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.

1.3 Problems and challenges

TODO?

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.”[9]

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 [9]. 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”[9]. 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 [9] 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 [9], 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

```
interface AvailabilityAggregator:
    calendar, availabilities

interface AvailabilityAwareParkingLot:
    position, availability

component Vehicle features AvailabilityAggregator:
    knowledge:
        batteryLevel = 90%,
        position = GPS(...),
        calendar = [ POI(WORKPLACE, 9AM-1PM), POI(MALL, 2PM-3PM), ... ],
        availabilities = [ ],
        plan = {
            route = ROUTE(...),
            isFeasible = TRUE
        }
    process computePlan:
        in plan.isFeasible, in availabilities, in calendar, inout plan.route
        function:
            if (!plan.isFeasible)
                plan.route <- Planner.computePlan(calendar, availabilities)
        scheduling: triggered( changed(plan.isFeasible) OR changed(availabilities) )
    process checkPlanFeasibility:
        in plan.route, in batteryLevel, in position, out plan.isFeasible
        function:
            plan.isFeasible <- Planner.isFeasible(plan.route, batteryLevel, position)
        scheduling: periodic( 5000ms )

component PLCS features AvailabilityAwareParkingLot:
    knowledge:
        position = GPS(...),
        availability = ...
    process observeAvailability:
        out availability
        function:
            availability <- Sensors.getCurrentAvailability()
        scheduling: periodic( 2000ms )
```

Figure 2.1: Example of DEECo component definitions in a DSL (Source: [9])

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

Figure 2.2: Example of DEECo ensemble definition in a DSL (Source: [9])

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[11] 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, 19]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.

```
@Component
public class Vehicle {
    public List<CalendarEvent> calendar;
    public Plan plan;
    public EnergyLevel batteryLevel;
    public Map<ID, Availability> availabilities;
    public Position position;

    public Vehicle() {
        // initialize the initial knowledge structure reflected by the class fields
    }

    @Process
    public static void computePlan(
        @In("plan.isFeasible") @Triggered Boolean isPlanFeasible,
        @In("availabilities") @Triggered Map<...>availabilities,
        @In("calendar") List<CalendarEvent> calendar,
        @InOut("plan.route") Route plannedRoute
    ) {
        // re-compute the vehicle's plan if its infeasible
    }

    @Process
    @PeriodicScheduling(5000)
    public static void checkPlanFeasibility(
        @In("plan.route") Route plannedRoute,
        @In("batteryLevel") EnergyLevel batteryLevel,
        @In("position") Position position,
        @Out("plan.isFeasible") OutWrapper<Boolean> isPlanFeasible
    ) {
        // determine feasibility of the plan
    }
    ...
}
```

Figure 2.3: Example of jDEECo component (Source: [9])

- **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.[19] 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 computational 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.[15]

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.[18] 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 computational 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.[18] 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.[12, 17, 14, 15, 18]

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.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  package="cz.kinst.jakub.example.app">

  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.GET_ACCOUNTS"/>
  <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

  <uses-feature android:name="android.hardware.camera" android:required="true"/>

  <application android:name=".ExampleApplication"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AsparagusTheme">

