simpli·city

The Road User Information System Of The Future

WP6 – Personal Mobility Assistant

D6.3.2: Mobile Application Runtime Environment Prototype II

Deliverable Lead: ASC

Contributing Partners: ASC, TALK, TIE

Delivery Date: 04/2015

Dissemination Level: Public

Version 1.0

This deliverable describes the work carried out during the development of the final prototype of the Personal Mobility Assistant component of the SIMPLI-CITY platform. It specifies the scope of this final version and the degree of fulfilment of the requirements to be covered by the component. It specifies how to install and execute the different subcomponents implemented.

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 318201.

Document Status	
Deliverable Lead	Jan Reehuis, ASC
Internal Reviewer 1	Philipp Hoenisch, TUV
Internal Reviewer 2	Antonio Paradell, WORLD
Type	Deliverable
Work Package	WP6: Personal Mobility Assistant
ID	D6.3.2: Mobile Application Runtime Environment Prototype II
Due Date	31.03.2015
Delivery Date	30.04.2015
Status	For Approval

Document History	
Contributions	V0.1, Stefan Schulte, Philipp Hoenisch, TUV, 04.12.2012, Added document structure. V0.2, Jan Reehuis, ASC, 27.02.2015, Initial Document Structure. V0.3, Jan Reehuis, ASC, 10.03.2015, Detailing the Sections V0.4, Pontus Lindström, TALK, 24.03.2015, Detailing Section 5.2.2 V0.5, Arturo Brotons, TIE, 02.04.2015, Detailing the Section 5.2.1 V0.6, Jan Reehuis, ASC, 07.04.2015, Finalizing the Document for Review V0.7, Arturo Brotons, TIE, 08.04.2015, Corrections before Review V0.8, Jan Reehuis, ASC, 13.04.2015, Corrections according to the internal Review V0.9, Jan Reehuis, ASC, 23.04.2015, Corrections according to the internal Review V0.10, Arturo Brotons, TIE, 23.04.2015, Corrections according to the internal Review
Final Version	V1.0, Jan Reehuis, ASC, April 27st 2015

Note

This deliverable is subject to final acceptance by the European Commission.

Disclaimer

The views represented in this document only reflect the views of the authors and not the views of the European Union. The European Union is not liable for any use that may be made of the information contained in this document.

Furthermore, the information is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user of the information uses it at its sole risk and liability.

Project Partners

Vienna University of Technology (Coordinator), Austria

Ascora GmbH, Germany

TIE Nederland B.V., The Netherlands

Technische Universität Darmstadt, Germany

IBM Research – Ireland
Smarter Cities Technology Centre

Forschungsgesellschaft Mobilität, Austria

Talkamatic AB, Sweden

Atos Worldline, Spain

Centro Ricerche FIAT, Italy

SRM – Reti e Mobilità, Italy

Executive Summary

This deliverable describes the work carried out during the development of the final prototype of the Application Runtime Environment component of the SIMPLI-CITY Personal Mobility Assistant.

The document starts by introducing the Application Runtime Environment component and describing the scope of this final prototype.

Afterwards, the degree of fulfilment of each requirement to be covered by the component and specified in the Requirements Analysis Report (D2.3) is described.

In this final prototype, the functionality of the Application Runtime Environment is given. It provides installation and execution of the Application Runtime Environment, which can be used by the user via the Multimodal Dialog Interface of the Personal Mobility Assistant. A direct access to the Application Runtime Environment is not available as it is a component which handles the PMA internals and does not offer an end user interface.

The latest sections of this document describe how potential users (i.e. SIMPLI-CITY users and developers) can prepare, install and execute the Application Runtime Environment component. For this, a step-by-step process to install and make use of the prototype is provided.

This deliverable D6.3.2 will conclude the work on the Application Runtime Environment.

Table of Contents

1 Introduction

SIMPLI-CITY – The Road User Information System of the Future – is a project funded by the Seventh Framework Programme of the European Commission under Grant Agreement No. 318201. It provides the technological foundation for bringing the “App Revolution” to road users by facilitating data integration, service development, and end user interaction.

Within this document, the final prototype of the Application Runtime Environment will be presented. The document accompanies the corresponding software prototype, which is the main content of the deliverable.

1.1 SIMPLI-CITY Project Overview

Analogously to the “App Revolution”, SIMPLI-CITY adds a “software layer” to the hardware-driven “product” mobility. SIMPLI-CITY will take advantage of the great success of mobile apps that are currently being provided for systems such as Android, iOS, or Windows Phone. These apps have created new opportunities and even business models by making it possible for developers to produce new apps on top of the mobile device infrastructure. Many of the most advanced and innovative apps have been developed by players formerly not involved in the mobile software market. Hence, SIMPLI-CITY will support third party developers to efficiently realise and sell their mobility-related service and app ideas by a range of methods and tools, including the Mobility Services and App Marketplaces.

In order to foster the wide usage of those services, a holistic framework is needed which structures and bundles potential services that could deliver data from various sources to road user information systems. SIMPLI-CITY will provide such a framework by facilitating the following main project results:

- Mobility Services Framework: A next-generation European Wide Service Platform (EWSP) allowing the creation of mobility-related services as well as the creation of corresponding apps. This will enable third party providers to produce a wide range of interoperable, value-added services, and apps for drivers and other road users.
- Mobility-related Data as a Service: The integration of various, heterogeneous data sources like sensors, cooperative systems, telematics, open data repositories, people-centric sensing, and media data streams, which can be modelled, accessed, and integrated in a unified way.
- Personal Mobility Assistant: An end user assistant that allows road users to make use of the information provided by apps and to interact with them in a non-distracting way – based on a speech recognition approach. New apps can be integrated into the Personal Mobility Assistant in order to extend its functionalities for individual needs.

To achieve its goals, SIMPLI-CITY conducts original research and applies technologies from the fields of Ubiquitous Computing, Big Data, Media Streaming, the Semantic Web, the Internet of Things, the Internet of Services, and Human-Computer Interaction. For more information, please refer to the project website at http://www.simpli-city.eu.

1.2 Deliverable Purpose, Scope and Context

The purpose of this document is to provide the means to use the final prototype of the Application Runtime Environment and exploit its functionalities. For this, the scope and requirements of the Application Runtime Environment and this prototype, the requirements and preparations for users and developers, an installation and usage guide as well as a brief overview of the envisioned enhancements for this second prototype of the Application Runtime Environment, are provided.

