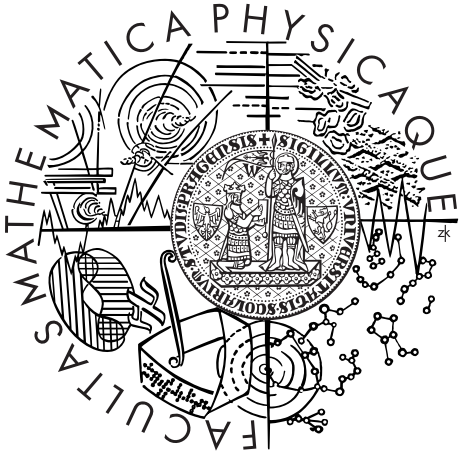
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Daniel Sipták

DEECo Component Model Framework on Android Mobile Platform

Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. RNDr. Tomáš Bureš, Ph.D.

Study programme: Informatics

Specialization: System architecture

Prague 2014

I would like to express my gratitude to my supervisor Tomáš Bureš for the useful comments, remarks and engagement through the process of this master thesis. Furthermore I would like to thank Ilias Gerostathopoulos and Jaroslav Keznikl for support on the way as also to whole development team of jDEECo software.

I would like to thank my loved ones, who have supported me throughout entire process kept me on the track and gave me time to create this master thesis.
I will be grateful forever to my wife and son for their love.

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 ..... date ..... signature of the author

Název práce: Framework pro DEECo komponentní model pro platformu Android
Autor: Daniel Sipták
Katedra: Katedra distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: doc. RNDr. Tomáš Bureš, Ph.D., Katedra distribuovaných a spolehlivých systémů návrh Abstrakt: Prezentována diplomová práce se věnuje tvorbě podpory DEECo komponentního modelu na platformě Android. Vytvorění distribuovaného systému schopného používat DEECo framework nad více zařízeními. Pro tento účel je využita jDEECo implementace DEECo komponentového modelu, která je portována na platformu Android. Návrh řešení synchronizace společného stavu je navržen a implementován za pomoci JGroups knihovny. Prezentovány jsou možná řešení a implementace vytvořeného produktu. Ako poslední je ukázána demo aplikace, na které je vidět použití funkcí vytvořeného frameworku.
Klíčová slova: DEECo, Android, JGroups, Komponentové systémy

Title: DEECo Component Model Framework on Android Mobile Platform
Author: Daniel Sipták
Department: Department of Distributed and Dependable Systems
Supervisor: doc. RNDr. Tomáš Bureš, Ph.D., Department of Distributed and Dependable Systems
Abstract:
Presented master thesis is dedicated to creation of DEECo component model supported on Android platform. Enabling distributed system of inter-connected devices to run DEECo framework. For this purpose jDEECo implementation of DEECo component model is ported to Android platform and synchronization solution creating common state is done on top of JGroups toolkit. Possible solutions are presented and implementation of created solution is described. At last demo application showing usage of created framework was developed and evaluated.
Keywords: DEECo, Android, JGroups, Component systems

Contents

1. Introduction

With increasing number, complexity and connectivity of computing devices there are new possibilities to use them in completely new ways. Addressing social and environmental challenges like smart city infrastructure, environmental monitoring, smart electricity grid, emergency coordination could be done by creation of large-scale Resilient Distributed Systems (RDS) [1]. As RDS are responding and influencing ever changing environment they have to cope with very dynamic environment. To overcome these challenges they are design to be highly autonomous and adaptive. Most of the complexity for creating of such systems is not in building new hardware or addressing network infrastructure but in designing software.

Part of current research is dealing with RDS challenges by identifying new subclass of existing component-based software architectures. This new architecture called Ensemble-Base Component System (ECBS)[1] is specifically created to address RDS. One of the proposed solution is DEECo (Distributed Emergent Ensembles of Components) component model[1, 2, 3]. This model is employing several topics from another research areas namely, component-based software engineering, agent-oriented computing, ensemble-oriented systems and control system engineering. For ECBS architecture and specifically DEECo these are most important features.

- Architecture of the system emerges at runtime.
- Runtime framework is managing belief maintained for each Component.
- Component is executed in fully isolated way only using it's belief.

In this thesis we will try to improve current pilot implementation of DEECo called jDEECo[1]. This implementation is supporting most of the features required by ECBS system. But it is missing multi node architecture support. Implementation is supporting multiple instances by using Apache River Technology. This is creating abstraction of single point where all beliefs are hold. To enable mobility testing we create implementation of DEECo on Android platform. So

nodes will be running on different devices which could connect or disconnect during usage of the software.

Main goals of thesis are

- **Design and implement DEECo component model on Android**
  Implement main ideas of DEECo component model under Android environment[android]. Support basic ideas for components, ensembles and knowledge exchange. If possible reuse code from jDEECo project which is based on Java and should be portable to Android.

- **Support background processing on Android**
  Solution needs to run in background without active user interface. This way DEECo framwork can be running under normal operation of the device. Also take account of resource consumption specific to mobile devices.

- **Support inter device communication and synchronization**
  Create solution supporting multi device communication with possibilities of adding new nodes, removing nodes and merging at least two separate groups of nodes. This will add mobility support which could be used for testing DEECo model in very dynamic environment with network failures and sporadic inter node communication.

- **Create demo application**
  Created demo application with full graphical interface. Main aim is to have application registering and reacting to the changes in DEECo component model it is using. Application doesn't need to server any real usable function.

By creating DEECo implementation with multi node capabilities we open up space for mobility testing. Where nodes are connecting, re-connecting and disconnecting often. In this environment would be possible to test how DEECo component model behaves during such deployment. Also it will provide more complex solution for RDS systems.

Next chapter Background (5) presents and describes theoretical basis of DEECo component model and also fundamental design of Android platform. After this more theoretical chapter Analysis (11) follow. Where we will introduce main problems of background processing on Android and multi-node implementation of DEECo component mode. There we also propose and explain solutions to those problems. Next two chapters (30,33) are devouted to implemetantion details and pitfalls encoutered during development. Also these chapters should improve clarity of understating the solutions from analysis chapter. In the testing chapter (41) we introduce ADEECo Cloud demo application together with few test results. At last we sumarize and evaluate our work in Conclusion chapter (47)

2. Background

In order to create DEECo component model on Android platform we need to review theoretical and technical solutions which are used. There are two separate parts and those are DEECo component model[1] and Android platform [4]. DEECo has it's pilot implementation in jDEECo project. Portation to Android and creation of multi-device support requires to create environment in Android application for DEECo component model.

2.1 Distributed Emergent Ensembles of Components

'DEECo is proposed as refinement of ECBS model which is tailored to be used in RDS systems. It is combining multiple research areas and creating one systematic approach for creating real-life software engineering procedures to build RDS systems. Two mains ideas of DEECo are components and ensembles. Component is self-sufficient unit of computation,development and deployment. And ensemble is dynamically formed group of components which manages inter-component communication. Idea behind creating self-contained components and managing their interaction with ensembles is separation of concerns. Runtime framework is providing all necessary services and controls life-time of components and ensembles.'[1]

More comprehensive description of components and ensembles is below.

Components and processes

In DEECo every component has it's state, interface used to access the state and processes which are manipulating the state. Knowledge reflects state of the component and it is represented as hierarchical data structure. This state is retrieved and stored by framework and the component processes are called with requested knowledge. Processes on top of components and their knowledge are basically soft real-time tasks that manipulate knowledge. Process can be

characterized as an function with input and output parameters which are in this case knowledge fields. Processes are scheduled and run by runtime framework which is also managing knowledge storage. There are two types of processes, scheduled and triggered. Scheduled process is run in soft-real time manner and triggered process is run once specified knowledge has been changed by another process.

Ensembles

An ensemble is dynamically created group of components in which one component assumes the role of coordinator and others the role of members. Thus ensemble is determining composition and interactions of the system. Runtime framework is determining which component is coordinator and which is member according to the membership condition defined by ensemble. As components do not explicitly communicate they use ensembles for indirect communication by using knowledge exchange.

Each ensemble comprises of

- **Membership condition**

  Condition which is evaluated with coordinator and each member and it detects if the member is part of the ensemble. If this coordinator/member pair is part of the ensemble than the knowledge exchange is done. Generally each component can be coordinator and member of multiple groups simultaneously.

- **Knowledge exchange**

  Knowledge exchange is declared as an function on top of coordinator/member pair. It encapsulates communication between components and in principle separating knowledge transfer from the components to the runtime framework. This relationship can be viewed as one-to-one in oppose to one-to-many in case of Membership condition.

In multi device environment processes of components are run on the device where the component was created. But membership condition and therefore

knowledge exchange is done on all devices. This feature is delivered by runtime framework and is creating implicit ensembles over all devices in the given ensemble.

