# WP3 – Architecture, Functional & Technical Specification, Security & Privacy Concept, Integration

# D3.1: Global Architecture Definition

Deliverable Lead: ASC

Contributing Partners: ASC, TIE, TUV, TUDA, IBM, TALK, TEMP

Delivery Date: 06/2013

Dissemination Level: Public

Version 1.10

This document defines the main components and their interaction. It is based on the general component structure outlined in the Description of Work and it takes into account the outcomes of deliverable D2.1. This deliverable provides a higher level of detail and it also involves a more detailed specification of the component interaction processes.

| Document Status | |
|---|---|
| **Deliverable Lead** | Dr. Sven Abels, Ascora GmbH |
| **Internal Reviewer 1** | Enrica Deregibus, CRF |
| **Internal Reviewer 2** | Giuseppe Liguori, SRM |
| **Type** | Deliverable |
| **Work Package** | WP3 – Project Management, Quality Assurance and Reporting |
| **ID** | D3.1: Global Architecture Definition |
| **Due Date** | 30.04.2013 |
| **Delivery Date** | 04.06.2013 |
| **Status** | Approved |

| Document History | |
|---|---|
| **Draft Version** | First draft circulated on 27.02.2013 by partner ASC |
| **Contributions** | V0.1, ASC, 25.02.2013: First internal version<br>V0.2, ASC, 05.03.2013: Improved internal version<br>V0.3, ASC, 12.03.2013: Architecture<br>V0.4, ASC, 22.03.2013: Example Chapter I<br>V0.5, ASC, 26.03.2013: Example Chapter II<br>V0.51, ASC, 28.03.2013: Template corrections, various updates<br>V0.6, ALL, 09.04.2013: Integration of partner contributions<br>V0.7, ALL, 06.05.2013: Pre-Version for internal Review<br>V0.8, ALL, 18.05.2013: Version for internal Review<br>V1.0, ASC, 04.06.2013 |
| **Final Version** | V1.10, TUV, 13.01.2014 (Approved 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

Tempos 21, Spain

Centro Ricerche FIAT, Italy

SRM – Reti e Mobilità, Italy

# Executive Summary

Within this document, the architecture of SIMPLI-CITY is specified. The deliverable starts with a recapitulation of the outcomes from the SIMPLI-CITY vision (D2.1) and the requirements definition (D2.3) on a very high-level view by highlighting how the outcomes of those documents impact the architecture of SIMPLI-CITY.

Afterwards, the document delivers a bird's-eye view of the SIMPLI-CITY components by showing the high-level architecture in a graphical way. Based on this, sections 4 and onwards describe each component in a detailed way. For each component, the following aspects are described:

- The component definition with a clear description of the purpose and scope.
- The Interaction of the component with other components, highlighting how this interaction will be achieved and also what type of information is exchanged.
- The technical foundations describing technical core decisions.

It should be noted that the technical foundations do not aim in providing a detailed technology selection as this will be performed within the technical specification document and as this selection highly depends on the functional specification. Instead, the technical foundations will describe those technical choices that are inherent based on the architecture, the requirements and the vision. For example, the technical foundation may describe that JSON will be used as a way to exchange messages but it will not select a library nor define the data format as those will be performed during deliverable D3.2.1 and D3.2.2.

SIMPLI-CITY consists of a wide range of components. Those components may be split into the following subgroups and are therefore grouped within the corresponding sections of this deliverable: (i) Server-Side Components, (ii) PMA Components, (iii) Developer Components and (iv) Security and Privacy Aspects.

Finally, the document ends with summarizing risks which might be occurring in the further pursuit of the project and details what should be addressed to prevent problems.

# 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 deliverable, the overall project vision in terms of its general positioning, the project’s business and research/technological objectives will be revealed. Further, the project stakeholders, the underlying vision enablers, and some preliminary usage scenarios will be presented.

## 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 applications 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 realize and sell their mobility-related service and app ideas by a range of methods and tools, including the Mobility Services and Application 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 Service 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, Semantic Web, Internet of Things, Internet of Services, and Human-Computer Interaction. For more information, please refer to the project website at www.simpli-city.eu.

## 1.2 Deliverable Purpose, Scope and Context

Within this document, the architecture of SIMPLI-CITY is specified. The deliverable starts with a recapitulation of the outcomes from the SIMPLI-CITY vision (D2.1) and the requirements definition (D2.3) on a very high-level view by highlighting how the outcomes of those documents impact the architecture of SIMPLI-CITY.

Afterwards, the document delivers a bird's-eye view of the SIMPLI-CITY components by showing the high-level architecture in a graphical way. Based on this, sections 4 and onwards describe each component in a detailed way. For each component, the following aspects are described:

- The component definition with a clear description of the purpose and scope.
- The Interaction of the Component with other components, highlighting how this interaction will be achieved and also what type of information is exchanged.
- The technical foundations describing technical core decisions.

It should be noted that the technical foundations do not aim in providing a detailed technology selection as this will be performed within the technical specification document and as this selection highly depends on the functional specification. Instead, the technical foundations will describe those technical choices that are inherent based on the architecture, the requirements and the vision. For example, the technical foundation may describe that JSON will be used as a way to exchange messages but it will not select a library nor define the data format as those will be performed during deliverable D3.2.1 and D3.2.2

SIMPLI-CITY consists of a wide range of components. Those components may be split into the following subgroups and are therefore grouped within the corresponding sections of this deliverable:

- *Server-Side Components*: Those components will provide all server-side functionality of SIMPLI-CITY including the service runtime environment and the cloud storage.
- *PMA Components*: Those components will provide all client-side functionality running on the mobile device such as the voice driven interface or the application runtime environment.
- *Developer Components*: Those components will assist developers for creating new services and apps for SIMPLI-CITY and will also provide a studio to prepare the submission and publication of those services and apps.
- *Security and Privacy Aspects*: Those aspects are not a closed component as such. Instead, security and privacy are important in all other components and will therefore have to be considered during the functional and technical specification of each component. Nevertheless, a set of functionalities considering security and privacy will be delivered by SIMPLI-CITY in form of a library which may be used by other components. This includes tasks such as encryption or secure data transmission.

Finally, the document ends with summarizing risks which might be occurring in the further pursuit of the project and details what should be addressed to prevent problems.

## 1.3 Document Status and Target Audience

This document is listed in the Description of Work (DoW) as “public” since it provides general information about the goals and scope of SIMPLI-CITY and can therefore be used by external parties in order to get according insight into the project activities.

While the document primarily aims at the project partners, this public deliverable can also be useful for the wider scientific and industrial community. This includes other publicly funded projects, which may be interested in collaboration activities.

## 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 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 positions the architecture in the context of the former deliverables D2.1 and D2.3. More precisely, it explains the impact that those documents have on the architecture.
- Section 3 characterises the architecture of SIMPLI-CITY from a high-level perspective and shows a graphical bird’s-eye view of it.
- Section 4 describes the server-side components of SIMPLI-CITY.
- Section 5 describes the client-side components (PMA) of SIMPLI-CITY.
- Section 6 describes the developer components of SIMPLI-CITY.
- Section 7 describes the general security and privacy aspects of the project and acts as input for deliverable D3.3.
- Section 8 provides an overview about potential risks and limitations of the architecture and the choices made in this document.
- Finally, section 9 concludes the document.

# 2 Impact of Vision and Requirements

To set this document into context, this section revisits the Vision Consensus Document (D2.1, see section 2.1) and the Requirements Analysis Document (D2.3, see section 2.2). It presents which changes from the initial Description of Work (DOW) have to be implemented in the Global Architecture Definition.

## 2.1 Vision

As can be observed from Figure 1, the SIMPLI-CITY system is comprised of various components, which exhibit a substantial amount of bi- and multilateral interaction. Specifically, the following major groups of components are identified:

- Mobility-related Data as a Service, which deals with collection, aggregation, consolidation, storage, and retrieval of data. A key component is the Cloud-based Information Infrastructure, which serves as highly scalable and flexible database. The corresponding components are developed in WP4.
- Mobility Services Framework, which facilitates development, publication, and execution of services and mobile apps. It targets both service consumers and developers and also provides intermediation between these two stakeholders. The constituent software components are implemented in WP5.
- Personal Mobility Assistant, which is the primary interface to SIMPLI-CITY for the end user, but also encompasses a mobile runtime environment. This group of components is targeted in WP6.

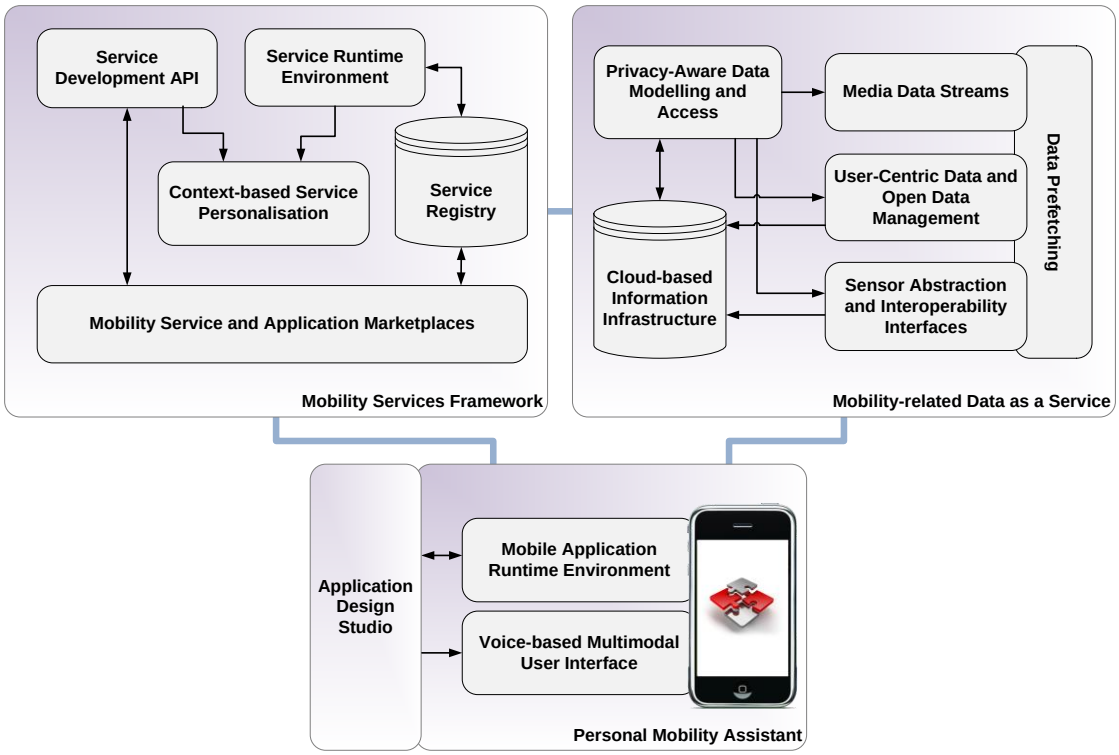


Figure 1: SIMPLI-CITY components

Privacy-Aware Data Modelling and Access Framework provides the means to represent heterogeneous mobility-related data in a ready-to-be-integrated framework (e.g., through spatial, temporal and dynamic representation – mainly for chronological tracking), following semantic representation standards, all ensuring privacy protection of the most sensitive data and easy access to data following principles of the Cloud-based Information Infrastructure. It aims at providing a semantic reference data model, primarily for spatial, temporal, and behavioural-based representation of data.