The final Application Runtime Environment prototype is the outcome of the discussions and implementation work done in project months 18 to 30. It provides the implementation of the functionalities of the Application Runtime Environment as provided with SIMPLI-CITY deliverables D3.2.1 (Functional Specification), and D3.2.2 (Technical Specification). This deliverable D6.3.2 will conclude the work on the Application Runtime Environment.

1.3 Document Status and Target Audience

This document is listed in the Description of Work (DoW) as “Public”, since the content is functional complete (beta prototype) but subject to changes, depending on the upcoming demands in other work packages and/or issues with the current implementation.

1.4 Abbreviations and Glossary

A definition of common terms and roles related to the realization of SIMPLI-CITY as well as a list of abbreviations is available in the supplementary document “Supplement: Abbreviations and Glossary”, which is provided in addition to this deliverable.

Further information can be found at http://www.simpli-city.eu.

1.5 Document Structure

This deliverable is broken down into the following sections:

Section 1 provides an introduction for this deliverable including a general overview of the project, and outlines the purpose, scope, context, status, and target audience of this deliverable.

Section 2 provides an overview of the scope and relationship of the prototype, showing were the Application Runtime Environment fits into the overall SIMPLI-CITY software framework and the outcome of the final prototype. Furthermore, an assessment of the requirements covered by this prototype is given.

Section 3 presents the requirements and preparations to be done by software developers and users if they want to make use of the Application Runtime Environment final prototype.

Section 4 states information about the installation and deployment of the provided software package.

Section 5 describes how software developers can use the provided functionalities.

Section 6 discusses the current limitations of the final prototype of the Application Runtime Environment and gives an outlook on the final prototype.

Finally, Section 7 provides a summary of the document.

2 Prototype Scope and Requirements Coverage

2.1 Application Runtime Environment (ARE) – General Information

The Application Runtime Environment component provides the runtime environment for SIMPLI-CITY apps in the Personal Mobility Assistant Android App. It covers the installation and update of apps from the SIMPLI-CITY App Marketplace and executes them on the Android device. A messaging interface enables the apps to communicate with other apps and/or the services running in the Service Runtime Environment. Also the ARE enables apps to access the SIMPLI-CITY Cloud-based Information Infrastructure easily, to store and get needed data. Also, includes methods to access the Sensor Abstraction Interface for the developers.

Figure 1 shows the location of the Application Runtime Environment in the SIMPLI-CITY Global Architecture. As being part of the Personal Mobility Assistant, only this architecture part is shown. For the full Global Architecture, refer to deliverable D3.1.

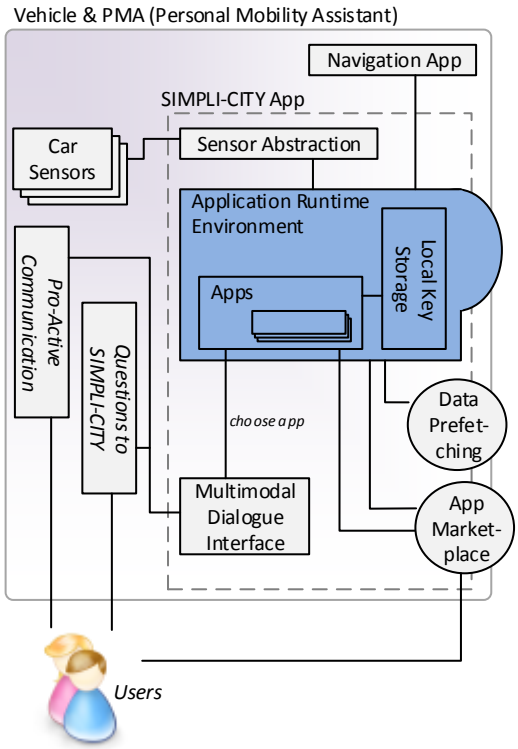

Figure 1: Location of Application Runtime Environment in the SIMPLI-CITY Global Architecture