Nodes mobility

Nodes mobility is defined as possibility for the node (device) to join, re-join and leave communication with other nodes. Each node can embody multiple components and use different set of ensembles. Where components are running on local knowledge and Membership function is determining composition of the system. Communication between the nodes is only through knowledge exchange and this exchange is also done on inter-node level. Runtime framework is managing synchronization of the knowledge between group of connected nodes. Change in knowledge by knowledge exchange, component process or by joining of another node are transfered to all nodes in the group which needs the information.

Knowledge repository

Knowledge repository is and abstraction layer in implementation of DEECo which servers as single point were all knowledge is kept. Interface whit transaction like functionality has only three basic operation on top of knowledge repository. Operation get,put and take. Additionally this abstraction layer provides possibility to alert about changes on top of knowledge.

2.2 Android Fundamentals

Android is an open source mobile operating system based on Linux kernel. It is primarily design to be used on touchscreen devices mainly smart phones and tablets.

Android application called apps are written in Java programing language. Which is compiled and packet together with any resources into one APK (Android Package). Each app has its own APK which is used by Android-powered devices to install the app.[4]

There are four types of app components. Which are essentially main building blogs of Android app.

Activities An activity represents single screen with a user interface. It represents main building block of most applications. And all user interactions with app is done by activities. More in sub-chapter about activit on page 9.

Services A service is a component of background processing. It can perform long-running processes or work on remote processing on the network. A service has no user interface and it is started by another components, such as Activities. More in sub-chapter about service on page 10.

Content providers A content provider manages a shared set of app data. It's main purpose is to store the data in file system, SQLite database on Web or on any other persistent storage.

Broadcast receivers A broadcast receiver is a component that responds to system-wide announcements like picture was captured, screen has been turned off or a battery is low. In most cases a broadcast receiver is an gateway for app to communicate with other apps in the system.

As android aims for mobile devices it supports multitude of network communication protocols and platforms. Like Wi-Fi, Cellular, Wi-Max, Wi-Fi Direct, Bluetooth or NFC. Most of these protocols and platforms creates connections

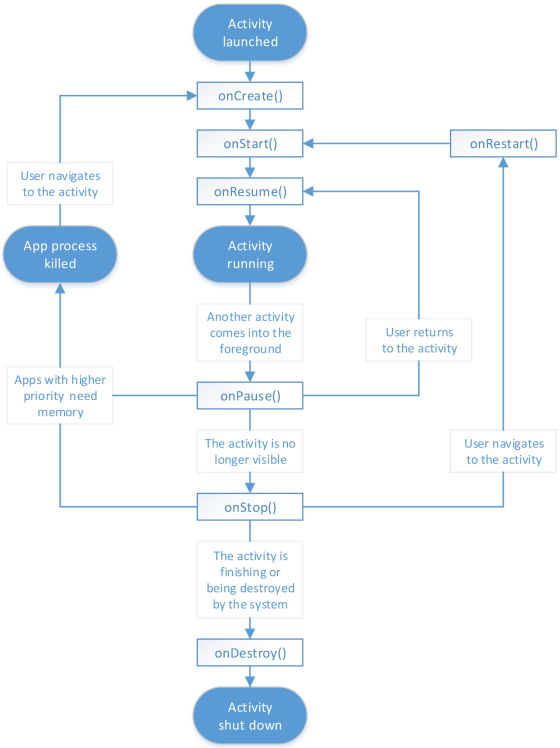


Figure 2.1: Activity life-cycle

enabling usage of Java standard network interfaces. In case of Wi-Fi Direct and Bluetooth technologies it is more about how the connection between devices is created than about special interface for communication.

In our apps we employ several activities and a single service. And therefore we give here more accurate description of these components.

Activity

Activity component used for user interaction is defined as subclass of activity class. All activities shares life-cycle which reflects usage of activities by the user and the system[4]. Once the activity is called onCreate() (2.1) method is fired up and creates graphical representation shown to the user. Activity is running as long as it is visible to user and if another activity goes to foreground activity is paused and later stopped. In case when memory is needed on the device whole activity is destroyed and that garbage collected by Java runtime.

Service

Service serves as an component enabling long-running tasks. On its behalf network communication can be managed. It can reside in same process as activities and it elevates the priority of that process as system would prefer to close proceses without services. Services are defined as an subclasses of Service class and has the same initialization procedures. Service can be either started as a long running or as an batch job. In later case service is closed after it finishes one specific action. In case of the long-running task, service will get possibility to control its own life-cycle.

More informations about Android Framework can be found on documentation pages of Android Project[4] .

2.3 Jgroups

Another software besides DEECo and Android which is used in this thesis is JGroups toolkit[5].

It is a multicast toolkit used for reliable messaging and cluster creation. It has wide range of uses and supports multitude of protocols for delivering customization. Abstraction created by JGroups on top of standard Java networking interface decreases complexity of networking applications. Precisely for this reason JGroups would be used in our thesis[5, 6].

3. Analysis

In this part of the thesis we will revisit our goals and analyze possible solutions to the raised issues.

3.1 Goals revisited

Design and implement DEECo model on Android

- **Managing component lifetime** Component is defined by the knowledge and processes it is using. Also it defines initial values of the components knowledge. Components creation is managed by runtime which initialize and run processes according to the specification. Starting of the components processes can be described as atomic loading of needed knowledge. Executing process functions with local copy of the knowledge and than atomic write of the changed knowledge.

- **Evaluating ensembles** Ensembles are defined as pairs of Membership condition and Knowledge Exchange functions. Runtime framework will evaluate membership condition on coordinator/member pair. If the pair is part of the ensemble it will do knowledge exchange. Both parts are executed separately from component processes and can be seen by components as atomic actions.

- **Knowledge repository** Runtime will store all knowledge defined by components in structure similar to tuple space[7]. For given key it can store whole complex objects. Interface of this part is supporting synchronization to get atomic like operations for other parts of framework. Most of the multi-node synchronization will be done on the knowledge repository.

To gain this functionality jDEECo implementation was re-used. With small changes it is possible to run runtime framework on top of Android Platform. More about porting jDEECo to Android can be found in JjDEECo sub-chapter on page 33.

Support background processing on Android

Created runtime needs to be run continuously and with Android application lifetime we need to create solution to keep framework running as long as possible. From the design of Android there is no way how to enforce that the process will not be stopped[4]. But several actions can be taken so the runtime will run in background and OS will not stop it until necessary.

As mentioned earlier on page 8 Android framework recognize two separate components an activity and a service. As a service is by design created for running in the background it is used to create background processing capability. Fitting DEECo on top of service with combination of activity opens up few different architectures.

- in-process architecture
- one-to-one architecture
- many-to-one architecture
- many-to-one separated architecture

Each of the proposed architectures have it's advantages as well as disadvantages. More complex architectures have possibility to support multiple application with only one DEECo runtime. This would improve local usage of the DEECo component model. But simpler ones have simpler development and don't have to use inter-process communication. Specification and comparison between suggested architectures is bellow. For initial implementation we have chosen simpler but effective solution of using one process for both activities and a service. More elaboration about possible architectures can be found in chapter Application architecture on page 13.

Support inter-device communication and synchronization

Implementation of inter-device communication require ability to discover nearby devices, initiate knowledge synchronization, replicate newly changed knowledge. Firstly we need to identify potential hardware channels for communication. Most promising are Wi-Fi, Wi-Fi Direct[8] and possibly Bluetooth. All are supported by Android Platform and in later two cases they would provide communication in nearby group. More on this in chapter Network communication channels on page 18.

3.2 Application architecture

Here we tackle the problem of application design. Specifically application architecture regarding split of the activity and the service into the different processes or applications. Some parts are given by the design of OS specifically that GUI is implemented in the activity component. And the DEECo runtime framework together with network handling is implemented under the service component.

Here we present four proposed architectures:

in-proccess architecture

This architecture on page 14 use just one process for service and activity handling. This is a standard usage on android platform if no special features are required. Activity of the process handles only GUI and pass all request for changes in DEECo runtime to the service. Service embodies runtime framework and is the main communication hub for the application. As there can be multiple activities in one application service will keep reference to the currently active one. In case that the application is running on the background ,activity might be removed by the OS and therefore most of the memory allocated to the activity should be released.