Cloud-based Information Infrastructure provides means for access, storage, and retrieval of mobility-related data performed within the Cloud-based Information Infrastructure. The component will act as a data storage solution for information and serves as a source for static data that can be used by apps and services.

Sensor Abstraction and Interoperability Interfaces provides the means to integrate real-time data coming from individual sensors, sensor networks, smart objects, telematics, and cooperative systems, using various wired and wireless communication technologies for the usage in SIMPLI-CITY Mobility-related Data as a Service (MDaaS).

User-Centric Data and Open Data Management are used in personalizing city and mobility information for end users. This is done by coupling and correlating information from (i) sensors, (ii) personal user data, and (iii) open and public information. The correlation will be mainly achieved based on spatial and temporal representations as well as user profile level.

Media Data Streams and Data Prefetching, in addition to integrating media streams, increases availability of media streams and other data sources by introducing relevant data prefetching techniques.

Service Development API provides functionalities to easily define, design, develop, and orchestrate mobility-related services, and functionalities to store data within the Cloud-based Information Infrastructure. The API also contains the functionality to combine services in order to provide value-added service compositions. Context-based Service Personalisation provides the means to alter the outcome (output) of a service invocation based on relevant (user) context data.

Service Registry and Service Runtime Environment provide functionalities necessary to register, execute, bind, and monitor backend services. Mobility Service and Application Marketplaces are two marketplaces for services and apps which are usable with SIMPLI-CITY. The voice-based Multimodal User Interface provides a generic user interface to a wide variety of apps. The user interface allows the kind of low-distraction – high-efficiency interaction that is required in an in-vehicle environment.

Mobile Application Runtime Environment provides an API that is usable by (third party) app developers to access typical functions of SIMPLI-CITY such as the communication with service-side systems. The app runtime environment will also manage a list of app commands derived from the app manifest of the design studio.

Application Design Studio provides assistance to app developers within an integrated toolset (Studio) for describing applications in form of a manifest file and preparing them for the deployment within the Application Marketplace.

## 2.2 Requirements

Deliverable D2.3 defines the requirements of the project. These requirements can be seen as the basis for the functional specification D3.2 and also present a direct input to D3.3, but they bear few implications for the Global Architecture Definition, and those are in an expected scope. The list of requirements for a SIMPLI-CITY implementation is presented in D2.3 both in the form of User Interface-related mock-ups and their underlying functions, which has some implications on the features offered by the user interfaces of components that directly present a UI or indirectly provide functionality or data in the UI.

The complete requirements list in D2.3 list has initially been collected from the use case companies and then been completed by the other project partners. Those requirements are related to the original component names (cf. Section 3.2.2 for more information) and their impact on D3.1 can be summarized as follows:

1. End user requirements (main group one) consists of the following subgroups:
    a. Usage types requirements group defines the target users of the system, from personal car drivers to truck drivers, from cyclists to pedestrians, from passengers to disabled drivers. The goal of this set of requirements is to cover in detail all kind of user that will be using SIMPLI-CITY as well as to prepare ground for their specific demands to be counted for in the design and development of the project.
    b. User experience group is a vast and detailed set of requirements that include strategic and tactical user goals (e.g., help to become green and reduce number of accidents as well as help to control traffic in a particular area). Also user interface is addressed here, in areas like multilingualism, usability and easy configuration. Proactive behaviour of the system, ability to provide offline access to data and social features like providing feedback and implementation of generic services (e.g.,contests, alert services) are among sub requirements in this section.
    c. The App marketplace section of requirements explains what a SIMPLI-CITY marketplace should deliver in the sense of payments, app managing/versioning/upgrading, quality assurance and ease of use.
    d. The Specific apps requirements section covers requirements a user would have for a car installed app. For example, this may be a possibility to adapt to changing road conditions, resume interrupted trip planning and also optimization of costs of the user by choosing cheaper parking and toll free roads.
    e. The Environment section addresses requirements to the surrounding app ecosystem, like hardware platforms where the app can run, possibility to store user data in the cloud as well as connectivity with other systems and data sources.
    f. Privacy and security groups of requirements cover an important area a modern system should support. These range from transparency to access control, from confidentiality to data encryption, from integrity to secure access by third party systems.

2. Developer requirements (main group two) consists of the following subgroups:
    a. Requirements from the Technical model section address areas of development like robustness and backward compatibility of apps and different APIs, defining access to different sensors, programmatic components and components of the car and data sources.
    b. The Business model requirements deal with financial, promotional aspects as well as monetization of the project.
    c. The App Marketplace requirements are grouped by tasks of publication, versioning, submission to the marketplace and removing of apps.
    d. The Services Marketplace deals with similar requirements, but related to publishing of services instead of apps.
    e. The Developer studio set of requirements covers the vast area of functionality of the developer studio set of tools, ranging from simulation and debugging, technical statistics and feedback, documentation, SDK (Software Development Kit) and APIs involved.
    f. Developing requirements define the way of interaction between apps and services, other exchange of data, user notifications and unified interface for user centric data.

## 2.3 Implementations

The architecture in section 3 is based on detailed discussions of the project team together with input from the Description of Work and from deliverables D2.1 and D2.3. As such, the architecture has been chosen in a way that all requirements of D2.3 can be fulfilled and the components have also been described in this way.

This does, however, not mean that all those requirements will be included in the prototype implementation of the WP4-6 components because the implementation will be realized based on the priority of the requirements and not all requirements may actually be implemented because of the nature of an RTD project which has to deal with limited time and resources for the prototype implementation tasks.

# 3 SIMPLI-CITY Architecture Overview

## 3.1 Overview

The diagram in Figure 2 gives an overview about the high-level architecture of SIMPLI-CITY. It is based on the concepts of the project Vision Consensus Document and the Description of Work but in contrast to those documents, it focuses on the components instead of the tasks and work packages. Both views are strongly related to each other but the components of SIMPLI-CITY are handling the concerns of the project in a more detailed way and also show the connection between them. As shown in the figure, the components of SIMPLI-CITY may be split into four different areas:

The first area is entitled with “Vehicle & PMA” and as such it focuses on the personal mobile device which will be the most visible element of SIMPLI-CITY from an end user perspective. Components located in this area contain the application runtime environment as well as all mobile apps. Access to car sensors is also covered via a reduced implementation of the sensor abstraction and a local storage is provided via a simplistic implementation of the data storage infrastructure. User Interaction is performed via the multimodal UI and its dialog interface. The area has three relationships to other SIMPLI-CITY components. The first one is the interaction between apps and services, which is completely handled by the application runtime environment which communicates with the service runtime environment. The second one is the data prefetching which will be realized based on a stream connection to the data prefetching server side. Finally, the third relationship is the app store which allows users to access apps from the market using a UI inside the device. This UI communicates with the market backend on the server side where all app market information is stored and provided.

The second area of the SIMPLI-CITY architecture is covering components that will jointly realize the European Wide Service Platform of SIMPLI-CITY. Those components will be located at the “server side” of SIMPLI-CITY meaning that they do not run inside the mobile device. The main component of the server side area is the Service Runtime Environment hosting and controlling all deployed services. It also coordinates the communication with the PMA and contains the components for context based personalization and data prefetching. The server side also covers the data storage facilities, which will be realized by the cloud based information infrastructure. Additionally, the data access of SIMPLI-CITY to external data sources will be handled by the server side components. This contains the communication with external sensors but also with user centric and open data information sets. Finally, a set of web consoles will be realized which target developers making it easy for them to access information from SIMPLI-CITY including the monitoring of services as well as the submission of bundles to the marketplace.

The third area of the architecture represents external data sources (sensors, information sets, calendars, etc.) and the final area covers those components that support developers in the creation, deployment and updating of apps and services.