2.2 Scope of the Final Prototype

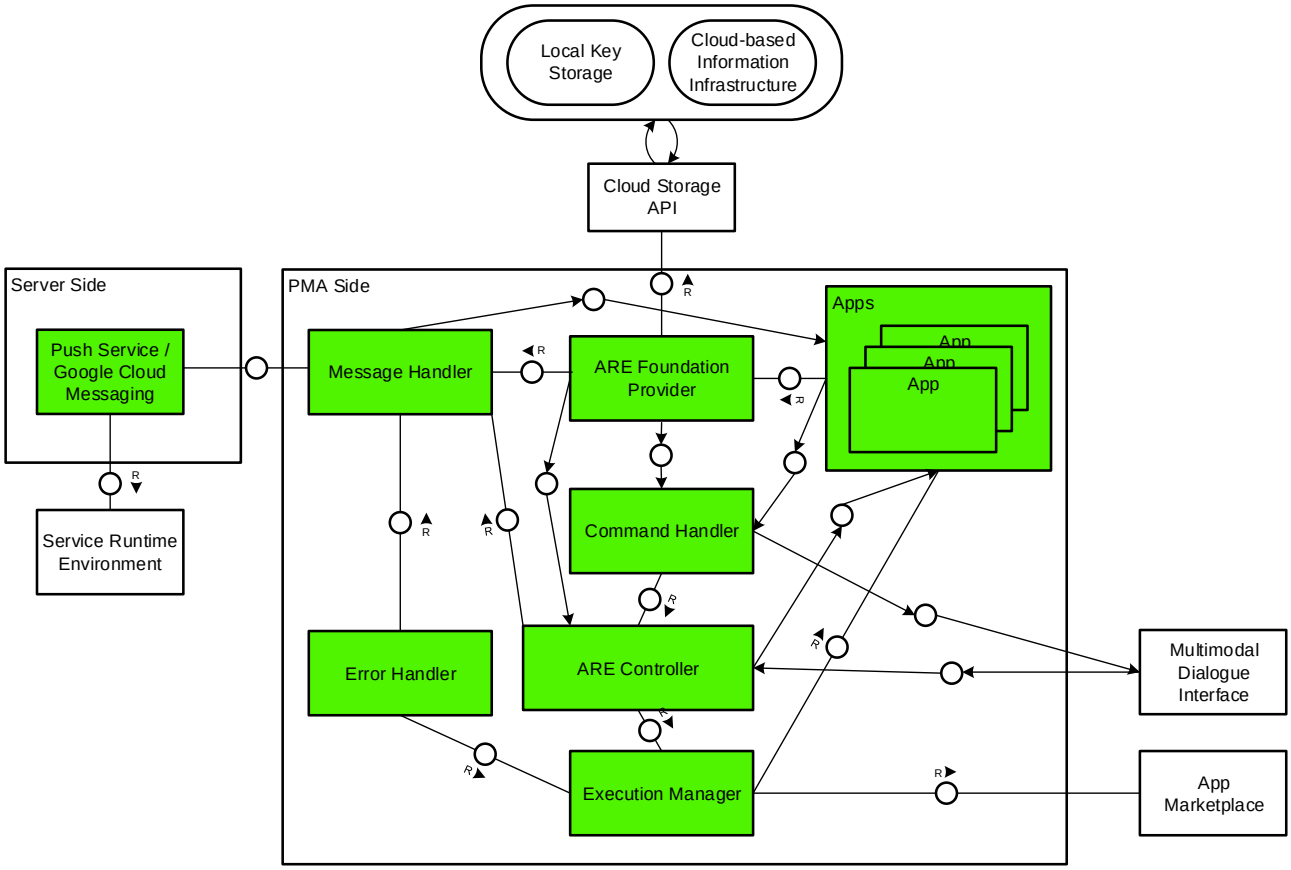

Figure 2: Scope of the Final Prototype of the Application Runtime Environment

Figure 2 depicts the status of development of the final prototype of Application Runtime Environment, showing the subcomponents that are covered within this final prototype. Note that this figure shows the App Marketplace, the Multimodal Dialog Interface, Cloud-based Information Infrastructure, Cloud Storage API, Local Key Storage and the Service Runtime Environment components within the Application Runtime Environment; therefore, also other parts of the SIMPLI-CITY Global Architecture, which are not developed within this component, are depicted.

The status of the implementation is shown using the following colour codes:

- Green: Fully implemented.
- Orange: Partially implemented.
- White: No implementation so far, or not part of this deliverable.

In the following subsections, the scope and status of the single subcomponents (as depicted in Figure 2) will be discussed in more detail. For the Functional Specification and Technical Specification of these subcomponents, refer to SIMPLI-CITY deliverables D3.2.1 and D3.2.2, respectively.

2.2.1 Message Handler

The Message Handler manages push and pull notifications via the Push Service on the Server Side of the Application Runtime Environment and provides a central communication center for apps. This component interacts with the Push Service via a REST Interface. The main functionalities of the Message Handler are the sending and receiving of messages, this means it is the main communication point between apps and services.

2.2.2 ARE Foundation Provider

The ARE Foundation Provider will provide an API with core methods used by apps. It is the main connection point to apps and will also wrap the access to the Message Handler, the Cloud-based Information Infrastructure, the command Handler and the ARE Controller.

These functionalities are exposed via a Java API, since they will only be used on the PMA and Java is the recommended programming language used for the Android operating system.

2.2.3 Error Handler

The Error Handler catches app exceptions and reports them to the App Marketplace via the Execution Manager. The Error Handler is a specific implementation of the Message Handler. Hence, it offers the same functionality but only sends and receives specifically defined messages, which are pre-specified error messages and serve as a wrapper for the native exceptions to make them more user-friendly and have them not crash the environment.

2.2.4 Command Handler

The Command Handler represents the connection to the Multimodal Dialogue Interface (MMDI) and handles the communication with the MMDI that is invoked by the Application Runtime Environment. It also provides the data needed by the MMDI to show the user interface. This includes all interactions that origin at an app. The Command Handler will format these interactions and messages into a format that is understandable by the MMDI.

2.2.5 ARE Controller

The ARE Controller controls the user interaction and inter-app communication facilities. It receives user input from the MMDI and dispatches them to the corresponding apps. Additionally, the ARE Controller initiates the sending of PMA-wide messages as well as messages between different apps on the PMA.

2.2.6 Execution Manager

The Execution Manager launches and handles apps during execution time. It is responsible for checking for app updates on the App Marketplace and will invoke the Local App Manager of the PMA Side of the App Marketplace. Therefore it is in charge of install an update on the current device, since the Execution Manager is responsible for the execution of apps. The Execution Manager also forwards error messages to the App Marketplace as invoked by the Error Handler.

2.2.7 Push Service

This service allows backend services running in the Service Runtime Environment to notify apps in case of events. The Push Service handles the delivery of messages and distributes them to the different PMA instances (e.g., for informing an app installed on many PMA devices). The Push Service can also be used to inform a specific app of a specific user (e.g., to inform the user about upcoming traffic jams on the current route).

2.3 Covered Requirements

This section describes the degree of fulfilment of the requirements to be covered by Application Runtime Environment and specified in the Requirements Analysis Deliverable (D2.3) and the Functional Specification (D3.2.1).

Table 1: Requirements Related to Application Runtime Environment and their Degree of Fulfilment

Requirement	Degree of Fulfilment	Comment
Must Have Requirements		
U103: Fault tolerance U104: Stability	100%	Most of the subcomponents of the Application Runtime Environment were developed with isolation in mind. That way, a crash of a subsystem will not affect other parts. Also, every error will be caught and the system tries to recover itself.
U72: Android support	100%	As the prototype is developed natively for an Android System, it integrates completely into the Android platform and makes use of its features.
U185: Standardized messages between apps	100%	The inter-app communication allows simple Key-Value data transfer like in the Android Platform. This standardize the communication, but leaves some flexibility to the developer what data he wants to transfer to another app.
U186: Registry of installed SIMPLI-CITY apps	100%	In the Application Runtime Environment, the SIMPLI-CITY app registry is included in the Execution Manager subcomponent.

