BSc Thesis Report

KUIServiceManager: A service management system for KUI-based Android mobile applications

Samuel Vonlanthen
samuel.vonlanthen@unifr.ch

Supervised by Prof. Béat Hirsbrunner and Pascal Brügger

Pervasive and Artificial Intelligence Group
Department of Informatics
University of Fribourg, Switzerland

February 2, 2011
Abstract

KUIServices are context-aware applications that run on a server (SmartEnvironment) and client (Android mobile) base. The context-awareness is supplied by the underlying uMove framework. The uMove framework together with the coordination manager defined in [Hadorn, 2010] offer all tools to create KUIServices, connect the Android mobile clients to SmartEnvironments and to enable communication between server and clients. However, the KUIServices have to be set up at system start. In this article we present a KUIService management system that allows dynamic handling and accessibility of KUIServices.

We introduce KUIServiceManager, a KUIService that extends the underlying architecture by the possibility of ad-hoc loading of KUIServices. And we present KUIServiceMarketForAndroid, an “Android Market”-like application that dynamically provides the user with the available KUIServices. The KUIServiceMarketForAndroid will be referenced to as KUIServiceMarket within this report.

The implementations we present in this report are transparent to the existing uMove and Coordination project. This means that no major changes have been done in the existing architecture.

Keywords: KUIService, KUIServiceManager, KUIServiceMarket, uMove, Coordination, Android.

---

1 KUI stands for Kinetic User Interface.

2 http://diuf.unifr.ch/pai/umove
Contents

1 Introduction 3
  1.1 From a globalized market to a local market 3
  1.2 Contribution 4
  1.3 Outline 4

2 Android platform 5
  2.1 The Android components 5
  2.2 The manifest file 6
  2.3 Intents 6
    2.3.1 Information contained in Intents 7
    2.3.2 Resolving Intents 7

3 uMove 8
  3.1 The uMove framework 8
  3.2 uMove: the development framework 8
    3.2.1 Conceptual framework 8
  3.3 Coordination and communication in uMove 9
    3.3.1 Inter-object communication 9
  3.4 Monitoring 10
    3.4.1 Monitoring mobile devices 10
    3.4.2 Services list update 10
  3.5 KUIEditor 11

4 KUIServices 12
  4.1 System- and public KUIServices 12
  4.2 KUIServices components 13
    4.2.1 Client part 13
    4.2.2 Server part 13
    4.2.3 Rule 14
    4.2.4 Meta data 14
  4.3 Preparing KUIServices to be loaded 15

5 KUIServiceManager (Server-Side) 16
  5.1 Parsing the Metadata 16
  5.2 Loading and starting KUIServices 17
    5.2.1 Mapping the ports 19
    5.3 Applying the rules 19
Chapter 1

Introduction

The boom of smart phones started a new age in computer science. It is predictable that in a rather short future, having a smart phone will be standard. This also means that everyone will be accompanied by a whole bunch of sensors such as accelerometer, GPS, etc. The values of this sensors can be accessed from software running on those devices, what brings a new motivation for context-aware applications. More precisely, it offers the fundamentals for localized software. With this project we want to supply smart phone users with useful applications, when they need it and where they need it.

1.1 From a globalized market to a local market

Smart phones lead to the situation that information is available nearly everywhere. But this also raises the need of having access to useful tools, i.e. programs, to deal with the data. Today’s operating systems on smart phones deal with this need by having a central, globalized market. Such a market is an application that allows the user to download or buy new applications, like the AppStore on iPhones or the AndroidMarket on Android systems.

Having a global market clearly makes sense for a wide range of generic and/or basic applications. These are applications the user will use from time to time depending on the current need. For example, a document viewer is typically used when the user wants to display a PDF file, a game may be played when the user is bored, etc. It makes sense that such applications are always available, since they come into play whenever a certain task has to be accomplished.

But when we think about customized and location-tailored applications, a globalized market may not be the best choice. Imagine an application that locally shows you the way to points of interest, e.g. a certain product in the supermarket or the nearest emergency exit in case of fire. Or think about a small application that guides you through the museum you are currently visiting. Such software is different from the ones described above, since its use depends on a context, i.e. on a certain situation. Such applications support the user only if the situation matches, e.g. if the user really is in the museum at the moment, or if the building really is on fire.

Having a market that offers appropriate applications implies that the market application itself must be capable of analyzing and evaluating the current situation of the user. Therefore we have chosen to build our KUIService management system over the very same framework the localized applications themselves will use.
1.2 Contribution

We propose the use of an additional, local market for localized applications. The global market will still be in place and will remain the users default choice to get applications. But additionally, the user may also check for services in the local market. Since the local market will be capable to recognize situations as described above, it may suit way better to the users needs.

Primary benefits from this approach:

- **Customized selection** - A local market would display only useful apps that correspond to the users current environment.

- **Remove footprints** - When the environment changes (e.g. leaving the museum), a local market could react by automatically uninstalling applications that are not useful anymore. This prevents the user from having installed numerous applications that only support the user in a certain case.

- **Separated worlds** - Since the local market is independent from the global market, they don’t interfere with each other. Additionally, it also clearly separates the two “worlds” of general and localized applications from the users point of view.

Our idea of this local market follows the server-client principle. On adequate places (e.g. a train station, a university, a museum, on public places, etc.), a SmartEnvironment offers the customized applications. Clients that enter the range of the server can connect to it and may get the services offered by this SmartEnvironment. In the area of the server, the contextualized services may be used and as soon as the user leaves the environment and the connection is terminated, the service is automatically uninstalled leaving no footprints.

1.3 Outline

In chapter 2 we give a very short introduction to the Android operating system, since the implementation we present here is designed for Android 2.0. In chapter 3 we present uMove, the framework for pervasive systems with which we have developed the KUIService management system.

In chapter 4 we specify what KUIServices (i.e. the KUI-enabled applications to be used with the local market) exactly are.

In chapters 5 and 6 we introduce the server and client part of the KUIServiceManager. In chapter 8 we present the KUIServiceMarket application.

The appendices A and B contain a step-by-step guide about how to install and use our management system and about how to develop KUIServices.

In this report we focus mainly on the choices and the design resulting from this project. Detailed information about the implementation is provided by the extensive Javadoc and the comments that come with the source code available from the CD-ROM attached to this report.

---

1 A SmartEnvironment is a server running a KUISystem. We generally assume that a SmartEnvironment is accessible within a certain area, e.g. via WIFI.
Chapter 2

Android platform

Android is an operating system and software platform for mobile devices, developed by the OHA. It is widely used among the current smart phones, reaching more than 25% market share in the third quartal in 2010 [Tudor and Pettey 2010]. The fact that wide parts of the Android world is open-source and the useful development- and debugging tools make it a perfect choice for research projects, although it has its limits as we found out during the work on this project.

In this chapter, we give a very short introduction to Android development. We will focus on the main topics only. A very readable and complete documentation including a good amount of examples can be found on the developers homepage of Android, http://developer.android.com.

2.1 The Android components

The Android framework is organized on a service-oriented way. Each application normally runs in its own Linux process and is therefor encapsulated from the other software. An Android app consists of different components that are built to accomplish a certain task (i.e. to offer a service). All the different components that exist on a device can be used within the same or other applications (e.g. if you need your software to display a web page, you will call an existing “web browser component” to display it for you instead of directly coding resp. including the necessary code into your app).

The components types are the following:

- **Activities** - Activities are components that display a visual user interface for a certain task. Normally, a user interacting with an Android application swaps from Activity to Activity. For example a music player could have an Activity to control the playback of the music and an Activity to compose playlists.

- **Services** - Services are components that run in the background for an indefinite period of time. Running in the background means that they are not visible to the user. Services typically keep running even if the user starts a different task on the phone. In the example of the music player, a Service could be used to effectively play the music since no direct user interaction is needed here and the music is suggested to keep playing if the user e.g. wants to compose an email meanwhile.

[http://developer.android.com](http://developer.android.com)

[1] Open Handset Alliance

[2] Open Handset Alliance
2.2: The manifest file

- **Broadcast Receivers** - Broadcasts can be compared to messages that announce events. Most of them originate in basic happenings in the operating system itself. Broadcast Receivers are components that run in the background and are activated as soon as the broadcast (they are listening to) occurs. In the music player example, we could use a Broadcast Receiver that is listening to incoming voice calls and automatically pauses the playback of the music when it is triggered.

- **Content Providers** - Content Providers are components that make data available to other components. The data itself is stored in the file system, in a SQLite database or similar. The Content Provider then offers handlers and other tools for accessing this data. The music player could offer the information about the currently played song and the playlist in use. Other applications might then use this information to offer a small display to always show the user what music he is currently listening to.