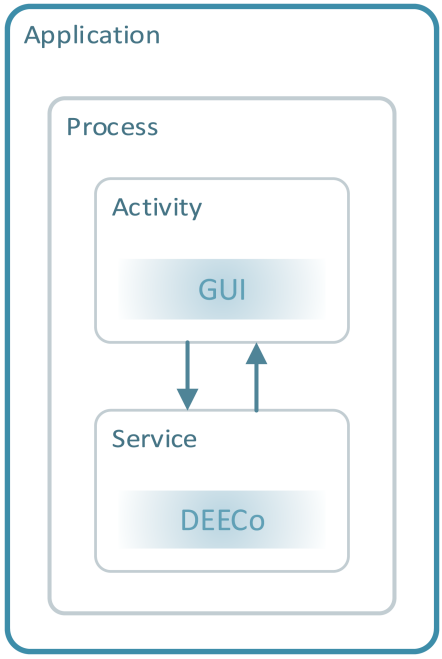


Figure 3.1: in-proccess

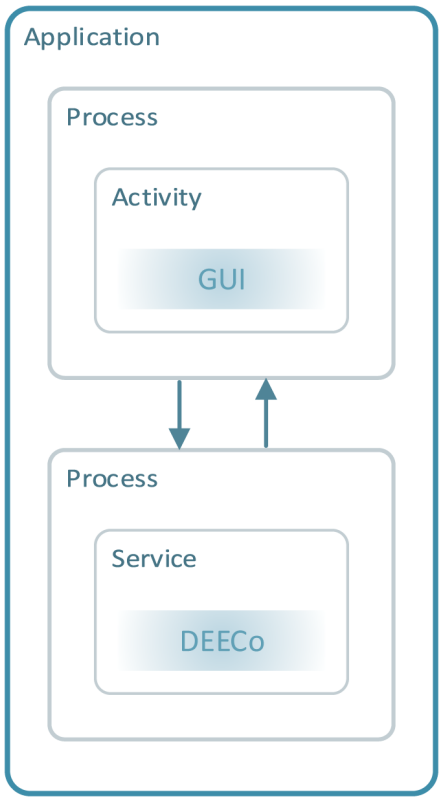


Figure 3.2: one-to-one

In this case communication between service and activity can be done implicitly in the code. The barrier dividing these two parts is not really visible which has the increased value of simpler development and easier reusing of the code. Communication model still has to account for a need of making changes in GUI only on main thread. Also usage of one process for the whole application increases the chance that it will be stopped by OS due to higher memory needs caused by usage of activity component in the same process as services implementation of DEECo framework.

one-to-one architectures

This case 3.2 is improved version of 'in-process' architecture 3.1. The main idea is to split the activity and therefore GUI handling more from the service and its DEECo runtime framework. Here the activity and the service uses different processes. So the process for the activity is running only as long as the application is directly used by the user. It will be removed completely once OS needs to gain more memory. But the service process which runs runtime framework will be

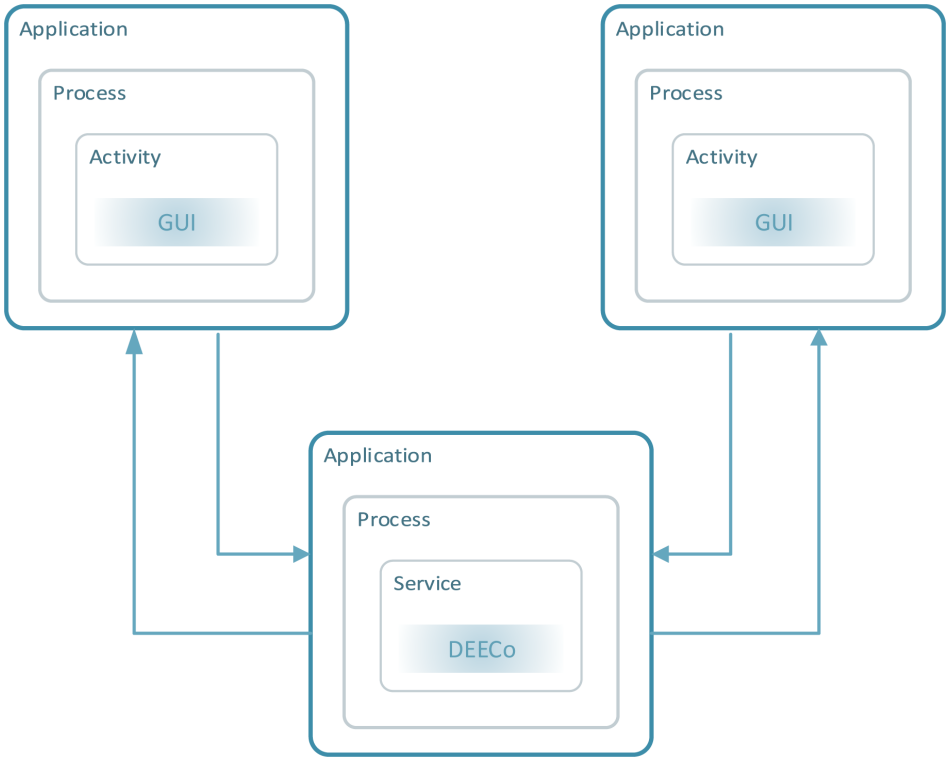


Figure 3.3: many-to-one

unaffected by the activity disappearance once it is started by the activity.

As the application has two running process one for active activity and one for the background service inter-process communication is used between them. This increases complexity for designing the application but will result in better separation of concerns, maintainability and cleaner interface between the two main parts.

many-to-one architectures

First two mentioned architectures were supporting only one application with DEECo runtime framework. But in case multiple application will be using the DEECo component model it would be beneficial to use just one DEECo runtime framework for all application. Proposed solution 3.3 creates this support by creating one common service for all application. Thanks to Android support with inter-application usage of services, there is a clean way how to implement this. Two separate types of applications are used.

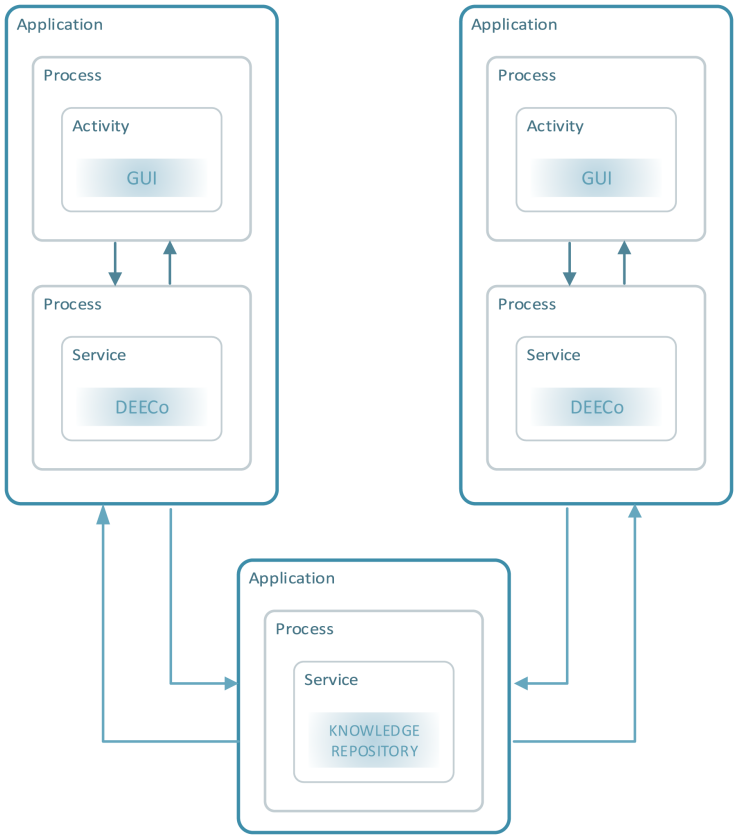


Figure 3.4: many-to-one separated

Main application creates the service which is holding DEECo runtime framework and this service is used by all other applications.
Standard application will hold GUI and its components and ensembles. Once standard application starts it will load its components and ensembles into the service where they will be handled by DEECo runtime framework. In this case even after the original application is removed from memory its components and ensembles will still run under main application service.

Most complexity of this architecture arises from the need of running applications components and ensembles under another application service. This requires dynamic loading of Java class code into the service which could be handled by OSGi platform. OSGi platform is working under Android but with very limited set of features and would account for a big increase in memory usage. Also service would have to handle multiple connections to activities and serve two side communication channels for them.

many-to-one separated architectures

This architectures 3.4 tries to solve problem with dynamic loading of code from many-to-one architecture. Here each standard application would have its own service which would be running DEECo runtime framework but knowledge repository would be stored only in one main service. This ensures that applications on the same node use only inter-process communication and not networking for knowledge transfer. Main service with the implementation of knowledge repository could be implemented as content provider. This is another component of the Android framework specifically design as a component storing data shared between multiple applications.

Major problem of this architecture is bigger memory footprint even than in-process architecture. As it needs one extra main service or content provider for common knowledge repository. Also it is using inter-process communication much more that in-process because network communication would be used only once something has changed in repository but in case of this architecture every read will terminate in inter-process communication.

In our project main goal is to run DEECo component model application across multiple devices and not multiple application on one device therefore we choses to use in-process architecture. This enables to easier development and even in case of multiple application on one device it is still fully functional.

3.3 Network communication channels

In order to provide inter-device connectivity we have researched possible communication channels. After research into multiple ways of connection we had to resolve only to use standard network connections because only those fulfills the needed requirements. The main problem of ah-doc style of communication is explicit need for user authorization on Android Platform.

Here we present the list of considered communication technologies and standards

Standard networking

By standard networking we mean cellular network, Wi-Fi, Wi-Max and Ethernet communication standards. All of these are almost indistinguishable from each other on Android Platform and so we gain all of them once we implement usage of networking API. The part that is quite different for these technologies is searching for available hosts and support for multi-cast communication. So different solutions for host discovery might be used on different networks. Main implementation focus is on Wi-Fi which enables connection to the Ethernet network.