    <activity android:name=".ui.activities.MainActivity"
      android:icon="@drawable/ic_launcher"
      android:logo="@drawable/logo_font"
      android:label="@string/app_name">
      <!-- This activity is the default activity listed in the app drawer on the device. It provides an entry point for the user. -->
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
    <activity android:name=".ui.activities.SettingsActivity"
      android:icon="@drawable/ic_asparagus"
      android:label="@string/settings">
    </activity>
    <receiver android:name=".sync.GcmBroadcastReceiver"
      android:permission="com.google.android.c2dm.permission.SEND">
      <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE"/>
        <category android:name="cz.kinst.jakub.example.app"/>
      </intent-filter>
    </receiver>
    <service android:name=".sync.SyncAdapterService"
      android:exported="true">
      <intent-filter>
        <action android:name="android.content.SyncAdapter"/>
      </intent-filter>
      <meta-data android:name="android.content.SyncAdapter"
        android:resource="@xml/sync_adapter"/>
    </service>
  </application>
</manifest>
```

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

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 on 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.

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

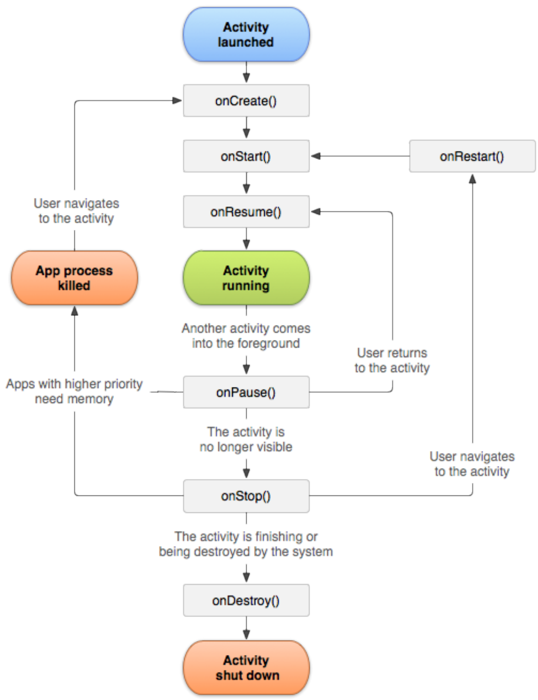

Figure 2.5: Android activity lifecycle diagram. (Source: `http://developer.android.com/reference/android/app/Activity.html`)

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.

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

Using EBCS as a control layer for computation offloading of mobile applications was proposed in [7]. Particularly, this paper 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. The proposed architecture can be used as a rough base for designing the reference architecture, but as it 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 us with 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.

The paper then generalizes 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".[7] 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 representing its deployment alternatives on different devices. It is responsible for calculating *NFPData* for that particular alternative. The Monitor