### 2.2 The manifest file

The manifest file is a XML file that specifies different information about an Android application and is present for each app, similar to the meta data we use for our KUIServices (Sec. 4.2.4). As clearly stated in [Developers, 2010a], all components (except Broadcast Receivers) must be known to the Android system before they are used. That’s the reason why they also have to be declared in the manifest file. Whenever a new APK is installed, Android will scan the manifest and register the defined components. This is necessary as Android needs to know what components are available and what tasks they can be used for (Sec. 2.3).

Using such XML meta data is a good approach to combine the needs of the Android architecture and user friendliness. However, it limits the flexibility of Android applications. E.g. it implies that plugin-based applications would need to be reinstalled completely each time a plugin is applied. This behavior also influenced our project as mentioned in Ch. 6.

```xml
<manifest ...
  <application ...
    <activity ...> ... </activity>
    <activity ...> ... </activity>
    <service ...> ... </service>
  </application>
</manifest>
```

Listing 2.1: Default structure of a manifest file

### 2.3 Intents

Intent objects are used to activate all types of components. The music player example we used in the previous section would use intents in different situations. For example, if the user hits the “Stop” button in the Activity for the playback control, an Intent will be fired to the Service that is playing the music and tell him to stop. Or when the user wants to edit the playlist, an Intent will cause the playlist Activity to appear in the foreground.

---

3APK stands for Android Package.
2.3: Intents

2.3.1 Information contained in Intents

Android Intents can carry some additional information describing the task to be performed by the called component. E.g. for the music player, an Intent that tells the background Service to start playing the music should also deliver the information what song to play.

The main fields that can be specified in an Intent are the following:

- **Component name** - This field can target a component explicitly (Sec. 2.3.2).
- **Action** - A string that specifies the type of action to be performed by the component (e.g. “Start” or “Stop” for the music playback Service).
- **Data** - A URI that can be used to specify additional data. This could be the URI to the song that should be played.
- **Category** - This string is used to specify the type of component that should be used to perform the desired task.
- **Extras** - A list of key-value pairs that can be used to place additional information.
- **Flags** - A set of booleans that can be set to influence the way how the component is started or to specify if a result is expected from the component, etc.

2.3.2 Resolving Intents

There exist two types of intents: explicit and implicit Intents.

*Explicit Intents* are used to activate components within a single application. Since all classes are known within one app, the developer can specify directly what component should handle this Intent by passing its class name to the optional Component name field. No resolving takes place in this case and the Intent is always delivered to the specified component, no matter if it is declared to handle such Intents or not.

Intents not having set the Component name field are called *implicit Intents*. They are used to address components that are not part of the same application, i.e. to use functionalities offered by other software. Since no component is specified directly, Android will try to find among all registered components the one that fits the best for the task to be done. This process of finding an appropriate component to a given Intent is called “Intent resolving”.

Resolving an Intent actually consists of matching it against the Intent Filters of all known components. Intent Filters can be included in the tags defining components in the manifest file. Android will then use the information from the Filter to evaluate if the component matches the Intent or not. Intent Filters have similar fields as the Intents themselves, and matching basically means comparing the fields of the Intent with the according fields of the Filter, as declared in [Developers 2010b].
Chapter 3

uMove

The uMove project ([Bruegger et al., 2009], [Bruegger et al., 2010]) focuses on the modeling of motion and activity-based ubicomp systems and offers a generic framework which allows to design and implement ubicomp systems and applications based on a model of the physical environment.

In this chapter, we will give a short introduction to the key ideas of the uMove framework to simplify the understanding of the following chapters in this report. It is obviously impossible to summarize a whole project in a few lines, therefore we highly encourage reading the different papers related to uMove.

3.1 The uMove framework

The uMove framework allows to define and implement a KUISystem on top of which different specific applications (services) will be developed. The framework contains two specific parts: the conceptual framework and the Java API. uMove allows programmers to easily create all entities, the relations between them and the connected sensors, and to load the activity and situation recognition modules (algorithms or classes). Therefore, uMove is a good choice to model physical environments and the entities within them (e.g. students on a university campus).

3.2 uMove: the development framework

3.2.1 Conceptual framework

The conceptual framework is the tool used by developers to theoretically design the system that will be observed. As shown in Fig. 3.1, a KUISystem has three layers responsible for the different objects interacting together. The sensor layer contains the sensors, which are the logical abstractions of the sensors connected to the system. The entity layer contains the logical representation of the physical entities (i.e. users, places, objects) being observed. Each entity (actor, places or zones) is defined by its identity, its role, its location within the system and its current motion and activity. The observation layer analyses the current situation of the actor based on their activity and contexts. They listen for any entity changes and forward them to the situation manager in order to have the new situation analyzed and if needed, inform the application (e.g. a “warning” message or a “critical situation” message).

1 stands for sensor gadget similar to the concepts of widget (Windows gadget) or phidget (physical gadget)
3.3 Coordination and communication in uMove

The communication in uMove is managed by a component called the Coordination manager. This component allows the different entities (actors, sengets, observers, viewers) to transparently communicate with each other and also allows different uMove systems to be connected to one another (Fig. 3.2). Each entity communicates together in the same way regardless of whether they are local (same uMove) or remote (on a mobile uMove).

3.3.1 Inter-object communication

The communication between objects is based on the message listener concept. For instance, an observer listening to an actor automatically receives messages any time changes occur. This type of asynchronous communication allows the system to be dynamic, possibly distributed, and guarantees that all processes run in parallel. Each object (senget, actor, observer) is connected to a port object.
which is dedicated to listening to a channel established and managed by the coordination manager.

3.4 Monitoring

uMove enables applications (so called Services) to run on top of its architecture. Services provide a server part that runs on the SmartEnvironment and a client part that runs on the mobile device. So how can services be discovered? A specific package called Monitoring has been developed in order for a KUISystem (e.g. mobile client) to be detected by an other KUISystem (SmartEnvironment) and to get the list of available services.

3.4.1 Monitoring mobile devices

Each time a mobile device running a KUISystem enters in a zone covered by a SmartEnvironment, it is contacted, identified and integrated into the SmartEnvironment. A mobile KUISystem entering in a WIFI zone broadcasts a ping message containing its open listening port for any echo message. If, in this network, a SmartEnvironment is active, it will answer with an echo message containing the open port address of the main service. This service is used to log the mobile device in the SmartEnvironment and creates a permanent communication channel. This channel will be used to pass all messages about public services available in the SmartEnvironment and all context changes (e.g. location, motion, temperature) from the mobile device to the server. It can happen that multiple SmartEnvironments from different building overlap and provide different services. To benefit from all of these services, our mobile KUISystem allows connection to multiple SmartEnvironments at the same time.

3.4.2 Services list update.

Services are stored in a Service Registry in every SmartEnvironment and can always be enabled or disabled. Each time a service changes, the Coordination manager checks for all matching clients in order to send the update. The Service Registry also keeps track of which service is available for which client (meaning mobile KUISystems). This depends on the context of the user. For instance, a chat service might not be available to the user if he stands in a meeting room and his agenda has an entry “Meeting, priority 1” or the menu service is disabled if clearly the user’s activity shows that he is leaving to catch his train.

![Figure 3.3: Integration of a new mobile device by the KUI Session Service and service connections](image-url)
3.5 KUIEditor

The KUIEditor is a graphical user interface that provides all tools needed to model KUISystems (Fig. 3.4). The user can create, modify and remove all kinds of entities such as Actors, Places and Zones. The representation of the KUISystem is kept up to date by the uMove framework, allowing to monitor the behavior of the system, the contexts of the different entities and the relations between them.

Figure 3.4: The KUIEditor provides a GUI to model KUISystems
Chapter 4

KUIServices

In this chapter, we introduce the concept of KUIServices. KUIServices are context-aware applications that are build on top of uMove. The main components of a KUIService are the client part running on the mobile device of the user and the server part running constantly on the server interacting with the client parts (Sec. 4.2).

Having a clear notion of KUIServices allows us to formulate precise conventions (Ch. 9) and to provide useful interfaces and abstract classes. This improves transparency to the developers of KUIServices and allows our local market to deal with its “products” in a unified way.

4.1 System- and public KUIServices

We distinguish two different types of KUIServices: public KUIServices and system KUIServices.

1) Public KUIServices are the KUIServices that are managed by our KUIService management system. They are developed to offer a certain service to the user (e.g. a mobile tracking service or a
mobile to mobile communication service). In other words, public KUIServices are - depending on the situation - the gadgets the user can get from the KUIServiceMarket.