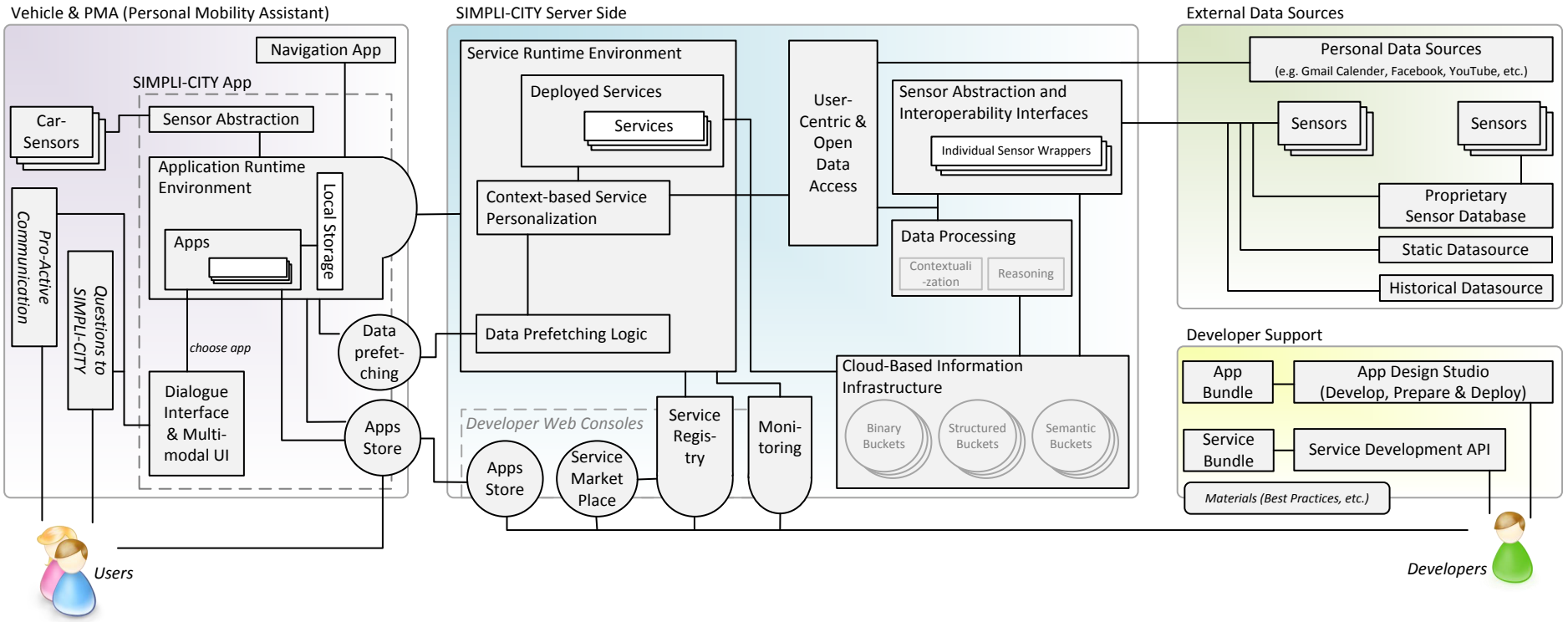


Figure 2: SIMPLI-CITY Architecture

## 3.2 Components: SIMPLI-CITY Server Side

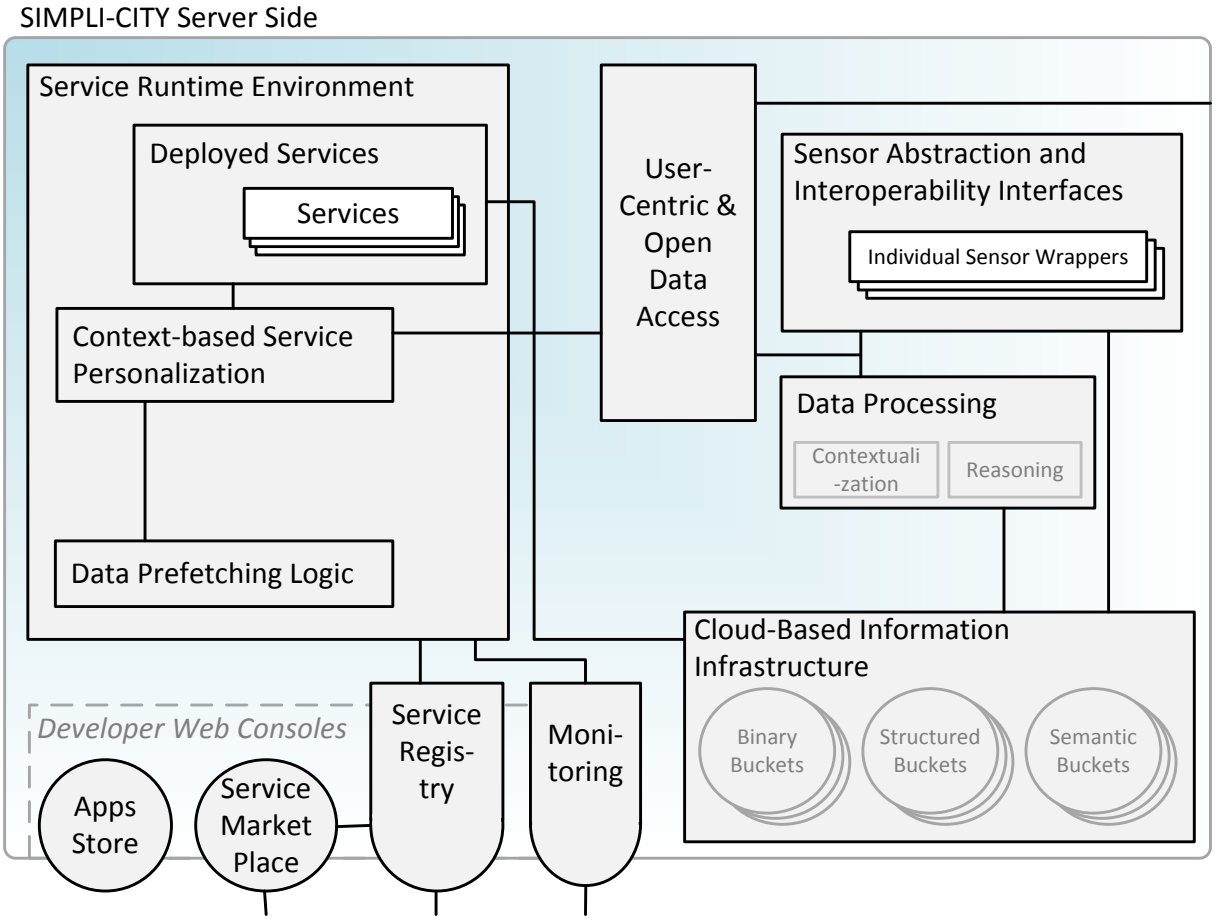


Figure 3: Server Side

The following table shows the components that are not located on the personal mobile assistant. Those components together create the European Wide Service platform, which is a major outcome of SIMPLI-CITY.

Table 1: Component Overview

| Component | Covered by Task | Discussed in Section |
|---|---|---|
| Service Runtime Environment | T5.3 | 4.1 |
| Media Data Streams / Data Prefetching Logic | T4.5 | 4.2 |
| Context-Based Service Personalization | T5.2 | 4.3 |
| Monitoring | T5.3 | 4.4 |
| Service Marketplace | T5.4 | 4.5 |

| | | |
|---|---|---|
| Service Registry | T5.3 | 4.6 |
| Cloud-based Information Infrastructure | T4.2 | 4.7 |
| User-Centric/Open Data Access | T4.4 | 4.8 |
| Sensor Abstraction and Interoperability Interfaces | T4.3 | 0 |
| Processing of Information from Data Sources | T4.3, T4.4 | 4.8 |

There is a range of external data sources which are shown in the following diagram. Those data sources are shown as examples; however, they are provided by external data suppliers and are therefore not components to be implemented within SIMPLI-CITY. They are wrapped and accessed via T4.1 and T4.3.

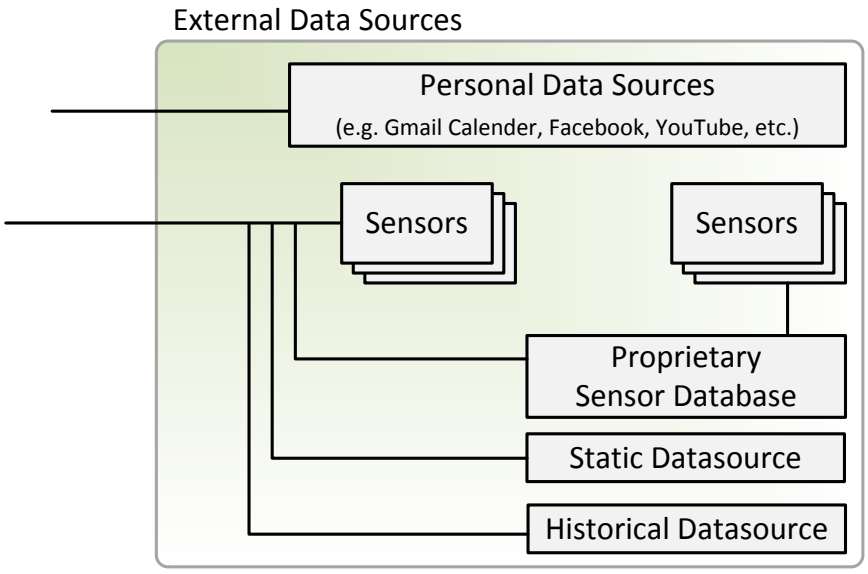


Figure 4: Data Sources

## 3.3 Components: Personal Mobility Assistant

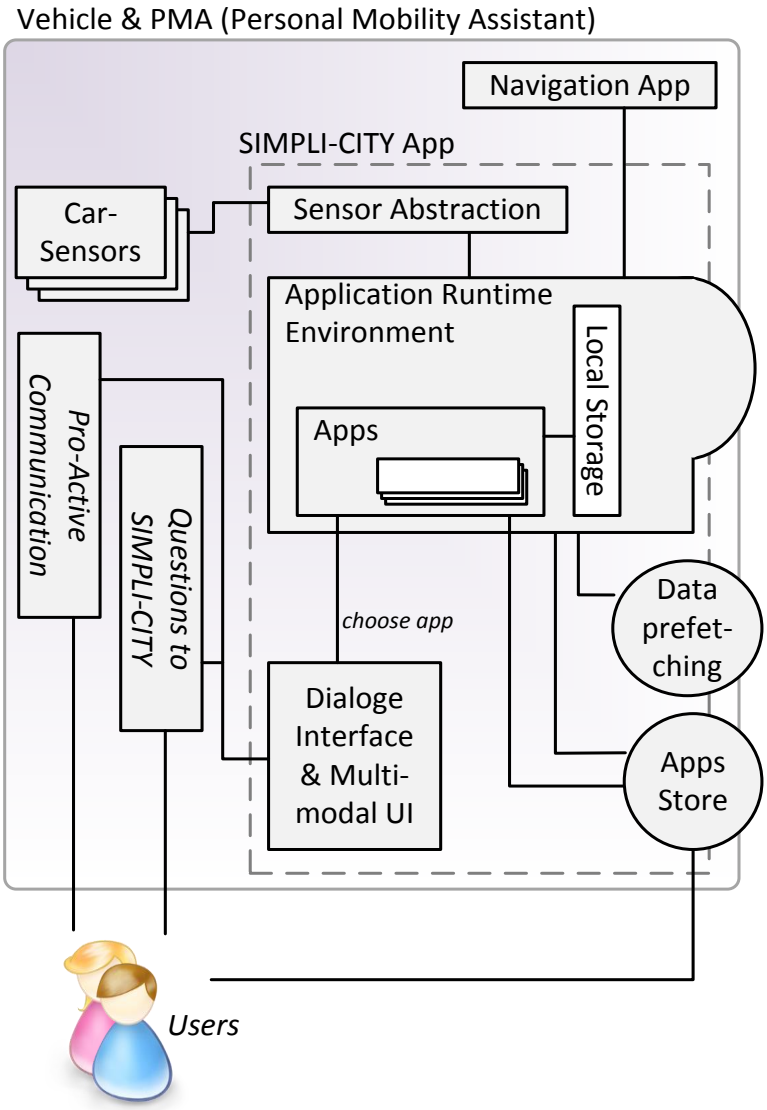


Figure 5: PMA

The following table shows the components that are located at the personal mobile assistant. Those components are running on the device and will mainly take care of user interaction, while services will provide backend logic which may be consumed by apps.

Table 2: Component Overview

| Component | Covered by Task | Discussed in Section |
|---|---|---|
| Application Runtime Environment | T6.3 | 5.1 |
| Application Marketplace | T5.4 | 5.2 |
| Dialogue Interface | T6.1 | 5.3 |
| Multimodal User Interface | T6.2 | 5.4 |

| | | |
|---|---|---|
| Sensor Abstraction | T4.3 | 5.5 |

## 3.4 Components: Developer Support

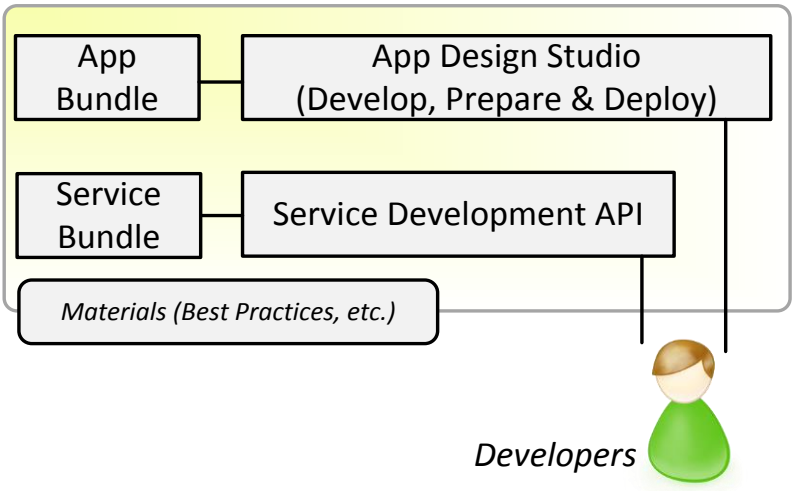


Figure 6: Developer Support

The following table shows those components that the project will provide as support for developers. This includes the studio of SIMPLI-CITY, but it also includes the provision of supportive materials such as documentation, best practices and How-To documents. Please note that multiple components will provide a small web UI for developers, e.g., to submit new services to the marketplace or to view the monitoring results of their services. As such, this entry of the table points to multiple locations of this document.

Table 3: Component Overview

| Component | Covered by Task | Discussed in Section |
|---|---|---|
| Application Design Studio | T6.4 | 6.1 |
| Service Development API | T5.1 | 6.2 |
| Materials & Documentation | T6.4, T5.1 | 6.1, 6.2 |
| Developer Web Consoles | T5.4, T5.3 | 4.4, 4.5, 4.6 |

# 4 Server-Side Components

This section describes the server-side components of SIMPLI-CITY. Those components will handle service executions, cloud based storage requests and issues like data source integration and access to external sensors. The communication between those components and the apps of SIMPLI-CITY will be performed via the application runtime environment and the service runtime environment.