Wi-Fi Direct

This is an ad-hoc variant of Wi-Fi technology which can be used to create small separate networks for inter-device communication. It offers same speeds as standard WiFi connection and also multi-cast support for the group of inter-connected devices. Usage of Wi-Fi Direct would add possibility to directly test mobility of the devices and behavior of DEECo component model under such environment. Where as the users would move with their mobile phones, they will connect to each other creating short-lived small networks. Which would be used by DEECo runtime to exchange knowledge of components running inside them. This would allow for mobility testing and would move DEECo implementation closer to ASCENT[9] e-mobility use-case study[].

However during investigation it was found that currently Android OS requires user action to initiate Direct connection. No possibility of remembering paired devices exists and therefore user action is required every time WiFi Direct discovery is started. This defeats main idea behind using WiFi Direct.

Bluetooth

In current version of Android 4.3 there is still missing full support for personal area network (PAN) profile of Bluetooth technology. This profiles enables creating of ad-hoc networks of nearby devices. There exists third party extensions which creates PAN support but as it not a standard part and it requires rooting the device our project will not use these third party extensions.

Current API for Bluetooth technology features possibility for inter-device communication on the required level but it is design to be used for one-time data exchange and not for frequent two-side communication. It would require to create protocol for knowledge exchange between at least two nodes interconnected by Bluetooth. After early stages of investigation latency of such connection was too high to be used in heavily ch nging environment as in case of DEECo. Usage of Bluetooth communicatio together with jDEECo implementation would require substantial changes in design of jDEECo. For these reasons and for that our goals are fulfilled by standard networking interfaces we do not include Bluetooth inter-device communication into the scope of the project.

NFC

Another technology currently implemented in Android for Inter-device communication is NFC (Near field communication). This technology is however not meant to create network between the devices but should be used for specific data exchange.

Although it should be possible to create protocol for knowledge exchange over this communication it is not in scope of this project.

3.4 jDEECo implementation

jDEECo implementation of DEECo component model is reused in this project and therefore we review here it's basic structure and what is needed to change in other to be used in our new framework. Whole jDEECo implementation will be a part of the service running in the background. Due to the reasons better explained in chapter on page 33 and mainly due to incompatibility of Android with Java 7 we used older version of the jDEECo framework. However main changes done to the older version are portable to the new generation of jDEECo.

On the example of initialization code of jDEECo we will show most important parts of the existing framework.

```
1 List<Class<?>> components = Arrays.asList(new Class<?>[]{
      NodeA.class,NodeB.class, NodeC.class });
2 List<Class<?>> ensembles = Arrays.asList(new Class<?>[]{
      MigrationEnsemble.class });
3 KnowledgeManager km = new RepositoryKnowledgeManager(
4         new LocalKnowledgeRepository());
5 Scheduler scheduler = new MultithreadedScheduler();
6 AbstractDEECoObjectProvider dop = new ClassDEECoObjectProvider(
      components, ensembles);
7 Runtime rt = new Runtime(km, scheduler);
8 rt.registerComponentsAndEnsembles(dop);
9 rt.startRuntime();
```

On the first two lines groups of components and ensembles are defined. In both cases the needed entities are defined as an classes. These classes are used by framework to initialize and run components and ensembles functions.

On the third and fourth line initialization of *KnowledgeRepository* together with decorated classes *RepositoryKnowledgeManager* and *KnowledgeManager* is done. This *KnowledgeRepository* holds all initialized knowledge and this part will be changed to provide synchronization with other devices. In this case only

LocalKnowledgeRepository is initialized which only holds locally stored *HashMap*. Which is used as an storage of knowledge.

In later lines supporting classes are initialized where *rt* is object holding whole DEECo runtime framework. On the eighth line registration of components and ensembles is done to the framework. This means that runtime will parse the classes for initial knowledge and processes of components as also for membership functions of ensembles.

On the implementation of *KnowledgeRepository* depends if,how and when the knowledge will be synchronized to another devices. In DEECo component model only knowledge of the components has to be accessible between nodes of the system. That means that information about components,ensembles, component processes, membership functions used by runtime are only local and does not propagate to another nodes. This in term defines that only synchronization on the level of knowledge is needed. Knowledge can be be access, changed or removed and this API is used for these purposes in jDEECo.

```
1  public interface IKnowledgeRepository {
2      /* get knowledge with entryKey identificator under defined session
           */
3      public Object [] get(String entryKey, ISession session)
4              throws KRExceptionUnavailableEntry, KRExceptionAccessError;
5
6      /* Alter knowledge with entryKey identificator */
7      public void put(String entryKey, Object value, ISession session)
8              throws KRExceptionAccessError;
9
10     /* Return and remove knowledge from the repostory */
11     public Object [] take(String entryKey, ISession session)
12             throws KRExceptionUnavailableEntry, KRExceptionAccessError;
13
14     /* Register listener with the callback once knowledge change */
15     public boolean registerListener(IKnowledgeChangeListener
```

```
16      listener);
17
18      /** Session is used for transaction like acess to the repository */
19      public ISession createSession();
20  }
```

Full implementation of this interface with synchronization to another nodes will in affect allow inter-node knowledge exchange. Therefore it brings multi-node support for the jDEECo runtime. This is done on current implementation of jDEECo with the use of Apache River technology. Which is creating single point of storage for all knowledge regardless of the node which uses the knowledge.

For explanation and illustration we use example DEECo application. Our application is derived from cloud demo application fro original jDEECo implementation. It is simple as each node consist only from one component and two ensemble processes.

Node component definition:

```
1   @DEECoComponent
2   public class Node extends ComponentKnowledge {
3
4       public Float loadRatio;
5       public String targetNode;
6
7       @DEECoInitialize
8       public static ComponentKnowledge getInitialKnowledge() {
9           Node k = new Node();
10          k.id = "Node_"+getUUID(); --universaly unique value
11          k.loadRatio = 0.0f;
12          k.targetNode = null;
13          return k;
14      }
15
16      @DEECoProcess
```

```
    @DEECoPeriodicScheduling(3000)
    public static void process(@DEECoIn("id") String
        id,@DEECoInOut("loadRatio") OutWrapper<Float> loadRatio) {
        loadRatio.item = new Random().nextFloat();
    }
}
```

This code defines component named node. Knowledge of this component consist of two values *loadRatio* and *targetNode*. Component also defines one periodically scheduled process which updates *loadRatio* value.

Next is simple ensemble function which creates and alert once any component reaches threshold value of *loadRatio*.

```

@DEECoEnsemble
@DEECoPeriodicScheduling(3000)
public class AlertEnsemble extends Ensemble {

    @DEECoEnsembleMembership
    public static boolean membership(
            @DEECoIn("coord.id") String cId,
            @DEECoIn("member.id") String mId
            @DEECoIn("member.loadRatio") Float mLoadRatio) {
        return cId.equals(mId) && mLoadRatio > 0.7f;
    }

    @DEECoEnsembleMapper
    public static void map(@DEECoIn("member.id") String mId,
            @DEECoIn("member.loadRatio") Float loadRatio) {
        System.out.println(mId + " overloaded with " +
            Math.round(loadRatio *
        100) + "%");
    }
}
```

In this case *DEECoEnsembleMembership* process is evaluated every three seconds. In case that *mLoadRatio* is higher than 0.7f (70%) it will create ensemble on top of this member. This ensemble will run *DEECoEnsembleMapper* process which prints an alert about two high load ratio for member component.

```
1  @DEECoEnsemble
2  @DEECoPeriodicScheduling(2000)
3  public class MigrationEnsemble extends Ensemble {
4
5      @DEECoEnsembleMembership
6      public static boolean membership(
7              @DEECoIn("member.id") String mId,
8              @DEECoIn("member.loadRatio") Float mLoadRatio,
9              @DEECoIn("coord.id") String cId,
10             @DEECoIn("coord.loadRatio") Float cLoadRatio) {
11         return !mId.equals(cId) && mLoadRatio > 0.7f && cLoadRatio <
               0.7f; }
12
13     @DEECoEnsembleMapDEECoEnsembleMapperper
14     public static void map(@DEECoIn("member.id") String mId,
15             @DEECoIn("coord.id") String cId,
16             @DEECoOut("member.targetNode") OutWrapper<String>
                   mTargetNode) {
17         mTargetNode.item = cId;
18         System.out.println("Balance load from "+mId+" to " + cId);
19     }
20 }
```

Another more complex ensemble in our cloud examples is *MigrationEnsemle*. *DEECoEnsembleMembership* process defines that if member has load over 70% and coordinator has node lower than 70% it will run *DEECoEnsembleMapper process*. This process will set that member's target node will be set to coordinator id.

This example is a simple application triggering load balancing between multiple nodes. As value targetNode is meant as instruction for application offloading

from current node to node set in this knowledge. We show it here to illustrate main ideas and usage in our implementation of DEECo.