2) **System KUIServices** on the other hand are special KUIServices that enable or extend basic functionalities. In the current implementation, there are two KUISystemServices: the **KUISystemService** and our **KUIServiceManager**. The KUISystemService is used for the basic communication between server and clients (Ch. 7). The KUIServiceManager is the part of our KUIService management system that enables ad-hoc loading of (public) KUIServices. The system KUIServices are static and cannot be accessed by the user. On the server part they are loaded as soon as the SmartEnvironment is started, on the client side when the KUIServiceMarket application is started. Our KUIServiceMarket by default hides system services from the user (Sec. 8.1.2).

### 4.2 KUIServices components

The notion of KUIService includes four components. The **server part** and the **client part** include the code that is effectively run on the server resp. the client device. The **rule part** is used by the Coordination manager to evaluate the availability of the KUIService based on a certain situation, as defined in the Coordination project. The **meta data** part includes the information needed by the KUIServiceManager to load the KUIService and some additional information about the KUIService that is displayed to the user.

Information about implementing those parts is described in the Appendix B.

#### 4.2.1 Client part

The client part is a standalone application for Android, i.e an APK file. It is the part of the KUIService that is effectively downloaded, installed and run on the mobile device whenever the user chooses to install this KUIService from the KUIServiceMarket. Therefore, the client part will contain all user interfaces the service offers for the end user.

The development of the client part is free of any restrictions (no interfaces or abstract classes to implement or extend). But to enable the communication with the server, the class **AbstractIntentHandler** (Ch. 7) must be implemented in the application (and registered as Service in the Manifest, Sec. 2.2).

#### 4.2.2 Server part

The server part consists of a standard Java project where all classes are compiled individually. As the name suggests the server part is the part of the KUIService that is running on the server side. Once it has been loaded and started on the server, this part will keep running and interact with the client parts of the different users using the corresponding KUIService.

To allow the class loading (Sec. 5.2) into uMove environments, all classes of the server part must be placed in the package ch.unifr.kuiservice.serverpart. One class of the server part is declared to be the "main class". This class extends **AbstractServerService** and is treated in a special way

---

1. ch.unifr.umove.KUISystem.services.AbstractIntentHandler
2. ch.unifr.umove.KUISystem.services.AbstractServerService
when the service is loaded: it is the only class (of the server part) that is instantiated and registered to the communication (Sec. 5.2).

### 4.2.3 Rule

The rule of a KUIService is defined in a single Java class implementing the interface `IRule`\(^3\). The purpose of Rules is to give the developer of a KUIService a tool to “manually” enable or disable the application based on a certain situation. This is evaluated by the invocation of the `check` method declared in the interface.

Since the rule consists of a single class, we don’t create a new Java project but put it directly in the server part project. The rule class goes into the `ch.unifr.kuiserver.rule` package.

Listing 4.1 shows the rule class of the “SMSService”\(^4\). In this implementation the `check` method always returns an “EnabledServiceAction” action, meaning that the KUIService is always available.

```java
public class SMSServiceRule implements IRule {
    @Override
    public Action check(IEntity pEntity) {
        Action pRetval = null;
        if (pEntity instanceof Actor) {
            Actor pActor = (Actor) pEntity;
            Actor pParent = (Actor) pActor.getParentEntity();
            if (pParent != null) {
                pRetval = new EnableServiceAction(pEntity, "SMSService");
            }
        }
        return pRetval;
    }
    @Override
    public String getName() { return "SMSServiceRule"; }
    @Override
    public boolean isFinal() { return false; }
}
```

Listing 4.1: Rule class of the SMSService

### 4.2.4 Meta data

Each KUIService must provide certain meta data specified in a XML file. All those values are used by the KUIServiceManager to load the KUIService and/or by the KUIServiceMarket to display additional information about the KUIService. The following categories of values must or can be declared in the XML file:

\(^3\)ch.unifr.coordinaton.rules.interfaces.IRule

\(^4\)The SMSService is a public KUIService that has been developed in line with this project. It is used to test the comportment of our KUIService management system and can be found on the CD-Rom attached to this report.
• **Name** - The Name value is used to identify the KUIService within a SmartEnvironment. Therefore, it must be unique within all other names of KUIServices running on a server, as specified in §4.3. It is used by all parts of the KUIService management system.

• **Descriptive data** - These values deliver additional information about the KUIService. In the current implementation, the required values in this category are Version and Author. Additionally, the optional fields ReleaseDate and Description can be specified. Like the cover of a product in a supermarket, this data is available to the user of the KUIServiceMarket before he connects to a KUIService. It supports the user by finding adequate KUIServices that fit his needs or preferences. Therefore filling in also the optional fields is highly recommended when creating the meta data for a KUIService.

• **Load data** - As described in §4.3, all components of a KUIService must be accessible to the server when a KUIService is loaded. The values in this category specify file names and file paths used by the KUIServiceManager to find the needed components. The fields defined here are ServerPartRootDirectory, ServerPartMainClass, ClientPartDirectory, ClientPartAPK, ClientPartPackageName, RuleRootDirectory and RuleClass.

A more detailed listing of the different fields is presented in Sec. 5.1.

It is actually only this meta file that is directly passed to the KUIServiceManager when a new KUIService is loaded into the system. The other parts of the KUIService are loaded directly from the hard disk using the paths set in the XML file, as described in §5.

### 4.3 Preparing KUIServices to be loaded

Once all four components of a KUIService are developed, it can be dynamically loaded into any running SmartEnvironment (if the SmartEnvironment is running our KUIService management system). To successfully load a KUIService, all necessary files must be available on the server. These are namely the APK file of the client part, the compiled class files of the server part (including the rule class) and of course the XML file containing the meta data.

The loading process can then be started by the system administrator of the SmartEnvironment using the GUI of the KUIServiceManager. This is done by passing the XML file to the file manager in the GUI as explained in Sec. 5.5.
Chapter 5

KUIServiceManager (Server-Side)

Since the KUIServiceManager is a KUIService itself, it also consists of a server- and a client part (although the client part is handled in a special way, see Sec. 6). In this chapter, we have a closer look at the server part of the KUIServiceManager.

The server part of the KUIServiceManager has in particular three important tasks:

- **Load new KUIServices** - Whenever a new KUIService is loaded, the server part of the KUIServiceManager loads the corresponding classes of the server part into the virtual machine, instantiates the main class and binds it to the KUISystem running on the server.

- **Keep track of loaded KUIServices** - The server part of the KUIServiceManager keeps track of all KUIServices that have been loaded. This means that it keeps a registry for the different parts of the loaded KUIServices. This is needed for example to guarantee that all KUIServices within a SmartEnvironment are unique (Ch. 9).

- **Transmit client parts to the mobile devices** - Whenever a user has chosen a KUIService from the local market, the corresponding APK file has to be transmitted to the mobile device. Since we assume only a TCP connection between client and server (and not a connection to the Internet), the file has to be transferred using the existing communication channels provided by the Coordination manager. This means that the file has to be exchanged using an ObjectMessage as defined in [Hadorn, 2010]. The server part of the KUIServiceManager is in charge of answering requests for clients by preparing the corresponding ObjectMessage and sending it to the mobile device (Sec. 7).

5.1 Parsing the Metadata

The first phase when loading a KUIService consists of parsing the XML file holding all the meta information.

The parsed fields are the following:

1. **Name** - String holding the name of the KUIService.

2. **Version** - Integer values separated by dot-signs to indicate the current version of the KUIService.
3. **Author** - String holding the name of the developer of the KUIService (multiple author fields are allowed to enable having more than one author).

4. **ReleaseDate** (optional) - String representing the date when the KUIService has been released.

5. **Description** (optional) - String holding a (textual) description of what the KUIService does.

6. **ServerPartRootDirectory** - A file path to the root directory of the server part classes. The root directory does not include the folders representing the Java packages.

7. **ServerPartMainClass** - File name of the main class of the server part (extension excluded).

8. **ClientPartDirectory** - A file path to the folder where the APK file of the client part is stored.

9. **ClientPartAPK** - The filename of the APK file of the client part (extension included).

10. **ClientPartPackageName** - String holding the package name of the client part. Used within Android to install/uninstall the APK file.

11. **RuleRootDirectory** - A file path to the root directory of the server part classes. The root directory does not include the folders representing the Java packages. Since normally the rule is developed within the server part Java project, this field will generally be identical to the root directory of the server part.

12. **RuleClass** - File name of the rule class (extension excluded).