Please note: Each of the following sections will close with a short subsection discussing *possible* technical foundations. Those may be seen as a baseline choice for the implementation of the components. It should, however, be stressed that those choices are not fixed as they will obviously depend on the outcomes of the functional and technical specification documents. As such, they should rather be seen as input for D3.2.1/D3.2.2 and they do not aim in discussing a fixed or conflict-free technology choice.

## 4.1 Service Runtime Environment

The Service Runtime Environment is the fundamental server-side backend component in SIMPLI-CITY. It provides a deployment and execution framework for (composed) services, which offer the business logic for end user apps in SIMPLI-CITY, and provides additional features that are required in conjunction with the execution of services.

Hence, the Service Runtime Environment encompasses the core features that are essential to deal with services at runtime, rather than design time, i.e., all functionalities required after the successful development and registration of a service.

For that purpose, this component provides the following key features:

- Execution of services: Creation of individual service instances (technically, objects) based on previously developed and registered services (technically, classes), and subsequent invocation of these service instances.
- Service Level Agreement-based Monitoring and Management: Logging of the non-functional behaviour of services during their execution, detection of deviations or failures on the basis of pre-defined Service Level Agreements, and – where applicable – invocation of corresponding reactions or countermeasures. This functionality is implemented in a separate component, the Service Monitoring (see Section 4.4).
- Accounting: Tracing the invocation of services, hence providing input data for a subsequent billing process.
- Proxy-based support for late-binding and third party services: Late binding of context-dependent data services as well as integration of existing, third party, i.e., project-external, backend services through a proxy concept. This facilitates context-aware routing of queries to matching data services and wrapping external services, hence treating them in a similar manner as internal services.
- Communication with apps: The Service Runtime Environment supports two different approaches for apps to interact with services. First, the “classical” pulling approach based on service requests, i.e., apps request information from services and services respond to the request. Second, a pushing approach, allowing to automatically send a notification to an app which can then request data from a service or some other data source.

### 4.1.1 Component Definition

Given the large body of well-developed standards and technologies in the area, the Service Runtime Environment will be built on the concept of *Web services*. In the Functional and Technical Specifications (D3.2.1 and D3.2.2), it is to be defined if the so-called WS-* stack[1] or REST-based[2] services will be used for the description and execution of services.

#### 4.1.1.1 Service Host

For the actual execution of service instances, a Web server component, such as Apache Tomcat, will be used that acts as host. Once all service artefacts, i.e., functional and non-functional descriptions, as well as the actual implementing classes, have been uploaded to the Service Registry, they will automatically be deployed on the server, hence providing a technical endpoint for service developers to bind to. Upon execution, instances of the service will automatically be created by the server, hence providing isolation between different service consumers. The Service Host will be fully configurable both with regards to single aspects (e.g., backup intervals) as well as the underlying components (e.g., in order to distribute them on different machines for performance reasons).

#### 4.1.1.2 Client Library

The Client Library helps to hide boilerplate code calling a service from the app developer, thus flattening the learning curve and helping to avoid common mistakes. The Client Library is supplied as a JAR file to be included in an app project. It allows invoking named services, automatically enveloping the payload, supplying typed parameters, parsing the service response, or returning a typed instance.

The Client Library is also responsible for connection management. It handles connection errors and resumes interrupted connections where possible, re-requesting lost data. Furthermore, it provides security features by taking care of authenticating the calling app as well as providing secure data exchange, based on, e.g., Secure Socket Layer (SSL). The library allows the app to transparently participate in a session, if the service called supports that.

The Client Library is not a mandatory component to be used in the app development. The developer may select to implement all underlying functionality themselves. In fact, the library is supplied as a convenience layer and can be viewed as a reference implementation.

#### 4.1.1.3 SLA Management and Enforcement

The Service Runtime Environment is able to measure uptime and availability of services it hosts and log service invocation statistics.

[1] The term “WS-*” commonly refers to the set or stack of standards and standard proposals that have been made with respect to traditional Web services. The stack contains standards such as SOAP, WSDL, and WS-Security, many of which share the common prefix “WS”. For further information, please refer to http://en.wikipedia.org/wiki/List_of_Web_service_specifications.

[2] Representational State Transfer (REST) is an architectural paradigm for distributed software systems, proposed by Roy Fielding in 2000. It is often interpreted as a lightweight alternative to traditional, WS-* based services. Further information can be found at http://en.wikipedia.org/wiki/REST.

The Service Runtime Environment logs service invocation on per app user session level, thus accumulating usage statistics; through the Service Monitoring components, it is also possible to collect information about unsuccessful calls and underlying reasons, such as Central Processing Unit (CPU) overload, insufficient memory, lack of other resources, errors and exceptions in the service implementation, or malformed app requests (see Section 4.4). The SLA Management component allows detecting Distributed Denial of Services (DDoS) attacks or understanding the necessity to allocate extra system resources to a heavily used service.

The SLA Management flexibly controls the service usage quota, and, once an app exceeds number of requests and data, it is allowed to consume per user session or more widely per application per month, it will stop processing further requests until quota is restored at the beginning of the next period.

The feature accumulates the minimally basic usage information in an, e.g., NoSQL database, and then a batch job processes this information offline and generates user-friendly reports. Generated reports allow for easy visual understanding of the overall system health, possible performance degradation, bottlenecks and places of improvement. The alert system is built into the SLA management facility and informs system administrators at runtime about significant problems it is able to detect.

#### 4.1.1.4 Accounting

In order to permit for the proper accounting and billing of service use within the service marketplace, the Service Runtime Environment features a simple logging capability. The log contains the timestamp, the service ID, and the respective service consumer for each service invocation. This data is provided via a lightweight Web service interface (or a common database) to the Service Marketplace, which can – possibly in conjunction with metadata from the Service Registry – subsequently create actual billing statements from it, based on the respective billing scheme, and control the settlement process.

#### 4.1.1.5 Proxies and Proxy Generation

In order to permit for the integration of external services into the SIMPLI-CITY platform – i.e., services that are hosted by a third party, rather than originally published within the SIMPLI-CITY platform – the Service Runtime Environment will support proxy services and feature a corresponding proxy generator. This component will also facilitate a late binding of data services (e.g., traffic or parking information services), hence permitting the dynamic selection of such services depending on the user context.

For those two purposes, proxies offer four distinct, yet possibly combined functionalities: First, proxies provide a wrapper around existing Web services, such that exclusively the wrapper is exposed to the service consumer and the underlying wrapped service is invoked in a transparent manner. Hence, in its simplest form, the proxy acts as a relay that forwards inputs from the consumer to the wrapped service and forwards output vice versa. This permits to technically treat external services as if they were originally published within the SIMPLI-CITY platform, and hence, allows applying the accounting, monitoring etc. mechanisms that have described above to treat internal and external services in a similar manner.

Second, proxies abstract from different technical implementations of services, such that SIMPLI-CITY exclusively provides homogeneous service implementations to potential consumers. Technically, such abstraction will also be achieved using a proxy that acts as

wrapper. However, in contrast to a simple relay, this proxy will additionally provide the capability to convert input and output messages into the appropriate formats.

Third, for predefined categories of data services, proxies will be implemented. These proxies exhibit a predefined interface, i.e., methods and message data structures, for each specific category. This interface definition is precisely resembled in the specific data services that depend on different data sources. Through dynamic routing of service requests, based on the user context, such proxies will permit for a dynamic selection of different data services at run time.

Fourth, the data and media stream prefetching functionalities as described in Section 4.2 will also be realized through proxies.

While services for late binding, i.e., dynamic selection of data services, as well as the data prefetching proxies will be predefined within the project, proxies for the remaining two purposes may be dynamically created using a proxy generator. While the monitoring aspects of a proxy will be provided automatically, the abstraction from technical implementations may need to be supported by the service developer. For this, SIMPLI-CITY will provide according assistance, e.g., through the creation of commented proxy stub source code based on the WSDL description of an external service.

#### 4.1.1.6 Pushing Support

In addition to “traditional” services that follow a pull-based request-response invocation pattern, the Service Runtime Environment will also support services that are based on pushing. Such push services comprise of two major components: First, a management service that permits clients (here: apps) to register or unregister for push notifications, possibly specifying the specific information that is of interest. This service can follow a traditional request-response philosophy, i.e., it is only executed when needed. Second, an actual notification service, which establishes the link to the underlying data source (e.g., traffic information), and pushes the relevant notifications to the registered clients (apps). The latter service has to be continuously executed by the Service Runtime Environment, thus minimizing latency between data updates and push notifications.

### 4.1.2 Interaction of the Component

As the central component for the execution of SIMPLI-CITY backend services and as the basic interface between the data sources and the end user apps, the Service Runtime Environment will interact with a number of components. In the following, only direct interactions are listed in order to focus on the most important interactions:

- Interaction with Application Runtime Environment: The Service Runtime Environment offers the backend service endpoints for the mobile apps executed in the Application Runtime Environment. Therefore, all end user apps interact with the Service Runtime Environment; the Service Runtime Environment then invokes further components, if necessary for providing the user with the necessary functionality.
- Interaction with Context-based Service Personalization: The Context-based Service Personalization component can be invoked by the Service Runtime Environment to permit for the adaptation of service instances to the user context, e.g., location.
- Interaction with Service Registry: The Service Registry provides the actual service artefacts, e.g., functional and non-functional descriptions and implementing service

classes, e.g., a Web Application Archive (WAR) file) to the Service Runtime Environment. Furthermore, functional and non-functional service descriptions are centrally stored in the registry. Hence, the Service Runtime Environment will retrieve the required description artefacts for the instantiation and execution of services from the registry. Corresponding documents from the registry, such as Service Level Agreements, will also be used for monitoring and management purposes.

- Interaction with Service Monitoring: The Service Runtime Environment interacts with the Service Monitoring for the purposes of SLA management and testing. In fact, the Service Monitoring will be a subcomponent of the Service Runtime Environment.
- Interaction with Service Marketplace: The Service Runtime Environment provides a log of service invocations to the Service Marketplace via a lightweight Web service interface or a common database.
- Interaction with Media Data Streams & Data Prefetching: The data prefetching mechanisms will be realized through proxies.

### 4.1.3 Possible Technical Foundations

A large body of software has been developed in the area of Service-oriented Architectures in recent years, substantial portions of which are available under liberal open-source licenses. Thus, SIMPLI-CITY will aim to adapt and extend existing frameworks, rather than implementing a solution from scratch.

Specifically, an existing Web server will be selected, e.g., Apache Tomcat, as the basis for the development of the Service Runtime Environment. Hence, the actual services will be implemented in a programming language like Java and defined using corresponding annotations, which permits to exploit a large set of tools for development, testing, debugging, and subsequent publication and execution of services.

The Service Runtime Environment will be hosted within a virtual machine throughout the project, based on, e.g., the VMWare ESXi or Xen virtualization solutions. This approach permits to flexibly migrate the component between different physical hosts, e.g., for scalability reasons, and create snapshots or branches within the development and testing process.

The actual transmission of push messages will be realized through a third party service, e.g., Google Cloud Messaging, thus exploiting the capabilities and features of such messaging infrastructures, e.g., queuing of messages or in-depth integration with the mobile operating system.

## 4.2 Media Data Streams & Data Prefetching Logic

### 4.2.1 Component Definition

This component will handle the data prefetching and media streaming aspects of SIMPLI-CITY. As such, it may be split into two aspects:

- Streaming and prefetching (personalized) media information by reacting to app requests and pre-buffering media data. This includes the transcoding of media data in order to be adaptable to the current data connection quality.