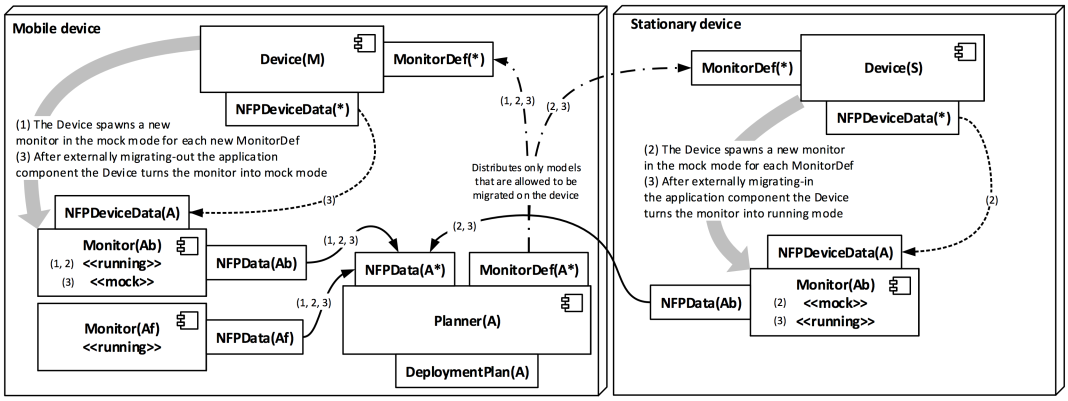

Figure 3.1: Adaptation architecture of the running example: phases 1 (M isolated), 2 (S discovered), and 3 (Ab migrated to S). Phases 1,2,3 are in the figure denoted by (1), (2), (3). (Source: [7])

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 [7], 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.

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.

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

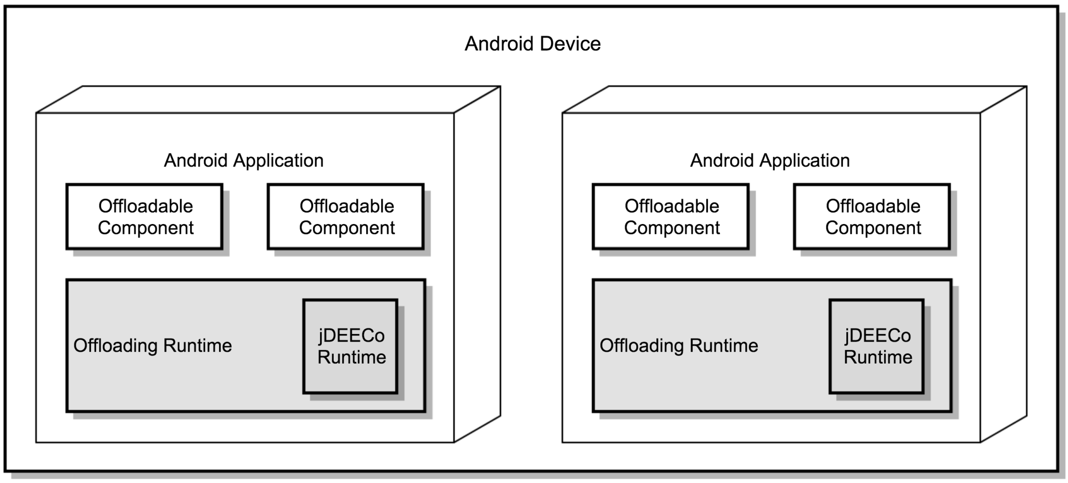

Figure 3.2: Each applications with its own offloading runtime

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

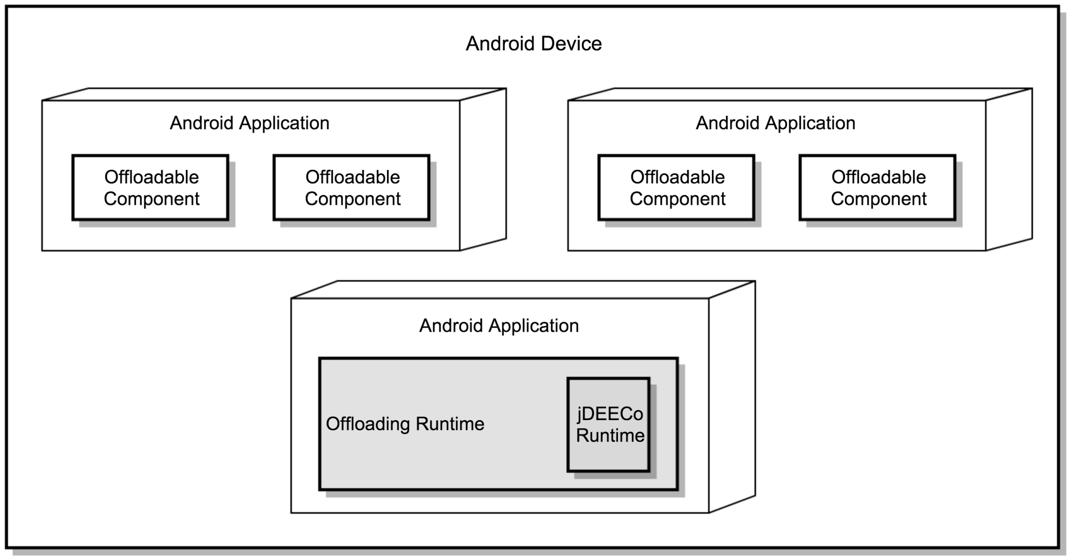

Figure 3.3: Each applications with its own offloading runtime

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

3.2.9 Presenting a reasonable demonstration application

Wise choosing a demonstration scenario for proposed reference architecture is undoubtedly challenging as well. The demonstration application should be 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 example of making 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 offloaded.

Chapter 4

Reference Architecture

4.1 DEECo inter-device communication

To allow deploying DEECo components on different nodes in a network and form ensembles across the nodes, some sort of network communication has to be incorporated to 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 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 incorporated in a 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 on each of the nodes, but with the same outcome on each 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 encapsulated components with a common interface, which provides interaction via network. In the reference architecture, an HTTP server is used as the interface. In fact, each offloadable component is represented by a RESTful[13] 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 different node (knowing its IP address) in a very straightforward way, which is almost transparent from 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 using the component. The server is able to identify the client when it is calling one of the component's methods and pick the right state from the 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 on all required nodes 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 should the client mobile application use for the actual execution. In other words, which node in the network to use for the computation offloading.