All those fields have to be put into a single XML element (the root element) named `<ServiceMetaData>` to indicate the beginning and the end of the meta data.

Most of the fields are required, but there are some optional ones. Missing or empty required fields raise an error and the loading process is stopped. Whenever an error occurs, it is displayed in a small separate frame - the `ErrorReporterGUI` - to inform the user. When parsing the data, the name of the KUIService is checked against the ones who are already loaded. Since all KUIServices within one SmartEnvironment are identified through their names, it is considered as an error if one tries to load a KUIService with a name that is already in use.

If the parsing succeeds, a `ParsedService` object is created. This object temporarily holds all parsed fields and is passed to the `ServiceLoader` who will perform the next steps to load the KUIService.

### 5.2 Loading and starting KUIServices

When the meta data is successfully parsed, the `ParsedService` object is processed by the `ServiceLoader`. He is in charge of finding the components according to the paths set in the XML file, loading the classes, starting the server part of the KUIService and binding it to the KUISystem of the server.

When loading new classes dynamically into the virtual machine, we need to be careful to respect the order of hierarchies in the classes. By default, a Java source file is compiled in a single class file. Such (regular) classes are encapsulated, meaning that they can be loaded in whatever order. However, not all source files can be compiled into a single class file. A common case where more than one class file is generated for a single source file is when the code contains nested classes. Additionally, auto-generated code often leads to multiple class files. In such cases, the Java compiler creates a (regular) class together with one or more inner class files. Here, the order of class loading depends,
since the regular class file needs to be loaded into the virtual machine before any of the corresponding inner class files can be treated.

The executed steps for loading a service are the following:

1. **Check the data** - Check if all required fields have been successfully parsed and contain valid values.

2. **Server part of the KUIService**
   
   (a) **Class loading** - All classes in the `ch.unifr.kuiservice.serverpart` package are considered to belong to the server part, and are therefore loaded here.
   
   - **Load all regular classes** - Load all classes that are not inner classes nor the main class of the server part.
   
   - **Load all inner classes** - Once all regular classes have been loaded, load the inner classes (except inner classes belonging to the main class of the server part).
   
   - **Load the regular main class** - Load the class that was declared to be the main class of the server part in the meta data.
   
   - **Load all inner classes of the main class** - Load the corresponding inner classes of the main class of the server part.

   (b) **Instantiate the main class** - Create a new instance of the main class. The `ServiceLoader` will use the default constructor for classes extending `AbstractServerService`.

   (c) **Bind the instance of the main class to the KUISystem** - The new instance of the main class is bound to the KUISystem running on the server. This is necessary to ensure accessibility of the KUISystem to the KUIService and to register the server part for communication. See also sec. 5.2.1 and ch. 7

3. **Client part of the KUIService** - see Sec. 5.4

4. **The rule** - Per definition, the rule consists of one class in the `ch.unifr.kuiserver.rule` package. However, it may be that this class has inner classes too.

   (a) **Class loading**
   
   - **Load the regular class** - Load the class that was declared to be the rule class in the meta data.
   
   - **Load the inner classes** - Load all inner classes belonging to the rule class.

   (b) **Instantiate the rule class** - Create a new instance of the rule class. The `ServiceLoader` will use the default (empty) constructor here.

   (c) **Apply the rule to the main class of the server part** - Bind the rule to the instance of the main class of the server part. This is important to make sure that the KUIService is enabled/disabled according to the evaluation of the rule.

5. **Create ServiceDescription** - If all previous steps succeeded, a `ServiceDescription` object is created. It holds any meta data about the KUIService that can be presented to the end user of the Service (e.g. the name of the service, the author, the description, etc.).

If any of those steps fail or produce an error, the whole loading procedure is aborted.
5.2.1 Mapping the ports

The Coordination manager defines that each KUIService is using its proper port for its TCP connections. This port number has to be set on the server side as soon as the KUIService is bound to the KUISystem. The client parts will adapt accordingly. So far, these port numbers have always been distributed in a hard-coded way, since the KUIServices were loaded in the very same manner.

To provide a dynamic handling of KUIServices, our management system has to coordinate the port registration. As it is the ServiceLoader class used to effectively load the KUIServices, it is also in charge of assigning the ports. Each KUIService is mapped to a different port. In the current implementation, the ports from 9030 to 9049 are reserved for system KUIServices and the ports from 9050 upwards are distributed to the public KUIServices.

The ports are selected by a simple counter, incremented for each KUIService that is loaded. The system KUIServices use static ports. Currently the KUISystemService is using 9030 and the KUIServiceManager is using 9031.

5.3 Applying the rules

The rules are important to evaluate, based on a certain situation, if a KUIService can be available for certain entities or not. However, the current implementation of the ServiceLoader couples the rule only with the main instance of the server part of a KUIService. This means that applying rules to individual entities within the KUISystem is not possible, as reported in chapter 10.

5.4 Preparing the Client File

When a new KUIService is loaded on the server, only the server part (and the rule) will be “installed” and started. The APK file of the client part stays on the server and will be transmitted to a mobile device whenever a user requests this service.

However, there are still some small steps to be made here. First, the ServiceLoader will test if the APK file actually exists and if it really is an APK file. Then a File object representing the file on the hard disk is stored into the KUIServiceManager. The KUIServiceManager maps this File with the service. This allows it to convert the APK file into a 7-bit safe Base64-encoded string before sending it to the devices. See chapter 7 for additional information about the Base64 encoding.

5.5 The GUI

As shown in Fig. 5.1, the GUI of the server part of the KUIServiceManager is divided into two parts. The upper part is used to load a new KUIService. The meta file is passed to the KUIServiceManager by entering the path into the textbox or by using the Browse... button. Once the XML file is declared, the KUIService can be loaded by clicking the LOAD Service! button.

The second part gives the system administrator of the SmartEnvironment an overview of the KUIServices that are currently loaded (they are all listed on the left part). By selecting an entry from this list, additional information about the KUIService is displayed on the right side (the additional
information consist namely of the meta data defined previously in the meta file).

The GUI of the KUIServiceManager is directly integrated into the KUIEditor (Sec. 3.5). It can be accessed using the menu “Services”.

Figure 5.1: The GUI of the server part of the KUIServiceManager
Chapter 6

KUIServiceManager (Client-Side)

In section 5.2, we described how the server parts of new KUIServices are dynamically loaded into the system by simply loading the corresponding classes, creating the necessary objects and binding them to the KUISystem. On the client side, we have a similar problem to solve. Whenever the user selects a KUIService from the local market, it has to be dynamically “installed”, started and especially be bound to the KUIMessaging system.

We first intended to use a very similar approach on the client side as we did for the server side, because we wanted to have compiled classes dynamically loaded on the mobile uMove. The advantage of having the client part in compiled classes too would be the simplicity of message passing between the server KUISystem and the mobile KUISystem. However, very soon we realized that this will not work. Of course dynamic class loading is supported by the Dalvik VM. But the problem lies in the way basic components of Android (Sec. 2.1) are embedded in the Android architecture. As explained in Sec. 2.2, all components within an application must be declared in the manifest file before the application is installed. Therefore it is not possible to dynamically add such components to an existing software. And since Android applications would be extremely limited without these elements, it would not make sense at all to restrict developers of KUIServices to work on this base for their client parts (e.g. without Activities, it would not even be possible to implement a GUI in the client part).

Therefore, we have chosen an other approach for the client side. In the current implementation, the client parts of KUIServices are standalone applications (i.e. whole APK files and not just compiled classes). This simplifies the tasks of installation and execution of the client parts, since Android offers basic functionalities to install and uninstall new applications from code. But on the other hand, it makes communication more difficult, as the client parts cannot be registered to the KUIMessaging system that easy anymore. Additionally, Android won’t let you simply access an installed application with another one without being trusted.

All these experiences and limitations brought us finally to the following solution: As stated above, the client parts are standalone applications. They run independently and are not directly registered to the KUIMessaging system. Instead, we provide a “generic MessageProcessor” that receives all incoming KUIMessages and distributes them to the corresponding client parts via Android Intents.

---

1 Dalvik is the virtual machine (VM) in Google’s Android operating system.
2 Android applications can be signed using a public RSA key. Applications signed by the same key are considered to trust each other.
In the other direction, it forwards the messages received from the client parts to the server parts. The topic of communication is discussed in detail in chapter 7.

6.1 Mobile KUI application

The mobile KUI application is in charge of enabling the communication from the server parts to the client parts of a KUIService, and vice versa. We implemented the mobile KUI application as an Android Service, since it is intended to run in background and does not need any user interaction. As this is the only task to be done, we have chosen to place this Android Service directly in the local market application. That way, users only have to install the KUIServiceMarket on their device to have the full functionalities discussed in this article (Ch. 1).