- Prefetching information from data services by automatically invoking data services and buffering their content. This includes automatic selection of data services as well as data expiration for outdated data.

#### 4.2.1.1 Media Streaming

The component will allow media prefetching (e.g., for music streaming). This may be used to create apps that support the playback of music or other media information. By nature, the consumption of media is sensitive to interruptions: Even a connectivity loss of a few seconds will give the user a bad experience if it happens in the middle of a song that the user is listening to. For this purpose, the component will integrate a media buffering solution by prefetching relevant data in a local buffer.

As data prefetching can lead to faster battery depletion of the mobile device, SIMPLI-CITY may develop new, context-based data prefetching mechanisms. This means that context data about, e.g., the user's location and destination would be used in order to find a suitable degree of data prefetching. If context information indicates that the user will soon be in an area with poor cellular coverage, the data prefetching mechanisms would retrieve a relatively large amount of media data in order to guarantee playback. The prefetching mechanisms would then be able to distinguish between a short-term (e.g., the user drives through a tunnel) or long-term need for prefetching (e.g., the area with poor coverage is relatively large or the user wants to download as much as possible from a wifi- instead of a 3G-based data connection).

Context data including usage statistics might be stored in the Cloud-based Information Infrastructure in order to provide users with enhanced experience. However, usually the actual data streams will *not* be stored within the Cloud-based Information Infrastructure, although the Cloud-based Information Infrastructure may manage information about where and how streams may be accessed. Despite this usual way, media data may of course – like all other files - be stored in the cloud if needed and the cloud storage will provide a possibility to stream them.

#### 4.2.1.2 Media Transcoding

The prefetching of data will be divided into two parts for media information: A server side component will be used to optimize the media stream based on the current connection quality and the bandwidth. For example, it may deliver music with a lower bitrate when detecting that the user is using a slow connection and with a good quality when the user is using e.g. Long Term Evolution (LTE). In addition to this, the device part of this component will provide a local storage for buffering the prefetched data. Transcoding will be performed on-the-fly for supported file types. SIMPLI-CITY will support typical formats such as MP3 and certain video formats (e.g. H264).

#### 4.2.1.3 Data Service Prefetching

As described above, the component will allow media prefetching (e.g., for music streaming) and service prefetching. Service prefetching will invoke services which the user is likely to use in the next time (Figure 6). This is performed by monitoring the user's behaviour and combining it with historical behaviour of SIMPLI-CITY users. The service selection will be made on the device, based on local user settings and local sensor data, and on the server side, based on historical behaviour of SIMPLI-CITY users. The results of service invocations will be buffered in a local prefetching storage.

For example, the component may be used to automatically receive a list of free parking lots as soon as the user drives to Barcelona downtown. It may then inform the user saying "Dear user, most people driving to Barcelona are looking for free parking lots. I've fetched this information for you automatically. Shall I launch the corresponding app for you?".

For realizing this feature, data services will be automatically executed. The list of executable data services comes from the apps installed on the device: Apps will be able to use the manifest to define weather a data service is used which is prefetchable and how long it takes until this data is outdated.

#### 4.2.1.4 Data Expiration

This component will prefetch the corresponding information and keep it in a local buffer for the time that it's requested by an app. This data prefetching is not possible in all cases as some data might be time critical and cannot be buffered for a long time. For example, parking information about free parking lots will be outdated after e.g. 20 minutes. Therefore, the component will support the expiration of data, which means that prefetched data will expire when it becomes too old. For supporting this, an app may mark services which it invokes as "prefetchable" data services and it may specify an "expired-after" flag to mark the data as outdated after a certain amount of time.

### 4.2.2 Interaction of the Component

The component will receive data and deliver from/to the following elements:

- Cloud Storage: The component may make use of the cloud storage environment in order to receive media data that has been stored in the cloud. For this purpose it will access to binary data buckets and it will use the server-side part of this component to transcode media if necessary. Transcoded media will then be streamed to the PMA side of this component.
- User-Centric Data Sources: When streaming music information or other media, users will most likely want to access their personal media, e.g. their private music library. For this purpose, this component will interact with the user centric data sources via task 4.4.
- Data Services: For prefetching data (e.g. free parking lots or a list of gas stations), data needs to be received from data services via the service runtime environment.
- Apps: Data will be provided to apps and as such, the component will act as a data source for apps installed on the PMA.
- Context Information: Prefetching data and/or media does not always make sense. For example, prefetching free parking lots in Barcelona does not make sense if the user is located in Madrid. As such, the component will make use of the T5.2 results for handing the users context.

### 4.2.3 Possible Technical Foundations

For media streaming information, this component will be split into two parts. The first part will sit on the application side and it will be able to request media data and to buffer it in a local cache. The second part will sit on the server side. It will handle the media data adaptation to the current user context. More precisely, it will transcode media data based on the current connection of the user. For example, media may be transcoded to low-quality mono encoded MP3 files if the user is connected with low-quality 3G and it may be

transcoded to stereo music with a high bitrate in case that the user processes a fast and stable LTE connection.

Prefetching data is only useful if it may be consumed by apps. As such, this component will act as a proxy for apps. This will allow apps to access data without having to modify their business logic. Data that has already been prefetched will be delivered to apps while other data requests will be forwarded to the original data services via the usual communication channel between the application runtime environment and the service runtime environment.

The component will allow apps to mark their data as prefetchable and to define attributes in their manifest such as the expiration date of data. This will help the component to distinguish between the data types and to detect the validity of data. Prefetched data will be selected and stored based on this information.

The component will make use of GPS information and other context based data based on the T5.2 outcomes. This will be used as input for the data selection process for data services.

## 4.3 Context-Based Service Personalisation

As the name already implies, context-based service personalization deals with services in SIMPLI-CITY that can make use of context information in order to realize personalized services for a particular user. In SIMPLI-CITY, context-based service personalization will be done with regard to the following aspects:

- Location-based data service selection: In many use cases in the SIMPLI-CITY scenario, a user wants to be provided with data that is related to the user's (current) location. For example, if a user is looking for a parking space in Barcelona, a data service providing information about parking spaces in Amsterdam will not help. There are two ways to achieve that the user is provided with the right data depending on the current location: Either, there could be a separate end user app for each city, which is tightly coupled to a corresponding data service. However, this means that the user needs to download and install a new end user app for each city. The second solution is to provide one general end user app, which invokes data services depending on the location of the user. For this, the Context-Based Service Personalisation component of SIMPLI-CITY supports the end user app by looking up the right data service based on the location of a user. This is achieved through a proxy, which reroutes requests accordingly. Notably, the proxy generation will be part of the Service Runtime Environment (see above); the Context-based Service Personalisation component is responsible for the logic to choose the right proxy.
- Proactive user notifications: Apart from reacting to particular user requests, services (and apps) should also be able to proactively provide information to the user, e.g., in the case there is an important update or if the user has predefined that to be reminded of ecological driving behaviour. For this, SIMPLI-CITY needs to calculate a value indicating the importance of a particular piece of information. In case a particular threshold is met, information should be forwarded to the user. Factors that need to be taken into account are the importance and value of the information as well as the driver distraction.

- Identification of prefetching-relevant context: This functionality enables to prefetch media data based on related user context data. This related data includes the upcoming journeys and locations (roaming, bad connectivity) of the user, including information about a possible mobile bandwidth cap and nearby WiFi networks where it is free to download data. Based on this information, the prefetching component (see Section 4.2) is able to reason if it makes sense to prefetch data and can trigger corresponding prefetching
- Context-based service execution: Apart from the location-based data service selection and context-based data prefetching, service executions in general could benefit from (user) context data, e.g., in order to automatically take into account user preferences. One central functionality needed to achieve this, is the identification of context data relevant in a specific situation.

### 4.3.1 Component Definition

#### 4.3.1.1 Context Manager

The Context Manager is the core component of the Context-based Service Personalisation. It collects and derives context information and delivers it to the components which invoke this functionality. Hence, it is the primary interface for other components. The Context Manager makes use of the Context Sensor component in order to subscribe to particular data sources and invokes the Context Reasoner in order to reason on it.

Context data might be information from a user profile, either provided by the PMA or stored somewhere else, the user location, information about different (context-based) variants of the same service, and information about service invocations. If necessary, the Context Manager will retrieve context data from eligible sources. For this, the Context Manager depends on an according context data model to be defined within SIMPLI-CITY. Furthermore, the Context Manager stores information about specific context sources in a local database. These context sources are accessed by context sensors.

#### 4.3.1.2 Context Sensor

Context Sensors subscribe themselves to particular data sources and provide this data to the invoking backend service and end user app. This allows the location-aware selection of different, loosely-coupled data sources through a single interface for backend services and end user apps.

Furthermore, context sensors monitor relevant context data sources and identify context events, i.e., if a change in context data is relevant for the user and therefore needs to be regarded within a service execution or when choosing a particular data service. Therefore, context sensors are supporting components invoked by the Context Manager.

Notably, different kinds of context data require different kinds of context sensors in order to identify relevant events. Context sensors also wrap different context sources so that the Context Reasoner and Context Manager can make use of a unified context data model.

#### 4.3.1.3 Context Reasoner

The Context Reasoner provides the logic for the different use case scenarios where context-based service personalization is envisioned. For instance, one particular use case,

with regard to the use cases mentioned above, is the possibility to derive the need to prefetch media data. Another one is the possibility to select data services based on the location of the user. Further use cases will be derived within the SIMPLI-CITY Use Case work packages.

Like Context Sensors, the Context Reasoner is a supporting component that needs to be invoked by the Context Manager. The Context Reasoner might either be based on Data Reasoning and Contextualization capabilities as presented in Section 4.8 or be implemented as part of a backend service, i.e., provide some service-specific, proprietary reasoning functionality.

### 4.3.2 Interaction of the Component

The component primarily interacts with server side components as has already been described above. However, context-based personalization heavily depends on context data, coming from various sources. Hence, the component needs to interact with these components that facilitate data access:

- Interaction with Service Runtime Environment: The Context Manager is invoked by the Service Runtime Environment if some type of service personalisation as described above can be triggered for a particular backend service or end user app.
- Interaction with Service Registry: Information about data services may be stored in the Service Registry. The Context Manager needs to be able to access this information in order to realise location-based data service selection.
- Interaction with Application Runtime Environment: Through the application runtime environment, the context-based service personalization component has access to the local phone storage and to sensors provided by the phone or to sensors which are accessible through the phone, e.g., if the phone is connected with the car. In the local phone storage, private user centric data is stored.
  Examples for this are app settings or other private preferences. Examples for phone sensors are GPS, i.e., for the current location, data-connectivity, i.e., if the device is connected via WiFi or via 3G, or does not have a working data connection at all. In addition to that, the phone might be connected to other devices, such as a car, and provides so access the current vehicle's speed, its health status or engine specific information.
- Interaction with User-Centric & Open Data Access: By connecting the Context-Based Service Personalization component with the User-Centric and Open Data Access component, it gains access to several different sensors and user centric data which are stored on server-side. This connection both sided, i.e. the Context-Based Service Personalization component can access data from the services and the other way around.
- Interaction with Data Prefetching Logic: The Data Prefetching Logic needs to interact with this component in order to determine if it is necessary to prefetch data for a particular service. For instance, if a user is in a car driving towards a tunnel it is very likely that the device's data connection may get interrupted. The information about that will be forwarded to the prefetching component, thus it can predownload necessary data, e.g., for the navigation app or music from an online stream.