Requirement	Degree of Fulfilment	Comment
U193: Exchange of information from apps to server U194: Exchange of information from server to apps	100%	The Application Runtime Environment implements the technical communication between the Apps and services on the server site. In this way, the developer can focus on the actual development. Technically, the server-app communication is realized by a PUSH mechanism, while the app-server communication is a direct call which can be synchron or asyncron.
Should Have Requirements		
U48: App crashes are minimized U49: A crash should not impact other apps	100%	Every action or query of a SIMPLI-CITY app is executed in a separate thread. So the implementation goes even further than isolate apps. It is even isolating some parts of the app code.
U100: Backwards compatibility of apps	100%	The Application Runtime Environment API is built to be enhanced by adding new functionality, but not to change the current interfaces. This way, all apps will be backward compatible.
Could Have Requirements		
U76: Tablet support	100%	As the prototype is developed for an Android System, it will also work on a tablet, if the same conditions as for a phone are fulfilled. This includes for example the minimum Android version 4.4 – KitKat and Sensors like GPS.

3 Preparations

This section provides information about what potential users (mainly software developers but also final SIMPLI-CITY users) need to prepare in order to use the functionalities of the delivered prototype.

As the Application Runtime Environment consists of a server and a client side, the prototype is split up into two parts. The specific preparations can be found in section 3.1, section 3.2 and section 3.3 respectively.

3.1 Server Side (System Administrators)

The server site of the Application Runtime Environment is written in the GO[1] programming language and so it requires a GO runtime environment for execution. Installation packages for the most common operation systems can be download at:

- https://golang.org/dl/

Detailed installation instructions can be found under:

- https://golang.org/doc/install

In this document, an already existing installation of the GO runtime environment is expected. During the development of this component, it was tested on Ubuntu Version 12.04, 13.10 and 14.04.

The Application Runtime Environment also communicates with the Multimodal Dialog Interface, so some server site preparations may have to take place. For detailed instructions, please refer to D6.2.2.

3.2 Client Side (App Developers)

Application Developers will have to setup the Application Design Studio component on their machines in order to create new SIMPLI-CITY apps and deploy them to the SIMPLI-CITY platform. This will be covered in D6.4 which is also due in project month 30. Please refer to D6.4 for the preparations which have to be done for the Application Design Studio.

3.3 Client Side (Device Users)

This prototype is delivered as an Android APK installation file. As a prototype it will not be available in the Google Play Store. In order to install any Android application from sources other than the Play Store, Android has to be configured to install Apps from “Unknown Sources”. This setting option is found under “Settings > Security > Unknown Sources”.

An easy step-by-step guide to enable it can be found in Figure 3 to Figure 7. After following these steps, user will be able to install any Android application in the form of an APK, either downloaded from Web or copied to the phone’s memory (i.e., SD card or internal memory, via USB cable, Bluetooth, etc.), including the ones that are part of this deliverable. As reference device for SIMPLI-CITY, the Google Nexus 4 was chosen. For this device the Android Version 4.4 (KitKat) is available, so the installation will be explained with this version.

[1] https://golang.org/

Figure 3: Opening the Notification Tray with Swipe Down Gesture

Figure 4: Change to the Quick Settings Panel

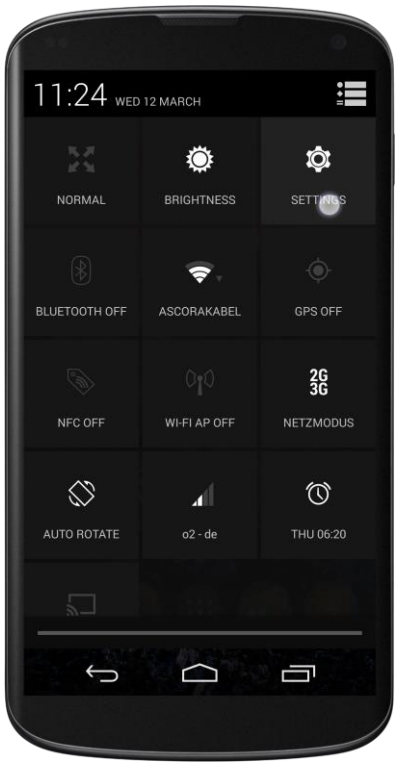

Figure 5: Tap “Settings” to Open the Android Settings

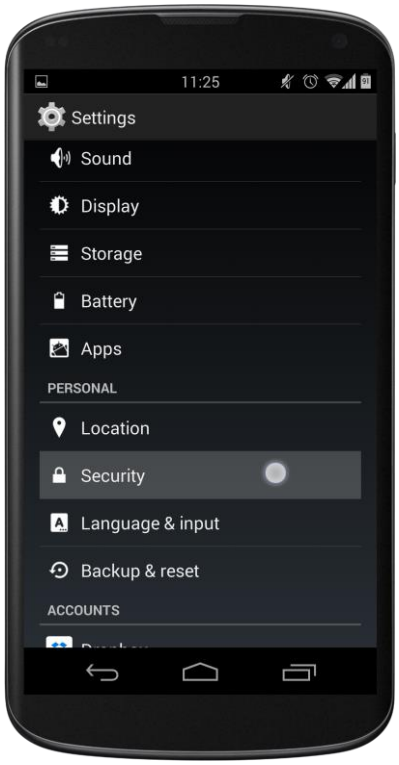

Figure 6: Scroll Down to “Security” and Tap it to Open

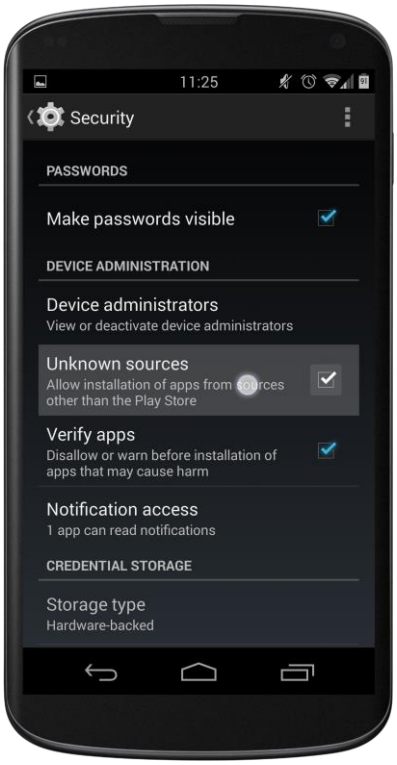

Figure 7: Scroll Down to “Unknown Sources” and Activate it with a Tap

If this setting is not set, Android will give a notification before the Installation of the APK, giving the possibility to jump directly to the security settings.

4 Installation (Deployment)

This section provides guidelines on how to install and deploy the final prototypes of the Application Runtime Environment.