3.5 Multinode comunication

In order to add multi-node functionality to our application we have analyzed ways how to exchanged knowledge. From the parts above we learned that we need to support only basic map like interface (get,put,take). Combined with standard networking we need to develop ways how to find another available nodes in the network. How to connect to them and exchange knowledge information. How to keep exchanging the knowledge as long as the other site is available. Current implementation of jDEECo has requirement that for each component knowledge only one process is changing the value of this knowledge. That means that each knowledge has one writer and multiple readers. One process in whole application should be changing the value of each knowledge. Only exception is creation of the knowledge done by initial knowledge setup done by framework.

For node discovery we can employ several techniques like broadcast discovery, multicast discovery or usage of directory service. Each of these can be used in our environment and has different set of advantages and disadvantages. As broadcast discovery is usable only in one network and directory service introduce single point of failure we have chosen to support multicast discovery. Another advantage is that multicast can be used not only for discovery but also for knowledge exchange. Main purpose of the node discovery is to find all reachable nodes and connect them to an group of nodes. This group of nodes will be regarded as an cluster.

In case of unicast communication sending node has to send as many packtes as is the number of connected nodes 3.5. In case of multicas sender sends on one packet and it is duplicated on the network. This improves scalability of the cluster of nodes. In case of single network segment ususaly broadcast can be used as replacement.

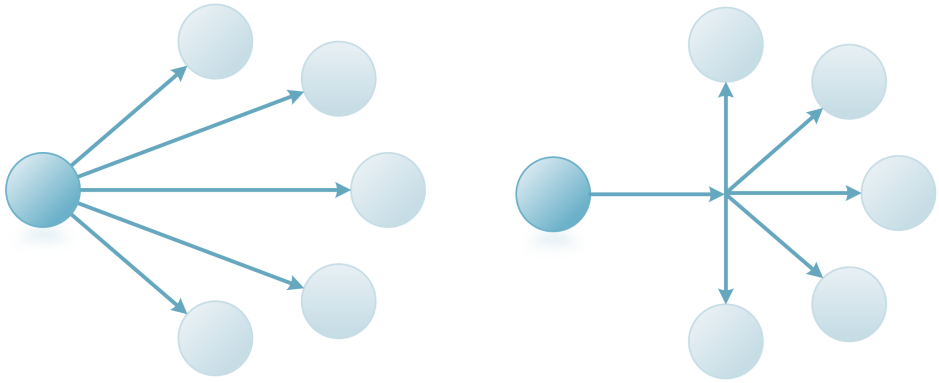


Figure 3.5: Difference beatwean unicast and mutlicast communcation

To properly show needed requirements and assumptions behind the design of network communication we will demonstrate it on our cloud demo example in figure 3.6.

From the above analysis we get these premises for communication protocol. List of what it needs to provide to the framework:

- Regularly show our presence on the network
- Detect freshly joined nodes
- Detect when node rejoin the network
- Send all updates done on top of knowledge to all cluster members
- Maintain order of messages send by individual nodes
- Synchronize whole knowledge once new node is detected

In order to support knowledge repositories basic functions like get,put,take in multi-node environment abstraction of single map data structure over whole cluster is used. In case of jDEECo this is done by using Apache River software in our case this can't be used. As Apache River is not supported on Android platform and as it creates single point of failure.

Each node needs to create its own representation of common knowledge and this knowledge needs to be synchronized to all other nodes in the cluster. As

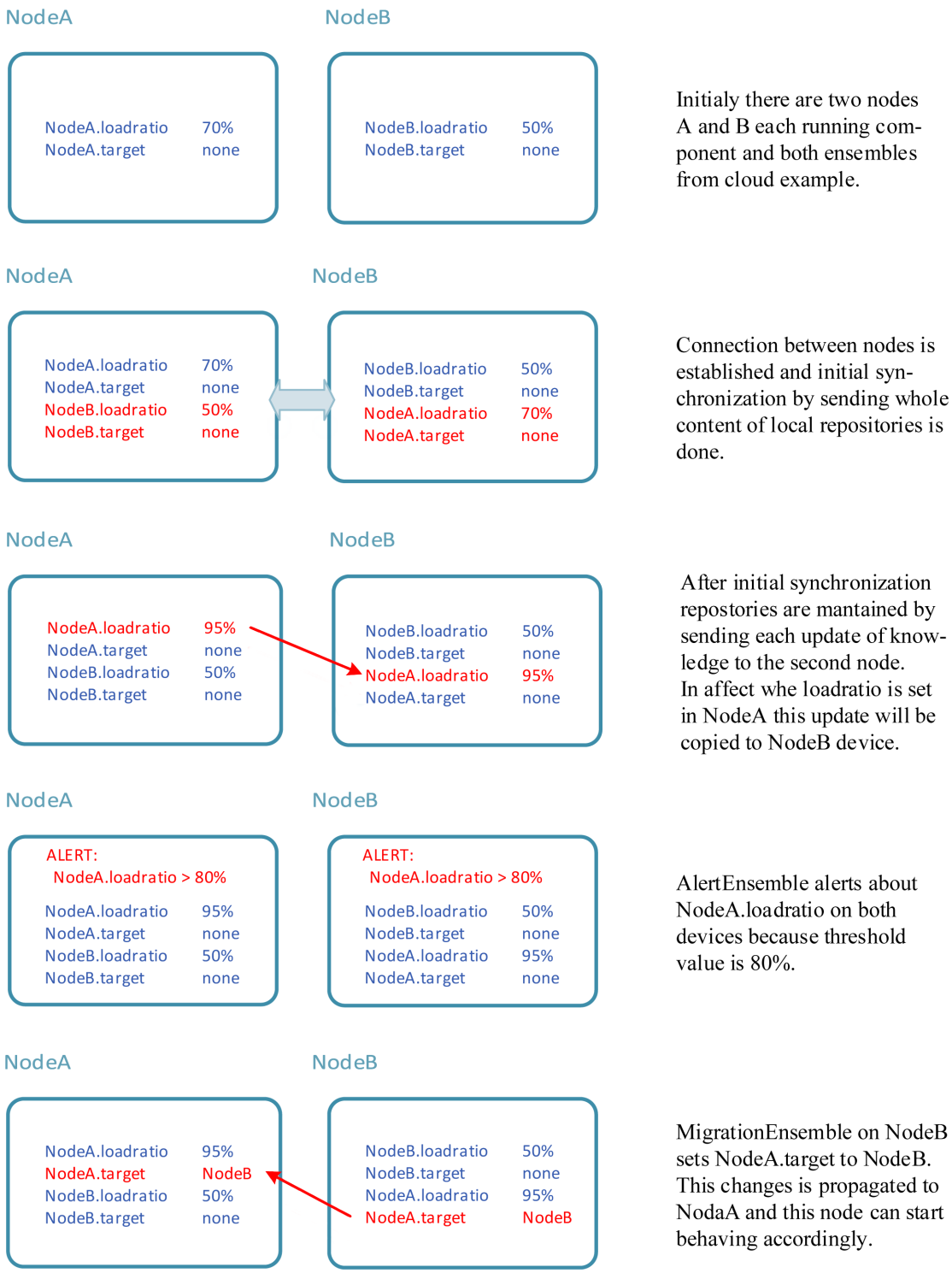


Figure 3.6: Knowledge exchange example

long as cluster is stable (no join or leaving of nodes) synchronization is done by sending multicast messages to all members. For handling changes in knowledge repository transaction like method is employed. Processes in components and ensembles are carried in sessions where firstly needed knowledge is retrieved.

After process ends framework stores changed knowledge back to the repository. Storing of knowledge need to be atomic and propagated to all other nodes in cluster. Atomicity of knowledge update can be achieved by usage of locking on top of all cluster nodes. As each knowledge entity has only one writer process only lock during storing phase is needed, as no other entity, should have changed updated knowledge during the time, when process was running. More complex scenarios of locking are handled by DEECo component model itself.

The problem with re-joining arises when two nodes with the same knowledge connects. It has to be determine which knowledge is valid. Validity of knowledge is based on how recent it is, as it better represents status of the system. For this versioning of knowledge can be employed. Where each update of the knowledge item increases version of the item. The version of the knowledge is than used during merging of two or more clusters and the knowledge item with highest version is kept in local maps of DEECo runtimes. Merging of too clusters is done when two separate groups of nodes connect and each group has it's own map abstraction of the knowledge repository.

As creation of whole multicast messaging cluster solution supporting ordering, versioning and locking isn't main goal of this thesis, we employ multicast toolkit JGroups.

Jgroups

JGroups [5] is a toolkit for reliable messaging, used to create cluster of nodes. It is fulfilling all the needed requirements of DEECo component model in multi-node environment.

Fulfilled requirements:

Node discovery Use multiple ways of discovery on Android platform broadcast and multicast messaging is supported.

Cluster handling It detects and handle joining,re-joining and leaving of nodes. Supports merging of two separated clusters into one.