### 4.3.3 Possible Technical Foundations

This component will be realized as part of the Service Runtime Environment (Section 4.1). It mainly targets technical experts (developers) and should therefore provide clean, well-defined interfaces, thus other services can make use of its functionalities and context-specific data.

If invoked, the Service Personalisation should neither increase the data payload nor the execution time of a service. Hence, the component should be designed in a lightweight, non-intrusive way.

Since this component shares the Service Runtime Environment with other components (deployed services, data prefetching, …) it is possible that these components will share certain libraries and other software components.

## 4.4 Monitoring

### 4.4.1 Component Definition

The SIMPLI-CITY Service Registry offers the functionalities to register, discover, and invoke several services. Each service may come with an optional SLA. These SLAs consists of several individual Service Level Objectives, each representing different requirements the service has to fulfil. Examples are Quality of Service (QoS) aspects like the maximum response time or the minimum amount of concurrent requests a service should be able to handle. In order to check if services are able to meet these non-functional requirements, each service invocation is monitored regarding:

- Services are responsive at all.
- Service Level Objectives are being complied with.

There are two basic approaches for monitoring of QoS aspects: Server-side monitoring can be applied when the actual service implementation is available and could be extended, as it is the case for the services deployed within the Service Runtime Environment. Client-side monitoring needs to be applied, if this is not the case. Then, either the service client or a proxy needs to monitor the service invocation. A monitor should be able to identify relevant events, i.e., if SLA obligations are violated, a particular event needs to be forwarded to the Service Runtime Environment so that according countermeasures can be executed. According countermeasures could be that an alternative service is automatically chosen (only possible for data services), an internal backend service is instantiated one more time, or the provider of an external service is informed about the SLA deviations.

Apart from the monitoring of SLA objectives, monitoring in SIMPLI-CITY also serves the purpose to generate service execution statistics and log error messages if they occur. Service execution statistics provide useful reports to service providers or services developers, including information about how many resources are used by single service invocations, etc. Error messages are provided to software developers to allow recapitulation of errors and to make use of this information in service updates.

#### 4.4.1.1 Monitoring Manager

The Monitoring Manager allows the configuration of the actual Monitor and interprets SLAs (as stored within the Service Registry). It is a helper component invoked by the Service Runtime Environment's subcomponent *SLA Management and Enforcement*.

#### 4.4.1.2 Monitor

This subcomponent provides the actual monitoring capabilities. It allows monitoring the following quantifiable aspects of a service:

- Service invocations including the service's invocation throughput over time.
- QoS aspects of each service invocation, i.e., availability and response time.
- Resource consumption of single services and service invocations.
- Produced costs, i.e. hand in hand with the resource consumption of internal backend services, the resulting costs will be monitored.

Based on expected behaviour (from the SLAs) and monitored data, the Monitor will generate events which indicate (negative) deviations from the expected behaviour. This allows other components, which receive these events, to start according countermeasures, if necessary. This allows error detection and fault handling in the Service Runtime Environment.

In addition, the Service Monitoring component may be used by the Service Registry in order to regularly control if data services are still alive.

#### 4.4.1.3 Monitoring Proxy

The Monitoring Proxy subcomponent allows the monitoring of external services (i.e., services not deployed within the SIMPLI-CITY Service Runtime Environment) by providing and configuring proxies for such services. These monitoring proxies wrap the functionalities of the actual monitor (see last subsection) for external services as far as possible. Configurations of the proxies are done through the Monitoring Manager. Service requests are automatically rerouted to the Monitor.

The Monitoring Proxy is managed by the generic Proxy Component of the Service Runtime Environment.

#### 4.4.1.4 Monitoring Data Storage

The Monitoring Data Storage component is responsible for the storage and retrieval of monitoring data. It makes use of the Cloud-based Information Infrastructure for this and provides a query interface which can be used by other components in order to retrieve this data.

### 4.4.2 Interaction of the Component

Being part of the SIMPLI-CITY Server Side components, the Monitoring component interacts with a number of other Server Side components:

- Service Runtime Environment: The main component the Monitoring Frontend is interacting with is the Service Runtime Environment, and through this, the single services. Monitoring of service executions is necessary in order to check if services are available at all and provide the necessary Quality of Service as may be defined in an SLA. Monitoring should be possible for both project-internal and -external

services. In contrast to external services, the internal ones are running inside the Service Runtime Environment. This can provide interfaces for monitoring the particular service. This is not the case for external services, where the service providers or service developers have to provide an interface which allows the Monitoring Frontend to retrieve the monitored data. In these cases, a proxy-based monitoring solution is also possible.

- Service Registry: The Service Registry stores the SLA for the single services. This information is needed in order to identify SLA deviations.
- Service Development API: Monitoring results are mainly interesting for the software developers and service providers. Information about how the service behaves under a given load, i.e., concurrent service requests, can tell them if the service works as expected or if it has to be improved. Further, it provides information about how many users are actually using the provided service or if the service is not used at all and can be removed in order to save resources for other services.
- Application & Service Marketplaces: Some monitored data might be interesting for users and developers. Examples are how many users are actually using a particular app/service or how much resources are needed for this service. This information will be provided to the App Store and the Service Marketplace.

### 4.4.3 Possible Technical Foundations

This component will be realized as part of the Service Runtime Environment (Section 4.1). As it is the case with the Service Runtime Environment, the Monitoring component should be based on existing frameworks instead of reinventing the wheel. It needs to provide well-defined interfaces which can be used to query monitoring data in other components, most importantly the Service Development API.

## 4.5 Service Marketplace

### 4.5.1 Component Definition

One of the main achievements that SIMPLI-CITY will deliver is its flexible way of adding new functionalities for road users. Those new functionalities will be provided by installing new apps on the PMA. Each app will usually be wrapped around one or more services, which will deliver data and perform server side computations. Services may be hosted by the service runtime environment and may be added to SIMPLI-CITY dynamically. If services are not hosted by the runtime environment, then a proxy service will be located at the runtime environment for controlling and monitoring access to this service. As such, all invocations will be routed via the service runtime environment. This allows developers to provide new services and to develop new apps on top of those services. Service providers may also offer their services – either free of charge or in a commercial way. With this model, SIMPLI-CITY will create an ecosystem for service and app developers.

Within SIMPLI-CITY, services will be offered in the service marketplace. This service marketplace is part of task 5.4 (Mobility Service and Application Marketplaces). While the app side mainly targets end users, the service marketplace targets developers and will therefore not be directly seen by end users as they will access all functionalities indirectly via apps installed on their PMA.

#### 4.5.1.1 Web Based User Interface

Generally speaking, developers may use the service marketplace for two main purposes: Firstly they may use the service marketplace for providing services to other developers allowing them to consume them. This includes updating service endpoints or activating / deactivating entries. Secondly, developers may use the service marketplace for discovering SIMPLI-CITY enabled services. Those are services that run inside the service runtime environment of SIMPLI-CITY and may be consumed by apps via the runtime environment.

For providing those two core functionalities, the service marketplace will provide a web application for developers. This web application will be usable from any modern web browser and allow developers to register and to login for managing the services that they are using or providing.

Developers may have multiple roles at the same time. They may provide services to other developers and they may at the same time consume services, e.g. for developing own apps for making of a service call inside their own service implementation. As such, developers may become “prosumers” (providers and consumers) for SIMPLI-CITY.

#### 4.5.1.2 Providers: Service submission and Review

Service developers may submit their services to the service marketplace. Once a service has been submitted to the marketplace it will be marked as “under review” for developers. At this moment in time, the service will not be listed in the marketplace for service consumers. Instead, the SIMPLI-CITY marketplace team will be notified about the new service submission and will perform a manual testing of the service. Alternatively a set of automatic test routines could be provided to shorten this phase. For allowing this, developers will be able to specify testing notes to describe all requirements and dependencies for service reviewers such as license keys or user credentials. The goal of this review process is to ensure a high quality of SIMPLI-CITY and its services.

Once, the service has been reviewed, it may be accepted or rejected and feedback will be provided to the service provider. Services that have been rejected may be resubmitted by developers and services that have been accepted may be deployed to the service runtime environment in order to make the service listed in the marketplace for consumption.

Service consumers may also provide feedback about a service which is described in section 4.5.1.6. However, consumers may also flag a service as inappropriate or damaged. In this case, the review process will be restarted, allowing the SIMPLI-CITY team to take services offline if necessary.

The following image shows the review process graphically:

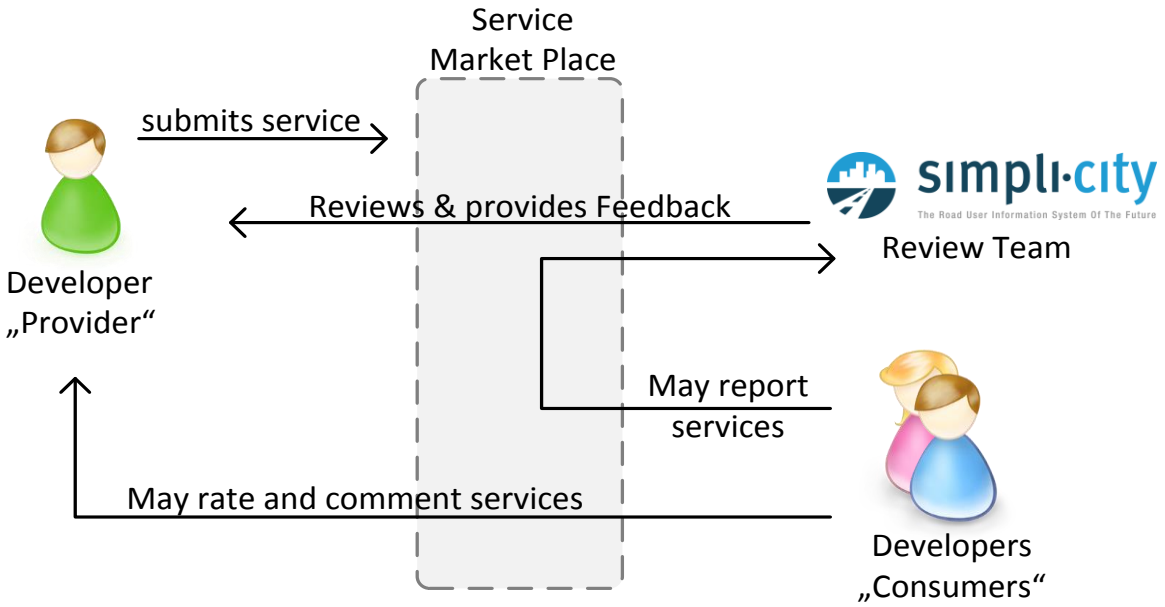


Figure 7: The SIMPLI-CITY review process for services

#### 4.5.1.3 Providers: Service Preparation

The process described in the last section raises the question on how services will be submitted in detail. For this purpose, service developers will be able to use a submission form on the web UI. This submission form will allow them to describe their services and to upload all files that are needed for the deployment. Those files include a technical deployment description in form of a technical manifest file and will be described in section 6.2 covering T5.1 (Service Development API). As such, the UI will use the same look & feel and a direct interlinking or integration with the UI of T5.1.

#### 4.5.1.4 Providers: Service Lifecycle, Registry and Deployment

Service providers may upload new service descriptions at any time and they may put services on hold by deactivating them. This process will then remove services from the list of available services in the marketplace. Whenever a new version of a service is uploaded, developers will be asked to keep their interfaces backwards compatible in order to avoid that an app gets broken because of a service change.