6.2 Loading the client parts of KUIServices

As mentioned above, loading the client parts of KUIServices on the mobile phone is rather easy. Androids PackageManager offers all functionalities needed to install or uninstall applications. For installation, it is enough to pass the path of the APK file on the SDCard to the PackageManager. For uninstalling an application, one needs to know the package name of the APK. This protects regular software to be randomly uninstalled by other (malicious) applications. In fact, this is the reason why the package name of the client part has to be declared in the meta data file of a new KUIService.

Listing 6.1: Sample Intent to install an application from “/sdcard/myapp.apk”

```java
//get an Uri to the file location:
Uri fileUri = Uri.fromFile(new File("/sdcard/myapp.apk"));
//create a new Intent with action set to ACTION_VIEW:
Intent intent = new Intent(Intent.ACTION_VIEW);
//set the Uri to be data and the type to be an APK:
intent.setDataAndType(fileUri, "application/vnd.android.package-archive");
//fire the intent:
startActivity(intent);
```

Listing 6.2: Sample Intent to uninstall an application with package name “ch.unifr.mypackage”

```java
//get an URI containing the package name:
Uri packageUri = Uri.parse("package:ch.unifr.mypackage");
//create a new Intent with action set to ACTION_DELETE:
Intent intent = new Intent(Intent.ACTION_DELETE, packageUri);
//fire the intent:
startActivity(intent);
```
Chapter 7

Server and client part communication within KUIServices

As mentioned in the abstract, the existing infrastructure is designed for static KUIServices that are directly imported or coded into the KUI server application and the mobile KUI middleware (Fig. 3.3). Therefore it is assumed that server and client part of all KUIServices are already available and installed on both sides when a mobile client connects to a SmartEnvironment. This allows to initialize the different parts and to register them to the Coordination manager in a static, hard-coded way.

Two different aspects have to be taken into account when talking about communication in uMove. One aspect is the system internal communication that takes place between the server and client part of the KUISystemService. The KUISystemService is, as the name suggests, a system KUIService defined at the Coordination level. It is used to monitor mobile devices, connect them to SmartEnvironments and to establish the communication sessions (Sec. 7.1). The other aspect is the high-level communication between the server and the client part of public KUIServices.

In this chapter, we explain how we managed to keep using the existing communication functionalities even with the dynamic loading of KUIServices. At first, we will have a quick look at the way the underlying architecture registers the server and the client parts to the Coordination manager. This allows us to better motivate the high-level solutions we have chosen in order to handle KUIService in an ad-hoc way and still support both aspects of communication.

7.1 Coordination Manager: Monitoring services

On a low-level base, all services are based on the same model which contains four components (Fig. 7.1): Service Object, Service Provider, Service Session and Service Client. The Service Object is the “processor” of the service. It is receiving, processing and sending the answer to any request coming from the client. This object (the class) is developed specifically for a service (e.g. menu service, meeting service). This is actually the class we called to be the main class of the server part in our definition of KUIServices (Sec. 4.2.2).

The Service Provider is the object that connects a service client to the service object. It uses a public port that listens to any request for a connection coming from a service client.
The **Service Session** is a dedicated object created by the service provider when a connection request comes in from a service client. A session is private and controls the communication between the Service Client and the Service Object. It is similar to the creation of a socket between two remote objects.

And the **Service Client** is the counterpart of the Service Session. It manages the communication between the application using a service on the mobile device and the service object on the server side. The Client Service is attached to a public port declared in a port list in the mobile KUISystem (i.e. attached to an actor as defined in the uMove project).

### 7.1.1 Connection to a service

When a client requests a connection to a specific service, three steps have to be done (Fig. 7.1):

1. The Service Client contacts the Service Provider using its public port.
2. The Service Provider creates a Service Session object.
3. The port of the Service Client is connected to the Service Session port.

Once the connection is complete, the Service Object and the Service Client can start to communicate. The protocol used is TCP-based (normally using wireless communication).

### 7.2 Monitoring dynamic KUIServices

Now we will bring the focus back to our KUIService management system. As our goal was to use the existing infrastructure, we had to find a work around for connecting the server- and client parts to enable communication. In this section, we explain how our system is solving this problem.

#### 7.2.1 Server side

As explained in Sec. 5.2, a KUIService can be dynamically loaded into a running KUI server application. This allows us to simply instantiate the main class of the service server part and get the **Service Object** of this Service. Then we need to bind the Service Object to the server-based KUISystem to let the KUIService be available for mobile KUISystems.
7.2: Monitoring dynamic KUIServices

7.2.2 Client side

Finding a solution for the client side turned out to be a bit more complex.

Problem

A previous implementation of a client base application simply created all Service Client objects as soon as the software was started, and then simply started the PeerFinder to start pinging for corresponding service server objects as shown in Fig. 3.3 provided by any SmartEnvironment. Again, this simple solution was possible as the client parts of the Services were accessible by the client base application.

In our implementation, no client parts of KUIServices are available at the initialization. Therefore it is not possible to create the Service Client objects when the system starts. This would not be a big deal, as we could simply create the Service Client ad-hoc when needed (as we do on the server part). But the situation we are dealing with is even more complicated, since the client parts of KUIServices as defined in Ch. 4 are standalone APK files (due to the problem of dynamic components in Android).

So we are in the undesirable situation that (a) the KUISystemService would need access to the client part to create the Service Client object and at the same time that (b) the client part - who can create the Service Client object - would need access to the mobile KUI application (resp. the KUISystemService) to enable the PeerFinder to ping for Service Providers that correspond. “Undesirable” because the mobile KUI application and the client parts of the KUIServices will all run in different independent Linux processes. So granting one of this applications access to the memory of any other would be a very difficult task (if not impossible at all due to the security policy of the Android operating system).

Solution

The solution we propose (and the one that is actually implemented in the current release of our application) is the following:

In order to bind the client part of a KUIService to the KUISystem and to enable communication with its server part, we have to create a Service Client object. As we receive the callback invocation of the PeerFinder when a new Service is found, we can create the Service Client ad-hoc when it is needed. But in order to really receive messages from the server part, we need to declare a MessageProcessor that treats them. Instead of having dedicated MessageProcessors for each KUIService, we use a generic one (the same for all KUIServices). This generic MessageProcessor receives all incoming messages from the server parts of the different KUIServices and dispatches them to the corresponding client parts. The generic MessageProcessor is linked to the AndroidMobile class and is part of the KUIServiceMarket. The KUIServices are identified by their name on the server part, and by their name together with the name of the SmartEnvironment on the client part.

That way, we are able to create Service Sessions for dynamically loaded KUIServices.

\[1\] A project called “SmartAppForAndroid”
7.3 Forwarding messages on the client side

The approach of having a generic MessageProcessor solves the problem of connecting the server and client parts of KUIServices. But this solution ends up in having all messages received on the mobile device processed by the AndroidMobile object. Therefore we need a way to forward the messages to the corresponding KUIService clients. To do so, we designed a small “bridge” (Fig. 7.2) for message passing that links the mobile KUI application with the client parts using implicit Intents (2.3).

The two classes that build a bridge are called MessageDispatcher on the side of the mobile KUI application and IntentHandler on the KUIService side. Both of these components are Android Services, since they do not need any user interaction and should run permanently in background.

For each active KUIService, a separate bridge is created (Fig. 7.3). The IntentHandler must implement the abstract class AbstractIntentHandler included in the uMove API in the package ch.unifr.umove.KUISystem.services. Developers of public KUIServices are in charge of correctly implementing the callback methods proposed by the AbstractIntentHandler for receiving messages. To send messages to the server part of the KUIService, the AbstractIntentHandler provides the methods textMessageToServer and objectMessageToServer, both taking one argument (a String holding the content for TextMessages resp. a String holding the base64-encoded representation of a serializable object for ObjectMessages).

There are currently three types of messages implementing the IMessage interface. All of them are supported by our project, i.e. can be dispatched by our generic MessageProcessor. The three types are:

- **TextMessages** - TextMessages are simple messages that can hold a String value. TextMessages are forwarded to the corresponding KUIService by the generic MessageProcessor.

- **ObjectMessages** - ObjectMessages can be used to transfer any serializable Java object. Like TextMessages, the generic MessageProcessor is in charge of delivering them to the correct KUIService.