Message delivery Supports reliable multicast messaging with various level of ordering.

Locking Supports multiple locking schemes on top of an cluster

As in our case JGroups is mostly used to create consistent state across multiple nodes in one cluster. It already has non-production building block *ReplicatedHashMap* which we used as template for *ReplicatedKnowledgeRepository*. More about how it is used can be found in chapter on page 33.

4. Android part

This chapter is devoted to explaining implementation of android part of the framework. It is mostly related to Support background processing on Android and lifetime of DEECo runtime framework on Android platform.

There were three separated projects created for this thesis and can found in attachment 53.

ADEECo This project handled Android part of the framework. It implements all necessary parts for fully functional Android app supporting DEECo component model. This should be taken as development project and not as example how the application should look like.

JjDEECo This project stands for jDEECo on JGroups. It is a modified jDEECo implementation with added support of replicating knowledge using JGroups. Implementation of this project are discussed in next chapter 33.

ADEECo Cloud This app is a demo showcasing usage of DEECo component model on Android platform. It has fully implemented GUI and show jDEECo's demo application Cloud. This application is discussed in chapter Testing on page 41

4.1 ADEECo

ADEECo is an template application for developing new application using JjDEECo implementation of DEECo component model supported by JGroups. It is split into main two parts. Activity handling which is handling user interface and service part handling long running jDEECo framework. In-process architecture as analyzed on page 13 is used.

One of the requirements of Android which affected design is that GUI changes needs to be done only on *MainThread* of the process. And as a service and specifically DEECo framework is running on background threads, application needs to employ internal communication between background threads and the

MainThread. For this purpose EventBus software is used. It is small-weight messaging system created specifically for Android. It is supporting sending events across parts of the application with small overhead.

4.2 EventBus

Using EventBus takes five simple API calls:

```
/** Implement any number of event handling methods in the subscriber.
    Type of the event is determining which posted events will be
    delivered to this function. */
public void onEvent(AnyEventType event) {}

/** For delivering events on MainThread use */
public void onEventMainThread(AnyEventType event) {}

/** Register subscribers */
eventBus.register(this);

/* Post events to the bus */
eventBus.post(event);
/* Unregister subscriber: */
eventBus.unregister(this);
```

Usage of this library simplifies design of application and removes the need for complex communication between service, activity and DEECo runtime. As EventBus is globally accessible it can be used for delivering status updates from component and ensemble processes.

```
    @DEECoProcess
    @DEECoPeriodicScheduling(3000)
    public static void process(@DEECoIn("id") String
        id,@DEECoInOut("loadRatio") OutWrapper<Float> loadRatio) {
        loadRatio.item = new Random().nextFloat();
```

```
5         String text = id+" load "+Math.round(loadRatio.item * 100)+"%";
6         /** this call informs all necessary subscribers about change in
              load of the node. This is not used in DEECo component model
              but in Android application for user interface updates. */
7         EventFactory.getEventBus().post(new MessageEvent(id, text)); }
```

This ensures that only small change in components allows delivering information from component processing to other parts of the application. It can be used for logging, showing status and triggering of actions. If used properly it doesn't conflict with DEECo component model confinement of the processes. There is no direct influence on the knowledge itself but it creates a link how to observe changes in knowledge and react upon them.

4.3 Background processing

To ensure long running of service embedding DEECo runtime framework foreground mode of service is used. Android requires long running services to inform users about its existence by creating notification to notification bar. According to system specification[4] foreground service will be stopped only after all unused activities are destroyed and system still exhibits shortage of memory. In case of service is stop all parts of the framework are shutdown and deallocated. This forces JGroups toolkit to leave current cluster and forget current knowledge. It is possible to store current knowledge and restore it once service is started. But as DEECo component model uses only the most current version of data there is no big benefit for implementing this storing solution. If needed it can still be done by usage of persistency support in JGroups toolkit.

5. Inter-node communication

It this chapter we introduce more technical parts of implementation of inter-node communication in our project. Main part of inter-device communication is done in JGroups toolkit. Project uses jgroups-2.12.0.Alpha3 version as this is the latest version ported to Android platform [10].
Most of this chapter is devoted to JjDEECo project.

5.1 JjDEECo

This project is using standard Java 6 and it is a fork of original jDEECo project. This enables that *KnowledgeRepository* created for Android and based on JGroups is usable on many other platforms. Other projects ADEECo and ADEECo Cloud are including sources from JjDEECo project. Project is based on older version of jDEECo which is using Java 6 because newer versions use Java 7 features incompatible with Android platform. Implementation of *KnowledgeRepository* backed by JGroups should be portable to newer jDEECo versions. JGroups is integrated into the projects as one jar file.

Parts of functionality is removed from original jDEECo project. Mainly interpretation of classes files as sources of components. As on Android there is no such possibility. OSGi support that is build in jDEECo had to be removed as used implementation of OSGi is not supported on Android. There are possibilities how to introduce OSGi to Android[11, 12] but as they are in development stages and OSGi support is not necessary for DEECo it wasn't investigated more closely.

5.2 JGroups

JGroups toolkit is used as core implementation of inter-device communication. Our usage is revolving around using non-production building block[5] named *ReplicationHashMap*. This building block is a part of JGroups toolkit and it shows a possibility how to create abstraction of one Map over multiple cluster

nodes. It is not production ready because the logic behind merging of clusters needs to be implemented in application specific way. In our framework JGroups with improved version of *ReplicatedHashMap* creates *ReplicateKnowledgeRepository* which is used as abstraction level of common knowledge for all nodes in a cluster.

Standard UDP configuration of JGroups is used with only minor changes. This implies that multicast is used as discovery method and also as main channel for communication between nodes. JGroups employ an abstraction of channel as most important part of outside interface. Channel creates interface for reliable sending of messages to all nodes in a cluster. It is composed of multiple layers of protocols which handles different parts of reliable message delivery.

Example code of creating JGroups channel in JjDEECo project which is part of *ReplicatedKnowledge* initialization:

```
1 System.setProperty("java.net.preferIPv4Stack" , "true");
2 channel = new JChannel("assets/udp.xml");
3 channel.connect("Adeeco");
4 replMap = new ReplicatedHashMap<String,
    ReplicatedList<Object>>(channel);
5 replMap.start(10000);
```

As currently Android supports multicast only on IPv4 networks we set preferred stack accordingly. Creation of new channel requires xml configuration used to setup protocol stack[5]. It is possible to create multiple JGroups channels in one application and in one network. For channel identification JGroups uses single string parameter in our case 'Adeeco'. This means that all reachable JGroups instances with channel 'Adeeco' will create single cluster.

5.3 Knowledge replication

All functions used by *ReplicatedKnowledgeRepository* are session oriented. This transaction like interface enables possibility for locking and rollbacks of done changes.

jDEECo uses slightly modified behavior of common map functions. Difference is that instead of storing one value for each key it is storing list of values. FIFO is used for ordering values in the list. This implies that function put doesn't replace currently stored value but it adds new value at the begging of the list. In case of function take it will return and remove all values in the list. And in get function array of all stored objects is returned.

```
1   /** pseudo code showing main ideas behind changed bahaviour of map
        like interaface. Session behavior is not considered.*/
2   public Object[] get(String entryKey, ISession session){
3       return map.get(entryKey).toArray();
4   }
5   public void put(String entryKey, Object value, ISession session){
6       vals = map.get(entryKey);
7       vals.add(value);
8   }
9   public Object[] take(String entryKey, ISession session){
10      vals = map.get(entryKey);
11      copy = Copy(vals);
12      vals.clear();
13      return copy;
14  }
```

5.4 Cluster merging

JGroups handles joining of nodes into the channel but from application side there is a need for state transfer. In our case current state of knowledge repository. There are two distinct ways how the node will be included in existing cluster.

Initial search

Once the JGroups channel is started it waits for determined period of time before releasing channel for usage. This time is used for discovering any node already existing in the network.

If another node is found in the network JGroups initiates state transfer from that node to the newly joining one. In case whole cluster is found JGroups choses one cluster node as the node which transfers the state. This initial transfer of states is done by *getState()* and *setState()* functions. In case of knowledge repository all knowledge of transfer node is serialized and send to joining node. This node fills its knowledge repository and starts receiving updates. In case no other node is found in the network node assumes it is the first node and carry on without any state transfer.

Merging

Another case is when two already initialized nodes found each other. In this case JGroups toolkit will starting Cluster merging. In case of two nodes it is considered as two clusters with node count of one. Merging of the state is application dependent and in our case preservation of knowledge from both clusters is required. JGroups will call *viewAccepted()* in all nodes. *View* represents connected nodes to the cluster and this function is dealing with changes in cluster composition. If the *MergeView* is send to this function we initiate merging of knowledge for the clusters.

As mentioned in the analysis on page 28 each knowledge has a version number associated with it. These version numbers ensures that newer knowledge will not

be rewritten by older knowledge. Using this feature suffice to resend all knowledge stored in the knowledge repository to synchronize knowledge repositories across all nodes.