All service invocations will be routed via the service runtime environment. As such, services will not be deployed by the marketplace directly but by the service runtime environment. However, the deployment and undeployment process will be supported by the marketplace because link to the logging/monitoring interface of T5.1. Optionally, the current status of the deployment may already be indicated in the dashboard.

#### 4.5.1.5 Consumers: Service Consumption

Once services have been published to the marketplace and deployed to the service registry and runtime environment, they are ready to be used. As such, the service marketplace will provide a list of services which allows developers to quickly find services based on a pre-defined taxonomy (in terms of categories) and also based on keyword searches. Developers will receive a summary list and may click on a service to view its details including the service description and the service endpoint. This description will also

contain technical information and examples for using the service if provided by the service developer.

#### 4.5.1.6 Providers & Consumers: Developer Communication

Consumers may be able to rate services and to provide feedback in terms of comments. This will allow other developers to judge the quality and usefulness of a service based on other users. App marketplaces such as the Apple AppStore or the Google Play Market have shown that this can lead to valuable information and will also give an indicator about the reliability of a service.

Service providers will be able to use the service marketplace to view the rating and comments of their services and they will be able to respond to a comment, e.g. by helping a consumer with a specific problem.

#### 4.5.1.7 Providers & Consumers: Business Aspects

During the prototype implementation of SIMPLI-CITY services will always be free of charge. However, once SIMPLI-CITY leaves the prototype stage, it will support commercial, and non-commercial services. This impacts three different elements:

- Service providers will be able to select a payment model for their services. This might be ‘free to use’, ‘pay-per-use’, ‘monthly payment’ or ‘one-time payment’.
- Service consumers will be able to purchase a service and to use their favourite payment method (PayPal, credit card, debit, etc.) to pay for a service.
- The service runtime environment will ensure that services can only be invoked if the user has an active subscription to invoke it.

### 4.5.2 Interaction of the Component

The component will receive and deliver data from/to the following elements:

- Service Providers and Consumers: The component mainly consists of a graphical user interface for providing and discovering services. As such, the main interface will be to the providers and consumers of services by providing them a web based UI.
- Service Monitoring: Service providers will be able to use the monitoring platform of SIMPLI-CITY to view the status of their service deployments. This monitoring will not be provided by the marketplace but the marketplace will provide direct links from each service to the monitoring interface.
- Service Registry & Runtime Environment: Services will be executed within the SIMPLI-CITY service runtime environment. The runtime environment will take information about the technical nature of a service from an own service registry. As such the service marketplace will need to interact with the runtime environment and the registry by providing links for deploying and undeploying services and by providing them an interface that those components can use to request the license for a service, e.g. for checking if a service payment has been performed by a service consumer.

### 4.5.3 Possible Technical Foundations

From a technical perspective, this component will be realized as a web application providing a web based UI for service providers and consumers. It will be targeting

technical experts (developers) and will therefore be able to make use of domain specific languages and expressions. Nevertheless, the UI should be user friendly and easy to use. In order to maximize the usability of the web application, the service marketplace should be usable from any modern web browser. Since tablets are becoming more and more important, it should be avoided to use technologies such as Adobe Flash for the realization of the marketplace as they are not supported by many mobile devices.

In addition to the service marketplace, an app marketplace will be developed. This will consist of a web based backend for app developers allowing them to submit new apps to the market and of a PMA based frontend targeting end users who want to download and install new apps on their device.

The app and the service market therefore have an overlap in terms of technologies as they both will provide a web UI backend for developers for doing submissions. From a technical perspective, the service marketplace and the app marketplace will therefore share a certain set of libraries and UI elements. The details about this process will be defined within the technical specification of D3.2.2. The following figure shows the interconnection of both components.

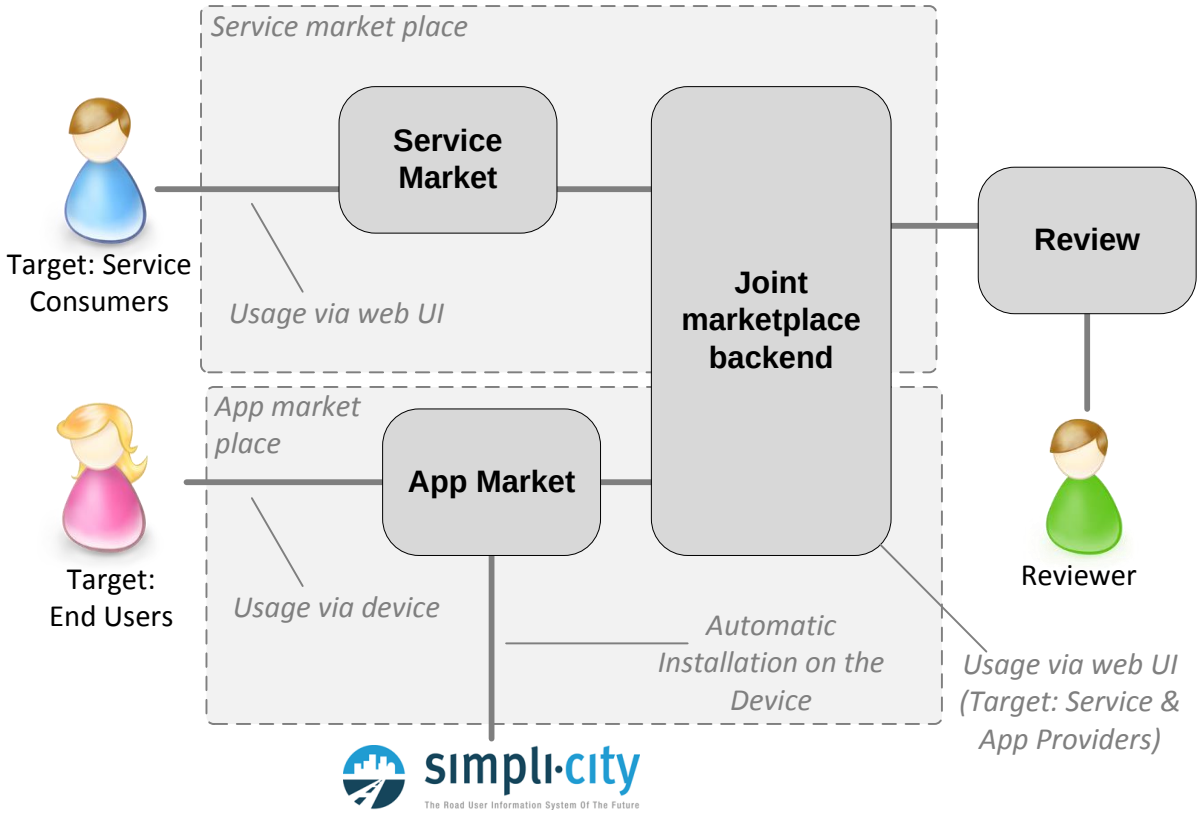


Figure 8: Overlap between service and app marketplace

## 4.6 Service Registry

### 4.6.1 Component Definition

In the real world, (business) services are described and listed in yellow page directories, so that customers are able to search for suitable services and their providers. Concerning software-based services, this requirement also needs to be satisfied. For this, service registries are applied. In general, a service registry is used to store information about services and find services based on some search parameters. Apart from this functionality as a service directory, in SIMPLI-CITY, the Service Registry also offers service repository functionalities, i.e., it is also possible to store service implementations which can then be deployed in the Service Runtime Environment.

The SIMPLI-CITY Service Registry only offers the backend functionalities to store and find services including the actual software artefacts and the service metadata. Its user interfaces is the SIMPLI-CITY Service Marketplace, which allows both service providers and consumers to store, register, and find services and alter both service information and the service artefacts (if stored).

#### 4.6.1.1 Service Registration

Service developers have to register their services so that they can be used by SIMPLI-CITY end user apps.

For this, two different options exist:

- Storage: If a service developer wants that a service runs within the SIMPLI-CITY Service Runtime Environment, the according software artefact needs to be uploaded to the Service Registry. This artefact is then stored in the File Storage (see next subsection). In addition, the software developer needs to submit service metadata, including information about the service interfaces, SLA information, usage fees, technical and security constraints, and information that is helpful for service developers and end user developers. The service endpoints will be automatically generated by the SIMPLI-CITY Service Runtime Environment and also stored within the service catalogue. Storing services includes registering.
- Registration: If services should not run in the SIMPLI-CITY Service Runtime Environment, but be hosted by some third party or the service provider himself, the service nevertheless needs to be registered, i.e., the abovementioned service metadata needs to be provided by the service provider. In addition, the developer will be supported in the deployment of a proxy, which is then automatically registered with the Service Registry and facilitates the actual execution and advanced capabilities, most notably monitoring.

Regardless if a service is stored or “only” registered, the service metadata will be stored within the service catalogues (see below). Service registration will also allow updating either service metadata or the software artefact (if applicable).

Regardless if a service has been stored or registered, it is possible to mark a service as “private”. In this case, a service will be registered and can be used by the service provider himself, but will not be listed on the Service Marketplace.

#### 4.6.1.2 Service Catalogue

The service catalogue is essentially a database allowing storing, retrieving, updating, and deleting service metadata as discussed above. This includes SLAs. The catalogue does not store service software artefacts.

In addition, the service catalogue allows storing service-related data needed by the service marketplace, e.g., service ratings and comments from users of the services.

#### 4.6.1.3 File Storage

The file storage stores the service software artefacts. Hence, it complements the service catalogue for these services that should not only be registered, but also stored.

#### 4.6.1.4 Service Selection

Once services have been registered they can be used by other services, in service compositions, and most importantly, by end user apps. In SIMPLI-CITY, backend services will be tightly coupled to end user apps, this means that the end user app states the ID of a specific service it wants to invoke. This ID will be provided from the end user app to the Service Runtime Environment, which will select a particular service based on it. For this, it looks up the service endpoint by querying the ID in the Service Registry.

The only services which may be loosely coupled within SIMPLI-CITY are data services, i.e., service-based access to the different SIMPLI-CITY data sources. This allows to automatically choose the right service for a particular context, e.g., a location, as described in Section 4.4. For data services, the invoking entity only states a taxonomy category of the service. This taxonomy category identifies data services which offer the same data for different locations and in the same format. If a data service is requested through its taxonomy category, the Service Runtime Environment will automatically invoke the Service Personalisation component, which will enrich the service request (query) with context information necessary to identify which of the services from the ones that are offered for a particular taxonomy category, should be invoked.

### 4.6.2 Interaction of the Component

The Service Registry provides a basic functionality needed in a Service-oriented Architecture. Hence, it will be invoked by several of the SIMPLI-CITY Server Side components:

- Service Marketplace: The Service Marketplace provides the actual user interface for providers and consumers of a service. It also allows to store and search service metadata, software artefacts, and further service-related information.
- Service Monitoring: The Service Monitoring component needs to access the SLA definitions of single services in order to be able to enforce them.
- Service Runtime Environment: The Service Runtime Environment invokes the Service Registry in order to look up where a service is located (service endpoint) based on a unique ID provided by the invoking entity (here: an app). Based on this information, the Service Runtime Environment is able to bind a particular service.
- Service Development API: The Service Development API is providing the user interface for the Service Registry.
- Service Personalisation: This component interacts with the Service Registry in order to identify the correct data service if context-based, location-aware data selection is

applied. For this, it searches services which match a particular taxonomy category or function.