---

2Functionalities to transform a serializable object into a base64-encoded String and vise-versa are provided by the FileBase64Serializer class in ch.unifr.umove.KUISystem.services
7.3: Forwarding messages on the client side

Figure 7.3: “Bridges” for passing Intents build between the MessageDispatcher and IntentHandlers

- **KUISystemMessages** - Such messages originate from the KUISystem and signalize changes in the state of the KUISystem. These messages are broadcasted to all KUIService clients.

All different types of intents used in this context are defined in a static class that has been included in the uMove framework to unify the communication.

7.3.1 Base64 encoding to serialize Objects

Android Intents are designed to be rather lightweight objects. They do not support transporting whole objects. So to deliver ObjectMessages over our Intent-driven bridge, we need to encode them in a different way. The ObjectMessage class, as defined in the Coordination project, accepts only serializable objects as content. Therefore we can assume that the object to be delivered will also be serializable and we transform the corresponding object into a character-stream on one side, deliver this stream as we do for TextMessages in an Intent to the other side of the bridge and recreate an object out of it. In order to avoid errors due to “dangerous” characters, we apply a Base64 encoding resp. -decoding on both sides. This results in a safe 7bit-ASCII encoded stream.
Chapter 8

KUIServiceMarket - a local market application

In this chapter we present the KUIServiceMarket, an Android application that acts as the base application on the client side. Its main purpose is to be used as access gateway to the different KUIServices available in the SmartEnvironments. But it also includes other parts of our KUIService management system and the basic components needed by the infrastructure of the Coordination project (i.e. the AndroidMobile class).

The main tasks of the KUIServiceMarket are:

- **Interaction with the user**
  - **Provide information to the user** - The user has the possibilities to get information about the KUIServices and SmartEnvironments he is in. In the current implementation, the details about SmartEnvironments are available only if the device is already connected to them. The descriptive data about the available KUIServices are always accessible, no matter if the KUIService has been downloaded/installed or not.
  - **Keep the user up to date** - The application offers the service to notify the user whenever a Service has become available or is no longer available within the SmartEnvironment.
  - **User interface for handling connections** - The user interface of the KUIServiceMarket allows the user to manage the connections to the SmartEnvironments.
  - **User interface to locally manage KUIServices** - A user interface is provided that gives to user the possibility to install and uninstall KUIServices.

- **Provide the basic components**
  - **Local KUISystem** - The KUIServiceMarket runs and maintains the local KUISystem (actor, observer, senget) that is needed by the Coordination project, e.g. to create the actor stubs.
  - **PeerFinder** - The KUIServiceMarket also includes the handling of the PeerFinder[1]. Additionally it provides all callback functions to the PeerFinders events.

[1]The PeerFinder makes part of the Coordination manager and is in charge of detecting SmartEnvironments and KUIServices.
8.1 The user interface

In this section we present the user interface of our KUIServiceMarket. Our focus here was to keep the interaction simple and to maximize the usability. The GUI consists of four Activities: the MainActivity, SmartEnvActivity, ServiceActivity and ServiceInfoActivity. Besides, a small options menu is proposed to the user to allow some basic settings.

8.1.1 Activities

MainActivity

The MainActivity is the first screen the user sees whenever he starts the KUIServiceMarket. Its appearance is divided into four parts (Fig. 8.1):

1. **Peer finder** - The first part contains the handling of the PeerFinder. In the current implementation, the PeerFinder can simply be switched on and off. If it is on, it will continuously scan for new SmartEnvironments and KUIServices and automatically connect to any SmartEnvironment found. If the user switches the PeerFinder off, all running connections to SmartEnvironments are terminated (Sec 8.3).

2. **Smart Environments** - This part lists all SmartEnvironments the user is currently connected with. When clicking on a label in the list, the corresponding SmartEnvironment is displayed in the SmartEnvActivity. The label consists of the name of the SmartEnvironment, its IP address and the basic Port (e.g. MySmartEnvironment@10.0.0.5:9000).

3. **Services** - This part only contains a button labeled Manage Services. It is only enabled if the mobile device is connected to at least one SmartEnvironment. When clicked, it brings the user to the ServiceActivity.

4. **Minimize & stay connected** - This button is for convenience only (Sec. 8.4).

SmartEnvActivity

The SmartEnvActivity is displayed to the user if he selects a SmartEnvironment from the list in the MainActivity. In the current implementation, the fields that are displayed to the user are the following (Fig. 8.2):

1. **Name** - The name of the SmartEnvironment

2. **IP** - The IP address of the SmartEnvironment within the network (e.g. WIFI).

---

2In the current implementation, there is another field UUID that shows the UUID of the SmartEnvServiceClient. This field is rather for debugging and could be removed in a next release.
8.1: The user interface

Figure 8.1: The appearance of the MainActivity when (a) the PeerFinder is off and (b) the PeerFinder is on and the device is connected to a SmartEnvironment

3. **Port** - The port of the SmartEnvironment

4. **Info** - This field is intended to hold a textual description of the SmartEnvironment. It is currently not implemented, since this feature is not supported by the current server application.

5. **Services** - A list of the KUIServices that are currently provided by this SmartEnvironment. The system KUIServices are shown depending on the user options (Sec. 8.1.2).

**ServiceActivity**

When selecting the **Manage Services** button in the MainActivity, the user gets to the ServiceActivity. This is the “market” where all KUIServices from all connected SmartEnvironments can be selected, downloaded and installed. The list of KUIServices is separated into the category **Available Services** and **Installed Services**, depending if the KUIService has been downloaded and installed or not. The system KUIServices are shown in a third category called **System Services**, if they are shown at all (Sec. 8.1.2).

When a KUIService is selected from the list, the available commands are proposed to the user via a context menu. The different choices are enabled according to the state of the corresponding KUIService. These are the options implemented in the current release:

- **Request/Display additional information** - This option allows the user to get the descriptive data of a KUIService. In fact, it starts the ServiceInfoActivity (8.1.1).

- **Download + Install** - When selected, the client part of the appropriate KUIService is requested from the KUIServiceManager of the corresponding SmartEnvironment. Once downloaded, it is automatically installed on the mobile device and registered for the KUIMessaging system.

---

3In fact, this is the port of the SmartEnvServiceClient. Remember that all KUIServices use a different Port for communication with the SmartEnvironment. [Hadorn, 2010]
8.1: The user interface

- **Uninstall** - Uninstalls the corresponding KUIService and deletes the APK file of its client part. The current release of the KUIServiceMarket always re-downloads the APK file of a KUIService when the user wants to install it. That way we can make sure that the new version is distributed when a new release of a KUIService is available. Therefore it would not make sense to leave the APK file on the SD card of the device.

- **Start** - This option is for convenience only. It allows to directly start a KUIService from the KUIServiceMarket. Of course, since the client parts of our KUIServices are “stand alone” applications, they will also appear in Androids “start menu” and can be launched like any other application from outside our local market.

- **Cancel** - This option is for convenience only. When clicked, no action is taken except of closing the current menu leading the user back to the ServiceActivity. In fact, this option simulates a click on the hardware key “Back” on Android mobiles.

**ServiceInfoActivity**

The ServiceInfoActivity is used to display meta data describing a KUIService. All those information are directly taken from the meta information that has been specified by the developer of the KUIService when setting up the meta XML file.

To avoid unneeded traffic between the server and the client, the ServiceDescription objects that hold those information are not automatically transferred to the user when the new KUIServices become available. Instead, it is requested from the SmartEnvironment when needed. This means that each time the ServiceInfoActivity is called to display the description of a KUIService, it checks if the needed information is already available in the memory or not (Sec. 8.2). Depending on this test, the data is displayed to the user (Fig. 8.4) or requested from the KUIServiceManager of the SmartEnvironment (Ch. 7).

In the later case, the user is informed about the request via an Android Toast element [Developers, 2010c]. As soon as the data has been transferred, the label *Request additional information* is changed
8.1: The user interface

8.1.1 ServiceActivity

8.1.1.1 The ServiceActivity showing the available KUIService SMSService and the system KUIServices.  

8.1.1.2 The context menu of the SMSService.  

8.1.1.3 The ServiceActivity after the SMSService has been installed (not showing the system KUIServices).

into Display additional information in the context menu of this KUIService, to indicate the user that the data is now available and can be displayed. 

A detailed description of the different fields is listed in Sec. 5.1.

8.1.2 Options menu

Android offers the possibility to set up an options menu for Activities. The options menu is the one that pops up at the bottom of the screen when the user hits the hardware key “Menu” available on all Android mobile phones.

The KUIServiceMarket offers a small options menu that can be accessed from within all different Activities that belong to the application (Fig. 8.5). It allows the user to change a few preferences that influence the behavior of the KUIServiceMarket.