As the Application Runtime Environment is very close connected to the Dialog Interface and the Multimodal Dialog Interface, all three artefacts are already bundled together and will be installed within the same APK installation file.

4.1 Server Side (System Administrators)

The prototype is delivered alongside a demo webserver, which is part of the archive provided with this deliverable. In this section, the setup on a Linux based system is shown. For Windows, some commands have to be adopted but the command sequences will not change,

Before the server is started, the $GOPATH environment variable needs to be set, this includes two steps:

```
cd <projectpath>/src
export GOPATH=`pwd`
```

Afterwards the missing go packages need to be installed, this is automatically done by the golang toolset:

```
go get
```

After installing all the missing packages the prototype is ready to run. The compilation and initiation of the prototype is done via a simple command:

```
go run main.go
```

Now the webserver is running on port 3000 of the local machine, so it is accessible via a web browser. The url for the server will be “http://localhost:3000”.

4.2 Client Side (App Developers)

As already mentioned in section 3.2, in this deliverable, the app developer is expected to use the Application Design Studio for writing SIMPLI-CITY apps. Please refer to D6.4 for the respective installation instructions.

4.3 Client Side (Device Users)

The Application Runtime Environment component is included into the Personal Mobility Assistant Android app in the form of a single Android app, contained in an APK archive: *MMDI-Frontend-release.apk*. This archive comprises some directories and files that are part of the usual structure of any Android application:

- META-INF/: Contains the manifest file, the certificate of the application and SHA-1 digest for every resource included on the APK.
- res/: Contains not compiled resources (e.g., images, sounds…).
- AndroidManifest.xml: Additional Android manifest file (in Android binary XML format).
- classes.dex: Classes compiled in dex file format.

- resources.arsc: Precompiled resources.

In order to install the Personal Mobility Assistant app, it's necessary to extract the APK archive from the *D6.3.2_Personal_Mobility_Assistant_App.zip*. Once it is extracted somewhere in the computer file system (e.g., C:\Users\Some.User\Documents), it has to be transferred to the Android device's storage, for example, using any of these following options:

- Connecting the device to the computer by USB cable and selecting *USB storage* or *Media device (MTP)* when notification shows up (see Figure 8), so phone contents can be accessed through the computer. And then, copying the APK archive to the Download folder in the phone or any other folder accessible from the File Manager.
- Transferring the APK archive using a Bluetooth connection between computer and device.
- Sending the APK archive to the phone through any cross-platform service installed on the phone, such as Dropbox, AirDroid or Email (i.e., attaching the APK archive to an e-mail).
- Installing the APK onto the device by using the following command from the Android Software Development Kit.

```
adb install <APK filename>
```

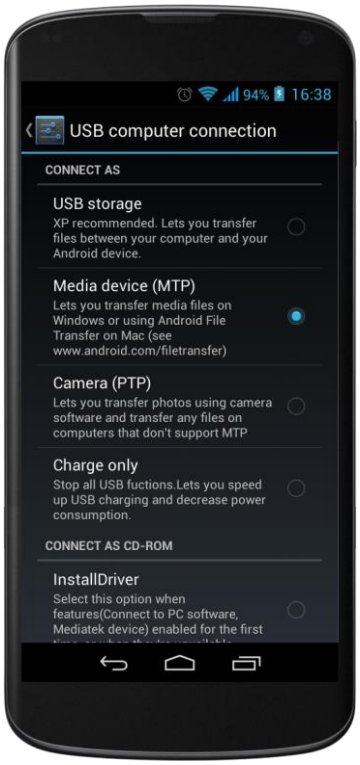

Figure 8: Available Options on Connecting an Android Mobile Phone to a Computer

Once the APK archive is transferred to the phone, the Personal Mobility Assistant Android app can be installed just by tapping on it and selecting *Install* (see Figure 9). For this invocation of the installation, any Android File Manager can be used.

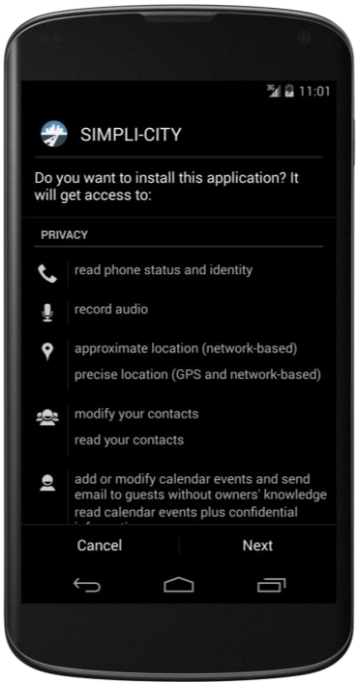

Figure 9: Package Installer Screen where all the Permissions Needed are Shown

If the installation process is successful, a launcher named *SIMPLI-CITY* with the corporative logo of SIMPLI-CITY will appear on the list of installed apps (shown in Figure 10).

Figure 10: Personal Mobility Assistant App Launcher after a Successful Installation

5 Execution and Usage of the Software

5.1 Server Side (System Administrators)

The server side functionality of the Application Runtime Environment is offered by a REST interface service, which will be presented in Table 2. It should be noted that just the needed interface method will be explained. Other internal methods must not be used by developers, and so they are not part of this documentation.

Table 2: Push Service Send Message REST Interface Description

Method	POST	**URL**	$API_ROOT/sendMessage				
Description	Sends message to all apps subscribed to the service id						
JSON Object	http://simpli-city.eu/ApplicationRuntimeEnvironment/PushService/Message						
JSON Attribute	sender	**Required**	yes	**Possible Values**	any UUID	**Description**	The sender of the message
JSON Attribute	message	**Required**	yes	**Possible Values**	any JSON object as message	**Description**	The message itself
Example URL	$API_ROOT/sendMessage						
Response	Updated HTTP status code						
HTTP Status Code		**Required**	yes	**Possible Values**	200 400 502	**Description**	200: Message sent. 400: Bad request: Invalid parameter. 502: Message could not be sent.
Example Response	HTTP/1.1 200 OK Content-Length: 0 Connection: Close						

Listing 1: JSON Example – Send Message

```
{
   "sender": "a2ae6cc9a7acfff494422585a43459c2",
   "message": {
      "operation": "data_updated",
      "data": [ ... ]
   }
}
```

The *sendMessage* (see Table 2) interface enables other SIMPLI-CITY components to send a PUSH message to the PMA. The system looks up which apps are subscribed to the particular service (given by the sender attribute) and forwards the message to them.

