACM International Workshop on Mobility in the Evolving Internet Architecture, MobiArch '12, pages 3–10, New York, NY, USA, 2012. ACM.

[37] M. Satyanarayanan. A brief history of cloud offload. *GetMobile*, 18(4):19–23, Oct 2014.

[38] Mahadev Satyanarayanan, P. Bahl, R Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, Oct 2009.