The options offered are the following:

- **Notify SmartEnv/Don’t notify SmartEnv** - This option can be switched on and off. If it is set to “Notify SmartEnv”, the user is informed by a Notification in Android’s Notification Bar each time a new SmartEnvironment has been detected or a connected SmartEnvironment has been lost. The icon used in the Notification is the same as for the application itself to directly inform the user that this Notification comes from the KUIServiceMarket. When the user selects the Notification, he is brought to the MainActivity where he can directly see changes applied to the list of SmartEnvironments. This option is by default set on.

- **Notify Service/Don’t notify Service** - This option, similar to the one above, informs the user whenever a new KUIService is available from the market, or if an available KUIService has been lost. This option is by default set off.
8.2 Tracking the available Services

The KUIServiceMarket needs to keep track of all KUIServices that are offered, as it needs to know at any time what its current state is. This is done by a component called LocalServiceManager which is directly implemented in the KUIServiceMarket. It maintains a local registry of LocalServiceTracker objects (see §8.2.1). The LocalServiceManager is the main instance to control the KUIServices. For each KUIService that is detected on the client side, it creates a new LocalServiceTracker instance. This tracker is organized in the local registry as long as the KUIService is available (see also section §8.3).

8.2.1 LocalServiceTracker

LocalServiceTracker objects are objects that represent a KUIService within the market. Whenever the state of a KUIService is changed (by the user or by external events), its corresponding tracker object adapts to the new situation. Therefore they can be compared to an ArrayAdapter backing up a Java list or the Actor stub in a server-based KUISystem that represents the Actor object in the mobile
The tracker basically stores two different states of the KUIService. One state reports if additional descriptive meta data for this KUIService has already been downloaded from the SmartEnvironment or not (this is the test evaluated by the ServiceInfoActivity [8.1.1]). The other state represents the real condition of the KUIService (Available, Downloaded or Installed).

Besides of keeping track of the KUIServices states, the LocalServiceTracker is also used to store all different items related to the KUIService. It mainly stores the String describing the package name of the application (Sec. 8.3), the ServiceDescription object (if it has already been downloaded) and keeps the path to the APK file on the SDCard once the client part of the KUIService has been installed. The current implementation also stores few additional objects that can be useful for debugging purposes.

### 8.3 Removing footprints

As stated at the beginning of the report, a very important goal for our KUIService management system was to leave no footprints. This means that ideally after a KUIService is no longer available, for instance if the user leaves the SmartEnvironment, the phone is left in the exact same condition as it was before the KUIService was found, no matter how many times it has been installed, used or uninstalled in the meantime.

Therefore, a special method called RemoveFootprintsForService has been developed in the KUIServiceMarket. This method is called each time the PeerFinder invokes the corresponding callback (i.e. when losing durably the SmartEnvironment connection). The RemoveFootprintsForService method is called iteratively for each KUIService that has been offered from this SmartEnvironment. In a first step, the application is uninstalled from the device if it was installed before (see below). Second, the APK file is deleted from the local SDCard if it has been downloaded. And in a last step,
the LocalServiceTracker is removed from the registry of the LocalServiceTracker. The rest of the job is done by the GC\(^4\) of the Dalvik VM.

This method guarantees in theory that no footprints are left over. However, the KUIServiceMarket cannot guarantee that no changes have been done by the services to the device at all. Since the client parts of the KUIServices are standalone applications, it is way beyond the scope of our implementation to control the internal behavior of KUIServices (e.g. temporal data written on the SDCard by the service). Therefore it is the responsibility of the developers to remove their own internal footprints.

8.3.1 User confirmation needed for (un)installing applications

There is another topic that motivates the statement of not leaving any footprints in theory. Our market application delegates the tasks of installing or uninstalling applications to the Package manager of the Android operating system, since it is the only way of modifying the installed packages allowed by Android. Due to security reasons anytime a package has to be installed, modified or uninstalled the user is asked for permission (Fig. 8.6).

![Figure 8.6: Androids Package Manager asking for user permission to (a) install resp. (b) uninstall an application](image)

Asking for permission when installing a new KUIService is not a problem at all. The KUIService is installed or not depending on the users choice. If the user cancels the installation progress, nothing happens and the KUIService remains in the Available state. The same applies for uninstalling a KUIService from within the ServiceActivity. On the other hand, the needed user intervention causes our footprint policy to fail. As we need to uninstall KUIServices, the user will again be asked for confirmation. Of course, if he denies uninstalling the app, it will remain installed on the device (the APK file on the SDCard and the LocalServiceTracker object are deleted in any cases). However, next

\(^4\)Garbage collector
time the user loads the service, the package will be overwritten on the phone.

8.4 Sticky MainActivity

As explained in [Developers, 2010a], all Android components follow a certain lifecycle. Since Android is designed to run on mobile devices, it is optimized not to waste resources such as memory. Therefore, Activities that do not have the current focus of the user may be paused or even stopped by the operating system at any time given. This feature should be taken into account by any developer of Android applications. However, there is the possibility to declare components to be “sticky” (needs user permission at installation time). Such components are paused or stopped only if memory space is very low.

Since the MainActivity of our market application hosts the AndroidMobile instance, the MessageDispatcher and other important parts of the mobile KUI application, stopping it would mean that all established connections to SmartEnvironments fail and would therefore cause all KUIServices to be unavailable. To avoid this, we declared the MainActivity of the KUIServiceMarket to be sticky.

But being sticky does not imply that the application is never stopped at all. The user still has the choice to exit it manually by hitting the hardware “Back” button on the mobile device a couple of times (depending on the Activities in the stack of the current task). According to some user tests, it seems that pressing the back button is the favorite way of getting back to the home screen of the device. To prevent users to quit the market unintended, we intercept this key event and replace it by the key event fired when the “Home” button is pressed. We also added a “Minimize + stay connected” button at the bottom of the MainActivity that does the same action.
Chapter 9

Conventions

In this chapter, we provide a listing of all conventions created for the KUIService management system. Those conventions give some guidelines for KUIService developers on what must be respected when designing and implementing a service.

9.1 General conventions

- Only applications that meet the definitions of KUIServices (Ch. 4) may be used with our KUIService management system.
- On the client side, SmartEnvironments are identified by their name. Therefore, it is not possible to connect the mobile device to multiple SmartEnvironments declared with the same name.
- Inside a SmartEnvironment, KUIServices are identified by their name. It is not possible to load a new KUIService into a running server application if a KUIService with this name is already running.
- KUIService names must not be equal to a name of a system KUIService and is not allowed to contain a “At” sign (@). This character is used to separate KUIService and SmartEnvironment names (e.g. SMSService@MySmartEnvironment).

9.2 Conventions for KUIService development

- Server part
  - All classes of the server part must be placed inside a package called ch.unifr.kuiservice.serverpart.
  - Exactly one class must be declared to be the main class of the server part. This class must extend the abstract class AbstractServerService.
  - Only the main class is instantiated when the KUIService is loaded. The developer is in charge that the whole server part is set up correctly when calling the constructor of the server part.
  - The name of the service used inside the code of the server part must correspond to the name specified in the meta data.
• **Rule**
  
  – The rule part consists of exactly one single class.
  
  – This class must implement the `IRule` interface.

• **Client part**

  – The client part is a stand-alone Android application that is based on the Android 2.0 API framework.

  – The APK file must not be signed or otherwise compressed, encrypted or similar.


Chapter 10

Limitations and future work

The KUIService management system described here is fully implemented and the source code can be obtained from the CD-ROM attached to this report. All features presented in the previous chapters are working properly (unless stated otherwise) and the implementations meet the goals formulated at the beginning of the project. However, since the main purpose was to deliver a “proof of concept” for the uMove framework and also due to the fact, that we describe the very first release of this code, there are of course certain aspects that could be improved in future revisions.

10.1 Limitations

The stability of our KUIService management system has been evaluated by performing extensive usage tests.\(^1\) Therefore we guarantee a correct behavior of our code (assuming that the conventions of Ch. 9 are respected).

However, we are not able to guarantee all aspects that are related to performance. Of course our code was designed to be robust but due to a lack of material, no test cases could have been done regarding those aspects. I.e. we can only assume that our project is able to handle “busy” situations. This includes the cases were e.g. numerous clients are connected to a single SmartEnvironment simultaneously, huge amounts of data has to be transferred at the same time and/or a huge mass of entities are tracked within one KUISystem. It is not the uMove framework, but rather the comportment of the Android architecture that raises those questions. The complex process of resolving implicit Intends in Android could lead to a “message overflow” in our message passing workaround used on the client side (Ch. 7).