5.2 Client Side (App Developers)

The app development in SIMPLI-CITY is intended to be done with the Application Design Studio, developed in T6.4. The preparation and installation is explained in section 3.2 and section 4.2 of this deliverable. Due to this, it is expected by the developer to use the Application Design Studio, as it takes care of most of the SIMPLI-CITY specific aspects. The development of Apps for SIMPLI-CITY consists of two parts:

1. App code in Java (Section 5.2.1)
2. XML and Python based definition of the Dialog Interface including the Speech Navigation (Section 5.2.2)

5.2.1 App Code Development

The app code base of SIMPLI-CITY Apps is using pure Java and the developer is expected to follow the common Java Code Style and Syntax guidelines with one exception. In the Application Runtime Environment the app actions and app queries are not simple methods of a Java class. Every action or query has to be a class which inherits from the abstract *AppAction* or *AppQuery* class. This offers the great possibility to save states across different action or query invocations without modifying states saved in the actual App. It also brings the isolation of methods in an app even further. In that way, the stability of the app and the whole PMA can be increased without a big programming language paradigms shift. To avoid time and resource costly reflection, every action and query has to be registered for the App. A best practice is to do it in the *init* method (see Listing 2) by the *addAction* (see Listing 13) or *addQuery* (see Listing 14) method.

For easy use, each SIMPLI-CITY app has to extend the abstract class *App*. In this class, the following methods (Listing 3 to Listing 33) are already implemented and provide the access from the app to the Application Runtime Environment and so to the whole SIMPLI-CITY platform. In the developers guide included in the Application Design Studio, the developer can also find these interface definitions, along with more rich examples, tutorials, guidelines and best practices.

5.2.1.1 General App Methods

Listing 2: Method Interface Example – App Initialization Method

```
/**
 * For initialization of the SIMPLI-CITY App.
 * Any constructor should be avoided (especially with parameters).
 * Only in init() it can be assured that all the PMA system is available
 * (e.g., Cloud Storage).
 */
public abstract void init();
```

Listing 3: Method Interface Example – Show Notification

```
/**
 * Send an Android Notification which redirects to the PMA
 * @param message to be displayed in the Notification
 */
final protected void sendNotification(String message);
```

Listing 4: Method Interface Example – Log Message

```
/**
 * Log a message for debugging purpose
 * @param message to be logged
 */
final protected void log(String message);
```

5.2.1.2 App Manifest Information Methods

Listing 5: Method Interface Example – Get App Id

```
/**
 * ID of the app, usually the full qualified name of the concrete app class
 * (e.g., "com.example.MyApp").
 * @return the ID of the App.
 */
final public String getAppId();
```

Listing 6: Method Interface Example – Get Author Id

```
/**
 * ID of the developer that created the app as found in the App Marketplace.
 * @return the ID of the app author.
 */
final public String getAuthorId();
```

Listing 7: Method Interface Example – Get Keywords

```
/**
 * List of keywords of the app (e.g., "safety", "race", "eco").
 * @return a list of all the app keywords
 */
final public List<String> getKeywords();
```

Listing 8: Method Interface Example – Get Description

```
/**
 * Full description of the App. Some html tags may be used as <em> or <strong>.
 * @return the description of the App.
 */
final public String getDescription();
```

Listing 9: Method Interface Example – Get App Name

```
/**
 * Complete name of the app. Spaces and some other characters may be used.
 * @return the app name
 */
final public String getName();
```

Listing 10: Method Interface Example – Get Version

```
/**
 * Version of the app (e.g., "1.0", "2.3b", "0.5-preAlpha").
 * @return the app Version
 */
final public String getVersion();
```

5.2.1.3 App Resource Methods

Listing 11: Method Interface Example – Get All Resources

```
/**
 * List all resources bundled with the app, usually media as images.
 * @return a list of all app resources.
 */
final public List<AppResource> getResources();
```

Listing 12: Method Interface Example – Get Specific Resource

```
/**
 * Get a resource by its ID.
 * @param resourceId of the resource.
 * @return the requested AppResource.
 */
final public AppResource getResource(int resourceId);
```

5.2.1.4 App Action Methods

Listing 13: Method Interface Example – Register an App Action

```
/**
 * Register an Action so it is available for the MMDI.
 * @param actionName as the name of the app action when invoked by the TDM backend.
 * @param appAction as the concrete app action object to invoke.
 */
final protected void addAction(String actionName, AppAction appAction);
```

Listing 14: Method Interface Example – Register an App Query

```
/**
 * Register a Query so it is available for the MMDI.
 * @param queryName as the name of the app query when invoked by the TDM backend
 * @param appQuery as the concrete app query object to invoke
 */
final protected void addQuery(String queryName, AppQuery appQuery);
```

Listing 15: Method Interface Example – Invoke an App Action

```
/**
 * Invoke an Action of the App, passing a set of parameters and getting
 * an Action Result that depends on how the AppAction has processed it.
 * It is one of the two main ways to communicate from the PMA to an App.
 * @param actionName of the concrete app action to invoke
 * @param actionParameters as input to the concrete app action
 * @return the result of the app action
 */
public final AppActionResult invokeAction(String actionName, Map<String, String>
actionParameters);
```

Listing 16: Method Interface Example – Invoke an App Query

```
/**
 * Invoke a Query of the App, passing a set of parameters and getting
 * an Query Result that depends on how the AppQuery has processed it.
 * It is one of the two main ways to communicate from the PMA to an App.
 * @param queryName of the concrete app query to invoke
 * @param queryParameters as input to the concrete App-Query
 * @return the result of the AppQuery
 */
public final AppQueryResult invokeQuery(String queryName, Map<String, String>
queryParameters);
```

Listing 17: Method Interface Example – Publish Action Started

```
/**
 * Notify to all the registered Command Callbacks that an Action has started.
 * @param actionName Name of the action as registered in addAction(name, action).
 */
final protected void notifyActionStarted(String actionName);
```

Listing 18: Method Interface Example – Publish Action Ended

```
/**
 * Notify to all the registered Command Callbacks that an Action has ended.
 * @param actionName Name of the action as registered in addAction(name, action).
 * @param parameter set of parameters that call backs may process.
 */
final protected void notifyActionEnded(String actionName, HashMap<String, String>
parameter);
```