The control layer is represented by a runtime which is 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 control components are mirroring components on the application level. The communication between the layers is restricted so that only corresponding components should talk to each other.

4.3.1 Application-level components

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 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 the state data to the backend components in ACTIVE state.

4.3.2 DEECo-level components

Each device 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 Backend-Monitor is responsible for periodic gathering of the *NFPData* - the value representing potential performance of the particular alternative (component X running on device Y). I 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).

StateDataMonitor A generic Monitor deployed for each BackendStateData component linking it to the DEECo control layer. It is also responsible for pulling the latest state data from the currently active Backend component to keep the local copy 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 alternative 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. Also Frontend-Monitor and StateDataMonitor (for each Backend definition) components are 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 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 it has at the moment. The

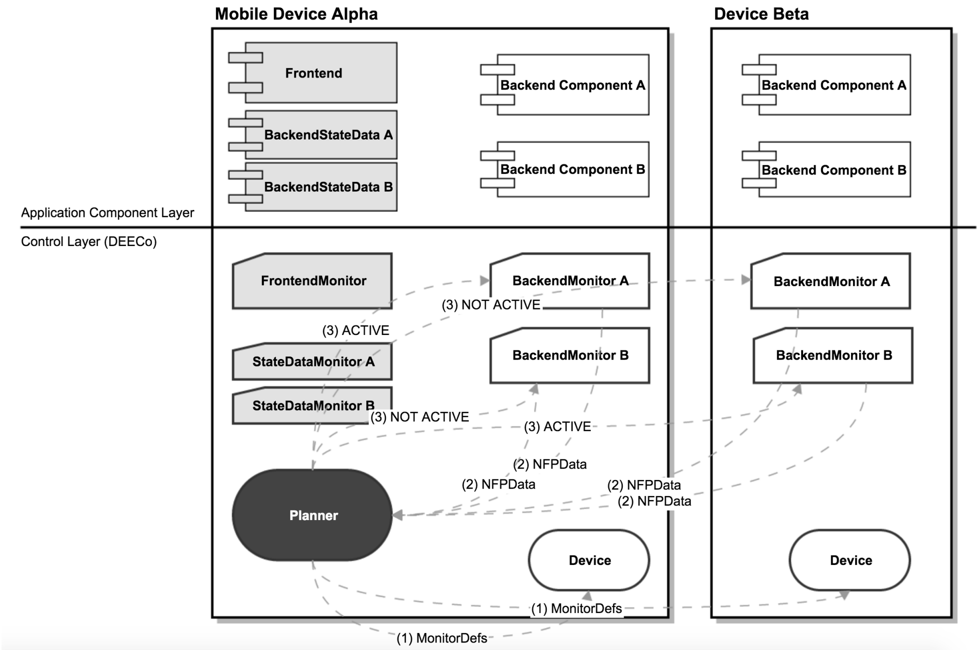

Figure 4.1: The first three steps of the control cycle (dashed line: ensemble knowledge exchange; solid line: inter-layer communication)

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 the 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 pass the new deployment plan to the application user interface via special Frontend component on the application level.

4.3.5 Timing constants

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 at all times. This

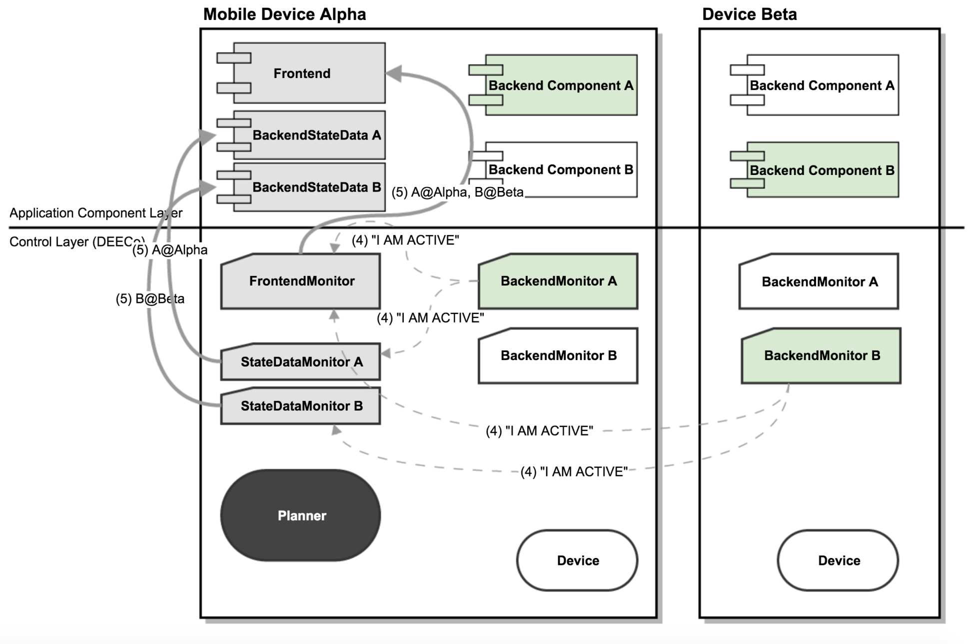