10.2 Future work

As mentioned in the introduction to this chapter, there is room for improvements. In this section, we list a few aspects that we think could be revised for future releases.

\(^1\)The tests included all predictable use cases and were successfully executed using different hardware on the server side, several WIFI networks for communication and a “Motorola Milestone” mobile phone using the Android 2.0 framework as client device.
Loading the KUIServices into the KUIServiceManager

A more flexible handling of the different components of the KUIServiceManager would be suitable. Instead of defining all meta data of a KUIService into a XML file, we could propose a “LoadWizard” that would guide the system administrator through the process of loading (ask the operator to point to the different parts of the KUIService and directly ask input for the descriptive fields).

As described in Sec 5.2, future work could also focus on the possibility to apply KUIService rules to single entities within the KUISystem of the server. A good solution could be to present a list with the current entities to the system administrator. From this list, he could then choose all entities that should be linked to this rule.

User interaction with the KUIServiceMarket

Another topic that could be improved is the user interface of the KUIServiceMarket. For example, the list of available KUIServices is presented to the user in a rather simple way (Sec. 8.1.1). It would be nice to introduce categories that group together KUIServices offering similar functionalities, as we have it in the Android market. This would support the user in finding adequate KUIServices based on the current needs.

Another important interaction concern is the handling of the PeerFinder. For future releases it would be preferable to let the user choose if he wants to get connected to newly found SmartEnvironment automatically (as it is the case in the current version) or not and/or add the feature to connect to single SmartEnvironments manually.

Plugins to develop KUIServices

Besides the extensions to the usability for the end user and the system administrator, also the development process for the developers of KUIServices could be further supported. It would be great to have plugins for Java IDEs such as Netbeans or Eclipse to provide template projects for all parts of the KUIServices.

---

2Integrated Development Environment
Chapter 11

Conclusion

In this report, we introduced a management system for KUIServices to be used with the uMove framework. We proposed a clear definition of the context-aware applications that we call KUIServices and then we focused on the way these KUIServices are handled by our management system.

On the server side, the KUIServiceManager extends the existing infrastructure by providing dynamic loading of KUIServices into running SmartEnvironments. On the client side we introduced the KUIServiceMarket, a local market application that controls the connections to the SmartEnvironments and allows the user to get access to the KUIServices available from them. Appendix B, together with the conventions in Ch. 9 and the outlined structure of the meta data files in Sec. 5.1 give all information needed to develop KUIServices.

The work on this project was very interesting, although not always easy. Especially at the beginning of the project, the regular updates on the uMove and Coordination projects, coupled with a few minor external problems (namely with the subversion server), slowed down the progress. Nevertheless the gain of knowledge was huge. Besides the specific insight into context-aware software, uMove, the Coordination project and of course the Android architecture, it will also influence the way of working in future thesis. Not only the fact of focusing for a rather long time on a single project but also working in a team and continuously adapting to changing needs was an experience that might pay off well also looking forward to future projects outside the academic studies. It has been a completely unknown situation not to start a project from scratch but to get involved into an already running project.

At this point I would like to thank all people who supported me during this thesis. Special thanks go out to Benjamin Hadorn, Pascal Brügger and Prof. Béat Hirsbrunner for the extensive, useful and reliable help offered at any time during the thesis.
Bibliography


P. Bruegger, B. Hadorn, and B. Hirshbrunner. Ssp: Smart service provider - a smart environment providing contextual services on android mobile devices. UIC 2010, Springer LNCS, October 2010. [cited at p. 8]


Appendix A

How to use our management system

A.1 Set up a client

Requirements:

- **Mobile device** - You will need a rooted\(^1\) smartphone running the Android 2.0 operating system or higher. (In fact, our implementations are build to be backward compatible. But no tests have been made to versions prior to 2.0). The mobile device must have a SDCard installed.

- **Underlying architecture** - The uMove and Coordination API is used by our system. The needed files `uMove2.jar`, `Coordination.jar` and `MobileMonitoring.jar` can be found on the CD-ROM attached to this report.

- **KUIServiceMarket** - The application `KUIServiceMarketForAndroid3.apk` is also available from the CD-ROM.

Installation:

1. Prepare the phone by following the steps explained in the appendix of \cite{hadorn2010}. The `uMove2.jar`, `Coordination.jar` and `MobileMonitoring.jar` have to be installed before running the KUIServiceMarket APK file.

2. Install the KUIServiceMarket application on the phone. You may use the ADB\(^2\) to do so or run the application from NetBeans (NetBeans will copy and install the APK on the plugged phone).

All required parts of the client side of our KUIService management system are included in the KUIServiceMarket application. You may start your local market now and start scanning for SmartEnvironments.

A.2 Set up a SmartEnvironment (Server)

Requirements:

\(^1\)“Rooted” means that you have root access to the device. A guide how to achieve that can be found in the appendix of \cite{hadorn2010}.

\(^2\)Android Debug Bridge, a command-line based tool to interact with plugged Android devices. The ADB is part of the Android Developer Kid (ADK) that can be downloaded for free from [www.android.com](http://www.android.com)
A.2: Set up a SmartEnvironment (Server)

- **Server** - The KUIServerApplication runs generally on a standalone computer.

- **IP-based communication** - The communication between the server and the mobiles is TCP-based and wireless. The wireless network needs to be accessible by the Android device and the ports from 9030 and 9050 upwards must be opened (Sec. 5.2.1).

**Installation:**

1. Compile the KUIServerApplication and run it. (For compilation, make sure to include the uMove, Coordination and MobileMonitoring jar files).

2. To start a SmartEnvironment, load or create a KUISystem using the Menu of the GUI of the server application.

3. To load a KUIService, open the KUIServiceManager from the Menu and point the service loader to the corresponding XML file holding the meta data (Sec. 5.5).
Appendix B

How to create a KUIService

For this guide we assume that you have:

1. Successfully copied and/or installed the uMove framework and the implementations presented in this report.
2. Installed a compatible ADK\textsuperscript{1} and set up a valid Android Platform 2.0 or higher.

Make sure to fulfill the conventions for KUIService development in Sec.\textsuperscript{9.2} We highly recommend using an advanced IDE\textsuperscript{2} such as Netbeans or Eclipse.

B.1 Server part

1. Start a new Java project (you won't need a runnable main method).
2. Include the libraries of the underlying projects.
3. In your project, create a package called \texttt{ch.unifr.kuiservice.serverpart}
4. Create a Java class in this new package and make it extend \texttt{AbstractServerService} (make sure to adjust your import statements). This class will be the main class of the server part.
5. Implement the abstract methods in the main class and create other classes and/or a GUI if you like. Remember that all classes must be contained in this same package!

B.2 Client part

1. Start a new Android Project.
2. Include the libraries of the underlying projects.
3. Create a class extending \texttt{AbstractIntentHandler} and implement the abstract methods.

\textsuperscript{1}Android Developer Kit
\textsuperscript{2}Integrated Development Environment
4. Specify this class as Service in the manifest file and register the three IntentFilters in Listing B.1. Make sure to replace MYKUISERVICE with the correct name of the KUIService, written in capital letters (e.g. “SMSSERVICE” for SMSService).

5. The rest of the client part can be designed completely independently.

Listing B.1: IntentFilters to be registered for IntentHandlers

B.3 Rule

1. Go back to the Java project of your server part.
2. Create a new package called ch.unifr.kuiserver.rule.
3. Create a Java class in this new package and make it implement the IRule interface.
4. Implement the abstract methods.

B.4 Meta data

The meta data will normally be the last component that is set up, since only at the end of development all information may be present. Beware that the paths you need to set for the client and the server part refer to the path of the file system of the server the KUIService will be running on. This means that under certain circumstances it will be system administrator of the SmartEnvironment that needs to specify the paths according to the place were the files have been stored.

1. Create a new XML file and specify it to use XML 1.0 (e.g. by placing `<?xml version='1.0' encoding='utf-8'?>` as first input line).
2. Create a root node called <ServiceMetaData>.
3. Inside the root node, create all nodes described in Sec. 5.1 and specify the corresponding values.

Listing B.2 shows the meta data file of the SMSService.

```xml
<?xml version='1.0' encoding='utf-8'?>
<ServiceMetaData> <!-- indicating start of meta-data -->
  <!-- the name of the service [any string] -->
  <Name>SMSService</Name>
  <!-- the version of the service [int values separated by dot-signs] -->
  <Version>1.0</Version>
</ServiceMetaData>"
Listing B.2: The meta data file of the SMSService