5.2.1.5 Service Communication Methods

Listing 19: Method Interface Example – Get Data from Service

```
/**
 * Get data from a service that runs on the Service Runtime Environment.
 * Uses a ServiceInvocationMessage, which contains the UUID of the Service,
 * the service method to call and an array of parameters.
 * Also, an optional call-back may be provided to run the request asyncronisly.
 * @return the result of the remote invocation of the method on the Service
 * when running the request syncronisly.
 */
final protected String getDataFromService(final ServiceInvocationMessage msg);
```

Listing 20: Method Interface Example – Subscribe to Service

```
/**
 * Subscribe to a message from a service that runs on the Service Runtime
 * Environment.
 * Uses a ServiceSubscriptionMessage, which contains the UUID of the Service,
 * an upper and lower threshold, the interval of invocation, and the starting
 * and ending of the subscription. Also, a ServiceInvocationCallback
 * call-back needs to be provided to run when subscribed message is triggered
 * under the restrictions of threshold and start/end.
 * @param subscriptionMessage the message with all the needed information to
 *        invoke Service method.
 * @param callback a call back to invoke when the message subscription is triggered.
 */
final protected void subscribeToService(ServiceSubscriptionMessage
subscriptionMessage, ServiceInvocationCallback callback);
```

Listing 21: Method Interface Example – Unsubscribe from Service

```
/**
 * Unsubscribes from a Service.
 * @param serviceId ID of Service as in Service Runtime Environment.
 */
final protected void unsubscribeFromService(String serviceId);
```

5.2.1.6 Sensor Abstraction Methods

Listing 22: Method Interface Example – Store File

```
/**
 * Get the LSA Connector, which allows querying devices and sensors.
 * @return the LSA Connector.
 */
final protected ILsaConnector getLsaConnector();
```

5.2.1.7 Cloud Storage Methods

Listing 23: Method Interface Example – Store File

```
/**
 * Stores a file in a bucket on the Cloud Storage.
 * @param bucketId The ID of the bucket where the file should be stored
 * @param file The data of the file that needs to be stored
 * @return The Cloud Storage API returns an identifier for the file in the
 * specific bucket, which is used for further operations concerning the stored file.
 */
final protected String storeFile(String bucketId, CloudStorageFile file);
```

Listing 24: Method Interface Example – Retrieve File

```
/**
 * Retrieves a file from the Cloud Storage.
 * @param bucketId The ID of the bucket where the file should be stored
 * @param fileId The data of the file that needs to be stored
 * @return The Cloud Storage API returns an identifier for the file in the
 * specific bucket, which is used for further operations concerning the stored file.
 */
final protected CloudStorageFile retrieveFile(String bucketId, String fileId);
```

Listing 25: Method Interface Example – Update File

```
/**
 * Updates the specified file in the Cloud Storage.
 * @param bucketId The ID of the bucket where the file should be stored
 * @param fileId The data of the file that needs to be stored
 * @param file The data of the file that needs to be stored
 * @return The Cloud Storage API returns an identifier for the file in the
 * specific bucket, which is used for further operations concerning the stored file.
 */
final protected boolean updateFile(String bucketId, String fileId, CloudStorageFile
file);
```

Listing 26: Method Interface Example – Delete File

```
/**
 * Deletes a file in the Cloud Storage.
 * @param bucketId The ID of the bucket where the file should be stored
 * @param fileId The data of the file that needs to be stored
 * @return The Cloud Storage API returns an identifier for the file in the specific
bucket, which is used for further operations concerning the stored file.
 */
final protected boolean deleteFile(String bucketId, String fileId);
```

Listing 27: Method Interface Example – Store Semi Structured Data

```
/**
 * Stores a JSON structure in the Cloud Storage.
 * @param bucketId The ID of the bucket where the file should be stored
 * @param data The data of the file that needs to be stored
 * @return The Cloud Storage API returns an identifier for the file in the
 * specific bucket, which is used for further operations concerning the stored file.
 */
final protected String storeJSON(String bucketId, CloudStorageJSON data);
```

Listing 28: Method Interface Example – Retrieve Semi Structured Data

```
/**
 * Retrieves a JSON structure from the Cloud Storage.
 * @param bucketId The ID of the bucket where the file should be stored
 * @param structureId The data of the file that needs to be stored
 * @return The Cloud Storage API returns an identifier for the file in the
 * specific bucket, which is used for further operations concerning the stored file.
 */
final protected CloudStorageJSON retrieveJSON(String bucketId, String structureId);
```

Listing 29: Method Interface Example – Update Semi Structured Data

```
/**
 * Updates the specified JSON structure in the Cloud Storage.
 * @param bucketId The ID of the bucket where the file should be stored
 * @param structureId The data of the file that needs to be stored
 * @param structure The data of the file that needs to be stored
 * @return The Cloud Storage API returns an identifier for the file in the
 * specific bucket, which is used for further operations concerning the stored file.
 */
final protected boolean updateJSON(String bucketId, String structureId,
CloudStorageJSON structure);
```

Listing 30: Method Interface Example – Delete Semi Structured Data

```
/**
 * Deletes the specified JSON structure in the Cloud Storage.
 * @param structureId the id of the saved structure
 * @param bucketId the id of the bucket the structure is saved in
 * @return true if the structure was successfully removed
 */
final protected boolean deleteJSON(String structureId, String bucketId);
```

5.2.1.8 Local Key Storage Methods

Listing 31: Method Interface Example – Store Key-Value Pair

```
/**
  * A string will be stored under the specific key. If the key already
  * exists the old value will be overridden
  * @param key, the key under which the value will be saved
  * @param value, the value to save under the key
  * @return the old value that was stored under the key, or null
  */
final protected String storeLocalKeyValue(String key, String value);
```

Listing 32: Method Interface Example – Get Key-Value Pair

```
/**
  * A string will be retrieved, which was stored under the specific key.
  * If no entry was found under the key, null will be returned.
  * @param key the key under which the string is stored to be retrieved
  * @return the value that was stored under the key, or null
  */
final protected String getLocalKeyValue(String key);
```

Listing 33: Method Interface Example – Delete Key-Value Pair

```
/**
 * Removes the value stored under the specific key.
 * @param key the key under which the data should be deleted.
 * @return the value that was stored under the key, or null
 */
final protected String deleteLocalKeyValue(String key);
```

5.2.2 Dialog Interface and Speech Navigation Development

A dialogue app resides in the Talkamatic Dialogue Manager (TDM) backend and can be divided into three modules, grammar, ontology and domain. The frontend holds a component that enables the developer to access external resources such as databases.