Figure 4.2: Steps 4 and 5 of the control cycle (dashed line: ensemble knowledge exchange; solid line: inter-layer communication)

means that parts of the system may not have the latest data available at all times, but eventually, everything gets always 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, since the DEECo model is designed to work similar to this itself (belief based behavior).

Therefore an important challenge is setting of the particular periods. They 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.

However, 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.

4.3.6 Putting it all together

TODO:

Chapter 5

Implementing the Reference Architecture

5.1 Structure of the solution

TODO:

5.2 Used build system and frameworks

TODO:

5.3 Offloadable components

TODO:

5.4 DEECo control layer

TODO:

5.5 Platform-specific code

TODO:

5.6 Problems and solutions

TODO:

5.6.1 Handling sudden node disconnection

TODO:

5.6.2 "Zombie" components

TODO:

5.6.3 Transitioning between networks

TODO:

5.6.4 Multi-application environment

TODO:

5.7 Compatibility

TODO:

Chapter 6

Demonstration Applications

TODO:

Chapter 7

Evaluation

TODO:

Chapter 8

Making an Android Application Offloadable

TODO:

Chapter 9

Related work

TODO:

Chapter 10

Conclusion

TODO:

10.1 Achieved goals

TODO:

10.2 What's next

TODO:

Chapter 11

Attachments

TODO: attached DVD

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
EBCS	Ensemble-based component systems
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JIT	Just-in-time
PLCS	Parking Lot/Charging Station
POIs	Points Of Interest
RDS	Resilient Distributed Systems
REST	Representational State Transfer
SDK	Software Development Kit
UDP	User Datagram Protocol
UI	User Interface
XML	Extensible Markup Language

List of Figures

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] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

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

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

[15] 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.

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

[17] 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.

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

[19] 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.