In fact it is enough that only one node from each cluster resends all its knowledge. As knowledge repositories are synchronized over the cluster all nodes contains the same knowledge. This resending is done on the background and doesn't stop standard processing. Once all resending is done clusters have merged in to one cluster and all nodes shares the same state.

5.5 Session support

Transaction like behavior created by sessions is implemented in all three main functions:

- get(String key,ISession session)
  Retrieval of stored knowledge is alway done from locally stored copy of replicated map. Session is storing all previously given values and if same item is asked again it will used stored value and not actual value in the knowledge repository. This prevents changing of knowledge value during session.

- put(String key, Object value,ISession session)
  Storing of new values to local copy of replicated map is postponed and stored in session. If the changed knowledge item is retrieved again session will return previously updated value.

- take(String key,ISession session)
  Retrieval and removal of all values for one knowledge item is stored in session. When repeated retrieval is called empty array is returned as take had cleared the value.

Implementation of session storing requires to store only the last performed function. Storing type of last function (GET,PUT,TAKE) and the value that should be stored in knowledge repository is sufficient. Closing of session is done

by cancel() all end() functions. In case of cancel() rollback operation is performed which means that no change is done on top of replicated knowledge repository. In case of end() function commit operation is stared. Knowledge repository local lock is acquired and all operations stored in session are applied to the knowledge repository.

This schema supports only weak locking as multiple pathological cases can be found. Locking is done only on local copy of the knowledge repository which is acceptable only as long as one writer per knowledge item rule is unbroken. JGroups supports locking schemes on top of whole cluster but in our case it will introduce too big performance hit for little gain.

5.6 Knowledge aging

One aspect of multi-node environment like proposed by our work is that nodes can leave the cluster for long periods of time or permanently. Problem is that their knowledge stored on another nodes continuous to be used and even replicated to freshly joined nodes. This implies that out of date knowledge would be used in ensemble functions.

There are at least two ways of solving the problem.

remove all knowledge belonging to disconnected node Currently there is no linkage between the node and knowledge stored in the repository. Creation of this linkage would require changes in jDEECo usage of *KnowledgeRepository*.

time-stamp the knowledge and remove when too old Producer and updater of knowledge would time-stamp the data. Data would be removed after some arbitrary time. This is adding requirement of well enough synchronized clocks across the cluster nodes. Another created requirement is detection of all knowledge for one component on the level of *KnowledgeRepository* as it needs to be removed at the same time.

Both approaches requires substantial changes in jDEECo usage of *KnowledgeRepository* interface. Currently no solution for knowledge aging is carried out in JjDEECo or in jDEECo implementations of DEECo component model. As not even DEECo component model is considering this problem, we leave this issue open for future improvements.

5.7 JjDEECo usage

Implementation of JjDEECo has completely compatible interface regarding definition of components and ensembles. Therefore all existing components and ensembles for jDEECo should be function under JjDEECo. Directly in project exists multiple examples used for testing of required features.

Only real difference from jDEECo implementation in how to use the framwork is replacement of *LocalKnowledgeRepository* or *TuplespaceKnowledgeRepository* for *ReplicatedKnowledgeRepository* class.

JGroups and possibility to use mutli-cast or unicast communication enables multiple scenarios of usage.

P2P Cluster Cluster created from equivalent nodes where all nodes reside in one multicast domain. This is the usage targeted by this thesis. To support binding multiple locations REPLAY[] protocol implemented by JGroups can be used.

Client / Server JGroups allows to set distant nodes as target of communication. If particular addresses are set they don't have to be in mutli-cast domain and still can be part of the cluster. For example all nodes will connect to one main node which is openly available on the Internet. Creating cluster of nodes in distant networks. Main node will impair distributed character of the cluster but can server as server for other nodes.

Cloud This scenario is improvement over Client / Sever in a way that server is distributed over many nodes. As server side would consist from high number of servers in different locations it would create network with cloud like computing.

All of the above use-cases can be implemented rather easily thankfully to JGroups. Already exists similar solutions build on top of JGroups toolkit. Deployments where hundreds of nodes cooperate with JGroups exists.

6. Example and testing

Main goal of this chapter is to introduce created applications and show test-cases. Testing is separated to two distinct parts. First part is testing and evaluating of ADEECo Cloud on page 41 application. Mainly they are evaluating performance of our solution on Android Platform. Time of inclusion to the cluster, battery usage and memory consumption.

ADEECo Cloud

ADEECo Cloud application is an example of usable Android application on top of JjDEECo implementation of DEECo component model. As it is using cloud demo components it doesn't demonstrate any particular purpose. But its graphical interface enables observation on how the implementation behaves in dynamic environment. Particularly how new joining members of the cluster are handled.

There two main activities in this application first one 6.1 is the lists all components which have their knowledge stored on the device. It can be either locally run component or component running on different node which knowledge was replicated to the given device. Additional information about component is shown after touching row with the component 6.2. For each value in knowledge repository belonging to the selected component interface shows its name, value, time of last update and version. Another main activity 6.4 is available by button in left upper corner. This activity shows all messages generated by components and ensembles about important events. From this information we can infer status of the DEECo application underneath. This application also supports tablet 6.3 size screen when selection of components and showing of additional values are done in one screen.

For testing and development of ADEECo Cloud demo application several devices were used:

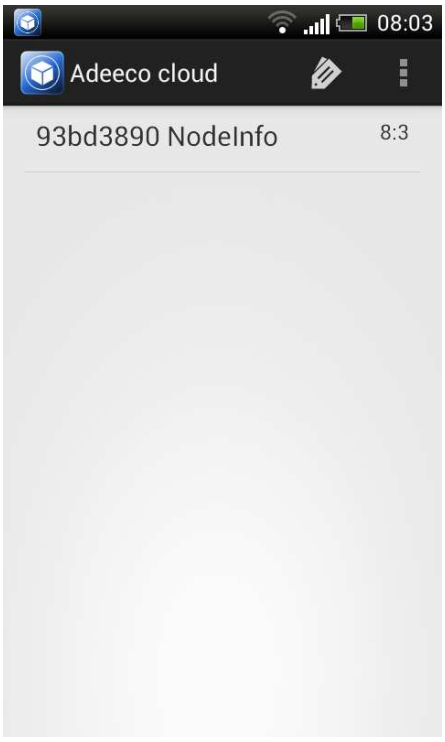


Figure 6.1: List of components

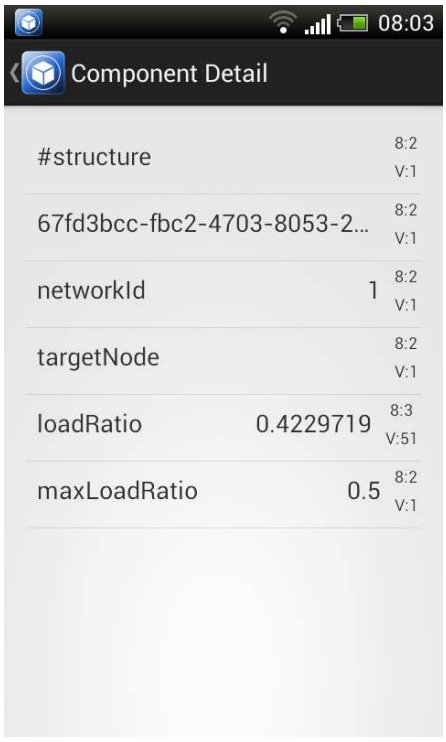


Figure 6.2: Components detail

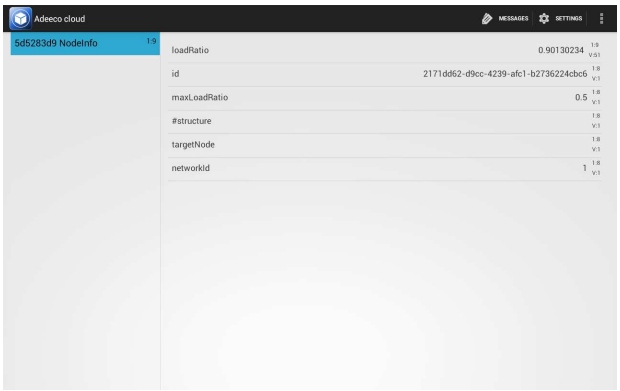


Figure 6.3: Tablet interface

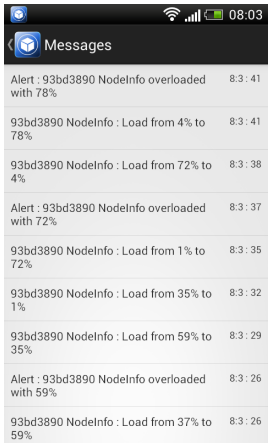


Figure 6.4: Meessage screen

- 2x HTC DESIRE X Android Mobile Phone
  OS version Android 4.1.1 with API level 16 and 4” screen
- HTC HERO Android Mobile Phone
  OS version Android 2.1 with API level 7 and 3.2” screen
- 2x SDK Device Emulator
  OS version Android 4.03 with API level 15 and 3.7” screen
  OS version Android 4.03 with API level 15 and 10.1” screen

On all devices ADEECo Cloud was fully functional and graphical interface behaved correctly. SDK Emulator doesn't support full emulation of network and there is no support for multicast messaging. This determined that multi-node operation couldn't be tested on SDK environment and also development of multi-device part of the projects could be done only on real devices.