The ontology defines concepts, entities and actions that the user and the system may reference in questions, answers and requests. A grammar contains the language model that is used by the system and it defines how the system utters questions and speaks about actions. The grammar also defines how the user may speak with the system.

The domain contains plans for the application. A plan describes the actions that need to be executed or questions that need to be answered. It also describes what information is needed to carry out the action or answer the question. By adding plan items, the system is able to query external resources to answer a question or perform actions on the Android device such as playing music.

Listing 34: Plan Example – Ask the Device for Information

```
<goal type="resolve" question_type="wh_question"
      predicate="current_trip_information">
   <plan>
      <dev_query device="EcoAssistantDevice"
                 type="wh_question"
                 predicate="current_trip_information"/>
   </plan>
</goal>
```

To enable the developer to create and run app specific code, a component called *Device* exists, which is divided into two sub components, backend device and frontend device. The backend device defines what actions and queries the frontend device can handle. A frontend implements the actions and queries specified in the backend device. The implementation can access a remote resource such as a database or getting information from the car.

Listing 35: Example Code – Backend Device forwarding a Query to the Frontend

```
class EcoAssistantDevice:
   def __init__(self):
      self.reset()

   def reset(self):
      self._in_moving_vehicle = False

   class current_trip_information(DeviceWHQuery):
      @send_to_frontend_device
      def perform(self):
         pass
```

There are two types of requests that can be handled by the frontend device, queries (AppQuery) and actions (AppAction). A query returns a list of options that the user can choose (as a list of strings) and an action returns a boolean value, to indicate if the action was successfully executed or not.

Listing 36: Example Code - Frontend Handling a Query

```
public class current_trip_information extends AppQuery{
  @Override
  public AppQueryResult invoke(Map<String, String> invocationParameters) {
    ArrayList<String> result = new ArrayList<String>();
    result.add("Current trip information");

    AppQueryResult queryResult = new AppQueryResult();
    queryResult.setResult(result);
    return queryResult;
  }
}
```

Listing 37: Example Code - Frontend Handling an Action

```
public class MyHistory extends AppAction {
  @Override
  public AppActionResult invoke(Map<String, String> invocationParameters) {
    AppActionResult actionResult = new AppActionResult();
    actionResult.setResult(true);
    return actionResult;
  }
}
```

When the user asks the system something that will start a plan, the system will find the appropriate plan and execute it. If the first plan item is an action or a query, the system will forward it automatically to the frontend and wait for a return value.

5.3 Client Side (Device Users)

The user will just use the Application Runtime Environment in an indirect way by using the Personal Mobility Assistant Android App. The ARE is intended to work in the background and handle all running SIMPLI-CITY Apps and takes care of the installation and update of SIMPLI-CITY Apps. To mark a SIMPLI-CITY app for installation or update, the user has to use the SIMPLI-CITY Application Marketplace which is in charge of managing this information. To use the Personal Mobility Assistant and so the Application Runtime

Environment, the user has to start the SIMPLI-CITY app on Android. See section 4.3 for more information about the installation and app start procedure.

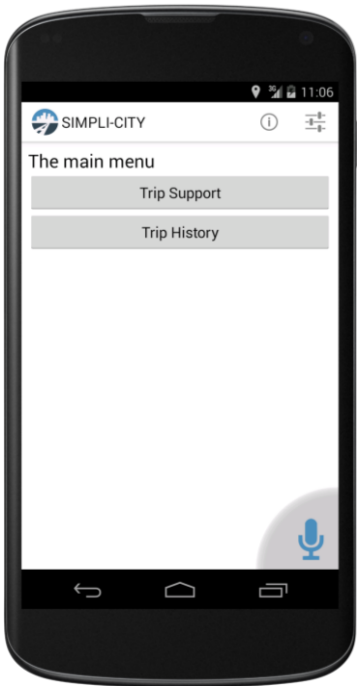

Figure 11: Main Screen of the Personal Mobility Assistant with an Example App

6 Limitations and Further Developments

This section provides a brief overview of what has been provided in the final prototype, with a focus on limitations and further developments envisioned.

6.1 Limitations

With this final prototype the requested functionality of the Application Runtime Environment is implemented. It includes the whole integration of the Sensor Abstraction Interfaces, the interface for easy access to the Cloud-based Information Infrastructure and the Prefetch Proxy of the Media Data and Prefetching component of SIMPLI-CITY. As by design this component is also closely integrated with the Multimodal Dialog Interface as the user interface to the SIMPLI-CITY user. Due to the prototype (beta) status of this deliverable, it is not intended to be used in a productive environment but it is functional complete and meet all requested requirements. With this prototype the work is concluded for this task without limitations.

6.2 Further Development

With this final prototype, the work on the Application Runtime Environment will be mainly finished and T6.3 – Application Runtime Environment will end. If required, upcoming issues with the ARE will be solved during the use case implementations of WP7 and WP8. This work will not aim at introducing new features but rather concentrate on fixing bugs and minor enhancements.

7 Summary

This deliverable present and describes the scope of the prototype of the Application Runtime Environment component of the SIMPLI-CITY Personal Mobility Assistant. Within this prototype, the development of the subcomponents has been finished. It is the basis for the Personal Mobility Assistant and connects the components Multimodal Dialog Interface, Application Marketplace and provides the runtime for all SIMPLI-CITY Apps.

With this prototype, the Personal Mobility Assistant is able to run multiple SIMPLI-CITY apps in a central Android application. This unifies the connection to the user interface and adds the support for a multimodal dialog interface. This homogenous interface for app developers also eases the implementation of custom SIMPLI-CITY Apps and increases the stability of the Personal Mobility Assistant.

Following on, an analysis of the requirements to be covered by the component showed that a complete fulfilment has been achieved. Including all the “must-have”, “should-have” and “could-have” requirements, specified by the Requirements Analysis Report (D2.3).

Moreover, all the required steps to install, deploy, execute and use the Application Runtime Environment component are detailed.

Finally, the limitations of the current prototype are specified, describing the maybe upcoming issues while this prototype is used in WP7 and WP8.