Battery test

Battery consumption test was conducted on HTC DESIRE X device. The initial conditions were alway fully charged battery no other background process application running. Application was started in required form and left for four hours to show battery decline. This test show how different modes of operations changes battery usage. As on mobile platform battery is one of the main resources we tried to measure impact of running our solution.

	WIFI	NO WIFI
Clean device	94 %	98 %
No components	92 %	96 %
One node	91 %	95 %
Two nodes without components	93 %	-
Two nodes	86 %	-
Ten nodes	87 %	-

Table 6.1: Results of battery test in seconds

There were six kinds of test performed on the devices (**??**). First three test were performed with or without network connection. Each test was carried out only once as each testes taken more that four hours. Together more than forty hours of testing was done. Regardless this doesn't allow for statistical analysis of the results as we have only on measurement for each kind of test. Also resolution of the discharge is only in percentage which creates big measurement error.

Clean device test is a baseline consumption when no ADEECo cloud application was running on the device.

No components case ADEECo Cloud application was running but no component or ensemble was loaded into the DEECo runtime framework. This test should show consumption of JGroups discovery protocol.

One node test show running standard ADEECo Cloud application when no other node was reachable.

Two nodes without components case have one reachable node and both without any component or ensemble.

Two nodes case is same as previous test but components and ensembles were running on the nodes.

Ten nodes case show how number of nodes is impacting battery consumption.

Altogether results are with agreement with expectation. Unfortunately small result set doesn't allow to check measurement error. In case of turned off Wi-Fi higher consumption should be due to CPU time spend on JGroups and DEECo framework scheduling. As even when no network is connected JGroups regularly ties and fails to send discovery packet. Almost no difference between two and ten nodes test is probably caused by operation Wi-Fi on full strength in both cases. Standard time for Wi-Fi to go to sleep mode is more than five seconds on Android [4]. And as discovery is done every three seconds and components updates their knowledge each second Wi-Fi willnever enter sleep mode.

From above test we can deduce that significant savings can be done by prolonging discovery time and by updating knowledge in longer interval. More test and investigation would be needed to propose any real values for savings or possible settings of the parameters.

At the end of each of these tests memory consumption of the application was checked. Resolution of the captured values is only in MB. Developer tools allow for more precise measurement but this would interfere with battery testing. Also memory consumption was always at 14 MB. Therefore we concluded that no more test are required.

Connection test

Purpose of this test was to measure mean, max and min time for an already runnig node to be added to the cluster. In this test case one JjGroups instace was running one notebook computer and HTC DESIRE X running ADEECo cloud connected to the same network as notebook used. Twenty tests were performed.

9	15	6	20	12	12	23	18	6	19
8	10	22	12	27	18	7	14	8	12

Table 6.2: Measured connection times in seconds

Maximal time of inclusion 27 seconds
Minimal time of inclusion 6 seconds
Average time of inclusion 13.9 seconds
Stadart deviaton of result set 6.13 seconds

Mean time for inclusion is 13.9 seconds which is quite high in comparison that discovery is running every 3 seconds. This result show us that JGroups toolkit needs time to start *mergingView* to join two clusters. This behavior might be improved by changes in configuration or possibly new discovery protocol can be implemented with better results. But for demonstrating mobility of ECBS system the acquired value should suffice.

JjDEECo

JjDEECo implementation is evaluated separately. Because JjDEECo has possibility to run on much greater range of environments. In case of JjDEECo mainly testing compatibility on platforms was tested.

Platforms on which JjDEECo successfully worked and communicated with other nodes in the network.

- Linux Mint 16 Petra using Linux kernel 3.11.012 on x86_64 architecture
- Windows XP Service Pack 3 with x86 architecture
- Raspbian Operation system on Raspberry Pi with ARM architecture
- HP-UX 11i v3 with Itanium architecture

This shows that JjDEECo and JGroups used by the project runs on many platforms which are supporting Java language. This enables variety of testing and deploying scenarios. Another benefit is than development of JjDEECo and mainly *ReplicatedKnowledgeRepository* is not thigh to Android platform.

During development phase standard Java work-flow was used and this saved considerable time as building, deploying and testing of apps on Android takes more time.

7. Conclusion

In this master thesis we have analyzed the problem of bringing DEECo component model [1] on to the Android Mobile Platform [4]. After careful analysis and identifying possible solutions, we have used current pilot implementation of DEECo component model called jDEECo [2, 3] as a basis for our implementation on Android. Finally with only slight modifications to the jDEECo we where able to run locally DEECo framework (33). Next step of bringing the solution on to the Android was to create synchronized shared state abstraction of knowledge repository on multiple devices (33). For this purpose we have used JGroups toolkit together with our own implementation of *ReplicatedKnowledgeRepository*. Bringing these two parts together in one coherent unit was finally done in our ADEECo Cloud example (41).

As for the goals of this master thesis we have fulfilled each one of them.

Design and implement DEECo component model on Android Portation of jDEECo enabled to have most of the current functionality available on Android Platform

Support background processing on Android By utilizing Androids components namely a Service we were able to create solution for almost permanently running DEECo framework.

Support inter device communication and synchronization Targeting Wi-Fi technology as main communication pathway and employing JGroups toolkit helped us to create robust synchronization framework. Which is permitting cooperation of components and ensembles in the multi device environment.

Create demo application By creations ADEECo Cloud application we have shown how the DEECo component model can be used on Android platform.

There is one hidden benefit of chosen usage of jDEECo and JGroups which wasn't required from this thesis. Not only Android devices can collaborate together on DEECo platform but they can do this with virtually any device supporting

standard Java and Multicast communication (3346). All this is possible by using our fork of jDEECo called JjDEECo which adds JGroups support and doesn't require any Android specific interface.

As for now there are still some pitfalls in the solution most notably knowledge aging (38) and battery consumption (43). But both of these problems are out of scope of this master thesis as they are not in our goals. These are possible places where future improvement can be done.

During writing of this master thesis we have encountered many problems. But by systematic analysis and small step improvements we were able to resolve them and create presented solution. We have learned a lot about DEECo component model and its pilot implementation jDEECo, also get familiar with JGroups toolkit and Android Platform. This work has given us experience and knowledge that with dedication and time these types of projects can lead to fruitful ends.

Bibliography

[1] Bureš T., Gerostathopoulos I., Hnětynka P., Keznikl J., Kit M., Plášil F.: DEECo - an Ensemble-Based Component System, Tech. Report No. D3S-TR-2013-02, Dep. of Distributed and Dependable Systems, Charles University in Prague, February 2013

[2] Bureš T., Gerostathopoulos I., Hnětynka P., Keznikl J., Kit M., Plášil F.: DEECo - an Ensemble-Based Component System, In Proceedings of CBSE 2013, Vancouver, Canada, ACM, June 2013

[3] Al Ali R., Bureš T., Gerostathopoulos I., Hnětynka P., Keznikl J., Kit M., Plášil F.: DEECo: an Ecosystem for Cyber-Physical Systems, In Companion proceedings of the 36th International Conference on Software Engineering (ICSE 2014), Hyderabad, India, ACM, poster and extended abstract, June 2014

[4] Android documentation project, developer.android.com

[5] Bela Ban,JavaGroups - Group Communication Patterns in Java, Dept. of Computer Science Cornell University, July 1998

[6] Bela Ban, JGroups documentation, jgroups.com

[7] Eric Freeman, Susanne Hupfer, Ken Arnold: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley Professional, 1. June 1999, ISBN 0-201-30955-6

[8] Camps-Mur D., Garcia-Saavedra A., Serrano p.:Device to device communications with WiFi Direct: overview and experimentation,NEC Network Laboratories in Heidelberg, Germany,2012

[9] N. Serbedzija, S. Reiter, M. Ahrens, J. Velasco, C. Pinciroli, N. Hoch, and B. Werther, Requirement specification and scenario description, ASCENS, 2011. http://www.ascens-ist.eu/deliverables/

[10] Yann Sionneau, Portation of JGroups to Android, github.com/fallen/touchsurface-android-jgroups, 2012

[11] Bouzefrane S., Huang D., Paradinas P. : An OSGi-based Service Oriented Architecture for Android Software Development Platforms, Arizona State University,Conservatoire National des Arts et Métiers, 2011

[12] Kuna M., Kolaric H., Bojic I., Kusek M., Jezic G. :Android/OSGi-based Machine-to-Machine Context-Aware System, Faculty of Electrical Engineering and Computing, University of Zagreb,2011

List of Figures

List of Tables

Attachments

1. Source codes of projects ADEECo, JjDEECo and ADDECo Cloud