### 4.6.3 Possible Technical Foundations

The Service Registry will be based on an existing Service Registry or Service Repository software like, e.g., ebXML (*freebXML*) or UDDI (*jUDDI*), if possible. The requirement towards the use of an existing solution is that it is able to store the actual software artefacts, i.e., a collection of classes, servlets, XML files, libraries, that together form an application, and some description of the service, e.g., in the form of metadata. For example, services could be stored as JAR or WAR files. Furthermore, it should be possible to connect the registry to an arbitrary database (for the storage of the service metadata), and to the Cloud-based Information Infrastructure (for the storage of the software artefacts).

## 4.7 Cloud-based Information Infrastructure

### 4.7.1 Component Definition

SIMPLI-CITY will allow users to benefit from a range of apps on their personal mobility assistant. Those apps will communicate with the server side of SIMPLI-CITY in order to make use of the service platform which will allow them to access powerful ways for receiving information and afterwards displaying it to the user or reading it to the user via the voice based components. Information that users will access within those apps may be provided dynamically from the services but they may also be based on persisted information from the SIMPLI-CITY data storage – the cloud based information infrastructure. This infrastructure will act as a service which is dedicated into managing different types of data in a persistent, scalable and efficient storage. The cloud based storage will be fed with information from apps (e.g. to store data from users) or from external data sources such as sensors or user centric data. It will be accessed via services, other components or apps but there will be no direct access from apps to the data storage: All access will be performed via the service runtime environment.

#### 4.7.1.1 Elasticity and Data Storage Types

Data will be managed in the Cloud, which will provide an elastic way of storing information. For components, apps and services, this will be hidden by the unified API. The storage itself will, however, be based on a distributable and scalable technology in order to prevent bottlenecks in terms of speed.

Generally, data stored by this cloud storage component may be of two different types: the first type is data which is up-to-date such as information stored by sensors or data about specific events. This data is referred to as ‘real-time’ information, e.g. a temperature value for a specific sensor. This data will be accessible in a very fast way. The second group is data which covers historical information. This may include historic temperature values or outdated traffic information. This data may be used to allow services and applications to query information from the past, e.g. in order to “learn” from former traffic data. Since the amount of data may be very large, the archive does focus on the scalability as first priority and speed as second. As such, queries on this information source may be significantly slower than queries on the real time data storage.

#### 4.7.1.2 Content Agnosticism

Data may be originating from all types of data sources, most prominently information coming from sensor sources including intelligent infrastructures and cooperative systems, as well as people-centric sensing, e.g., data coming from vehicle-internal sensors (telematics) and mobile end user devices that can be directly related to a particular person. Furthermore, data coming from open data sources and other social and sensor data sources will be storable. The Information Infrastructure is fully content-agnostic and as such, it does not know what the semantics of data are. It may therefore be compared to a database for managing information in a fast, scalable and reliable way. This concept is essential for the cloud-based information infrastructure as it allows an efficient implementation of data storage by separating concerns. This concept, however, means that apps, services and components need to decide on their own which information they want to store and how they want to receive it.

#### 4.7.1.3 Data Expiration

SIMPLI-CITY will develop mechanisms to estimate relevance of data changes and events regarding mobility, environmental impact, safety, and comfort. Only the data deemed to be relevant will be stored within the Information Infrastructure. For example, sensor data that is delivering 2,000 values per minute may not be of any value for SIMPLI-CITY if one value per hour is sufficient for apps and services. Since the decision about which data to store and which to dismiss is highly depending on the semantics of the corresponding data source/sensor, the cloud storage cannot decide it on its own. Instead, each sensor wrapper and service will decide this when feeding data to the storage.

However, the cloud storage will equip services, apps and components with mechanisms to define the “scope” of data by defining a rule which specifies when a data set becomes obsolete. As such, data may be flagged to expire after a certain time. After this expiration time, the cloud storage will either shift the data from the real-time data storage to the archive or it will delete the data. The control over this will be done by the apps, components and services using the storage. They will be able to define rules allowing the storage component to decide how long data is kept in the real-time data storage and when data is either deleted or archived. For example, a rule may define that temperature information is stored in the real-time data storage for two days and afterwards be moved to the historic data archive with one measurement per hour allowing the storage to delete the other data. As such, this mechanism allows data cleansing up to a certain extent, i.e., the identification of relevant and valid data and the removal of irrelevant data.

#### 4.7.1.4 Data Buckets

The component will allow services, apps and components to create isolated data storage spaces which will not overlap with those of other components, services or apps. Those isolated storage spaces are referred to as “Buckets”, which is a concept originating from the Amazon S3 storage solution and has proved to be robust in many Cloud-based storage solutions. The bucket concept allows the usage of different storage backends in order to support different types of data storage. The project will at least realize a NoSQL storage for structured JSON-based data, but may later also provide storage for other types, such as binary or even semantic data.

Attention will be paid to data privacy by putting adequate security mechanisms into place, in order to ensure that data is protected against malicious parties. This is performed by

allowing the data buckets to define their visibility using visibility tokens. Those tokens may either open a bucket to public read, public read-write, or restrict the data of the bucket to one specific component, user, app, or service.

#### 4.7.1.5 Local PMA Storage

Apps of the PMA may have the need to store some – limited - information on the mobile device. This may be settings such as the username or it may be payment information which should not leave the device by design. For those purposes, the PMA will provides a stripped-down implementation of the cloud-based information infrastructure, which will allow the storage and access of data in a key-value like way with a simplistic interface.

### 4.7.2 Interaction of the Component

The component will provide an interface and allow the provision and exchange of data (e.g. in a restful manner). The component will receive data and deliver from/to the following data sources:

- External Sensors: External sensors will not directly be connected to the data storage but via the sensor abstraction and interoperability interfaces. Those will provide sensor wrappers, which will handle the query of a sensor and forward this information to the cloud storage.
- Data Processing: Optionally, data may be routed via the data processing component which may filter information based on the sensor semantics and will then forwards it to the data storage.
- User Centric Data: Data from personal data sources such as calendars may be stored in the cloud based information structure. However, it should be noticed that personal information should only be stored in the SIMPLI-CITY storage in rare cases in order to respect the privacy of users and also in order to dynamically access data (e.g. from calendars) rather than replicating it to the data storage of SIMPLI-CITY.
- Services: Services may use the data storage to manage data for their calculations. This may include settings and user specific values. Services will always access the data storage via the service runtime environment.
- Apps: Apps may use the data storage only via the service runtime environment. They may use the storage to query information and to access data of the storage. However, in most cases, apps will use services for receiving and delivering data meaning that direct access to the data storage should be minimized.
- Components: All other SIMPLI-CITY components may use the data storage for their internal data management. For example, the market may use the storage to manage information about apps and services including their descriptions.

### 4.7.3 Possible Technical Foundations

From a technical viewpoint, the component will provide an open interface by making all its functionality available as service calls. This means that components will not be bound to a specific operating system or development language. Consequently, data that is received and delivered will also be wrapped in an open format based on e.g. JSON.

The data storage provides different data bucket types in order to support data storage of different kind of data: e.g., Binary, NoSQL, and Semantics. The concept of data buckets is

used since each of those data types is best managed by different implementations and concepts. For example, binary storage may be realized by a distributed file system and may even make use of existing cloud storage services such as Amazon S3. In contrast to this, semantic data usually has requirements such as advanced query functionality (e.g. SparQL), which is usually not needed for binary data. As such, another data storage implementation may be chosen.

In order to bring together those different requirements, the cloud storage will provide one holistic wrapper for different cloud storage types. This wrapper will provide basic CRUD (create, read, update, delete) functionality for all data types as well as advanced interfaces (e.g. SparQL) for specific data bucket types. Each component may create multiple data buckets and specify their type.

The storage wrapper will hide the physical implementation of each data store. Different technologies may be used for this and will be analyzed and selected during the functional and technical specification of SIMPLI-CITY. The different technologies should, however, be chosen in a flexible way, e.g. by using jClouds to access Amazon S3 storage instead of directly accessing it.

Figure 10 shows an overview about the structure discussed above:

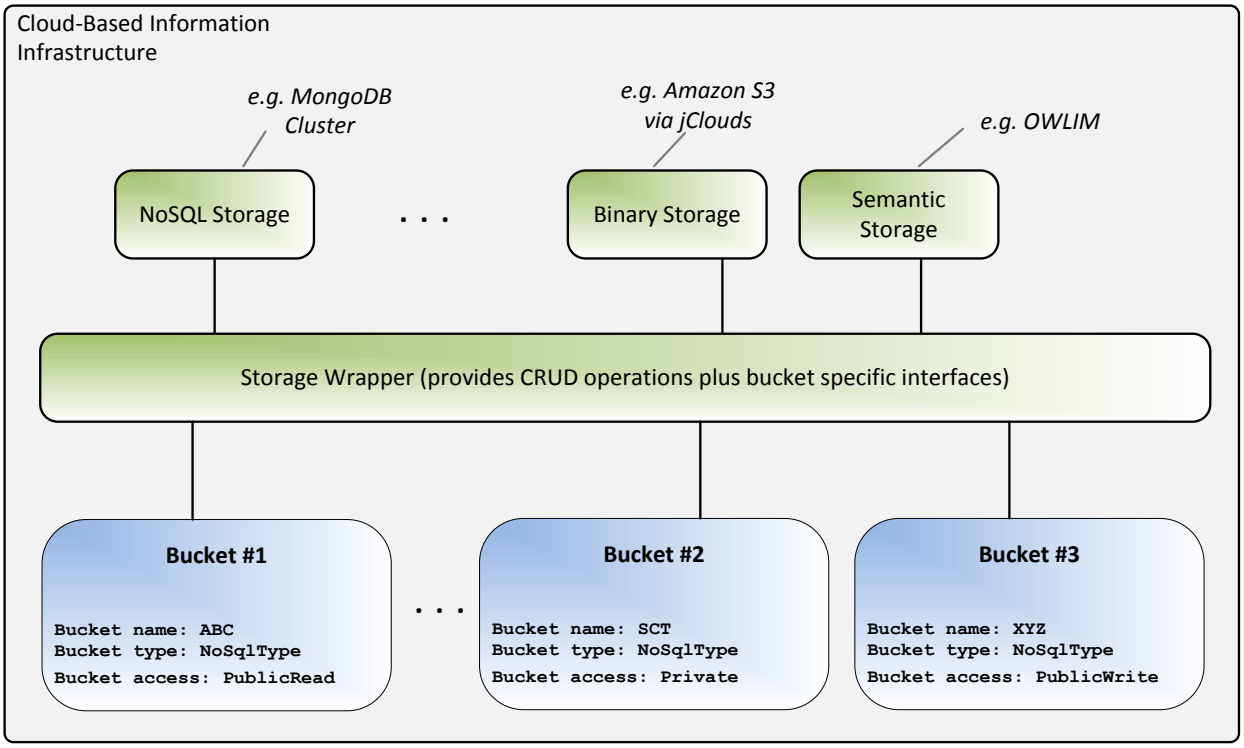


Figure 9: Structure of the data storage

## 4.8 Processing of Information from (User-Centric) Data Sources

The following diagram shows the overall architecture for Work Package 4 and its interactions. The main components of the diagram for this section are Contextualization/RDF-ization Reasoning Component, Open Data Access, Personal Data Sources and Service Runtime Environment interactions. The Cloud Storage is described in section 4.7 and the Sensor Abstraction and Interoperability Interface in section 4.9.

In the below diagram the thicker arrow represents ‘Request Flow’ and the thin arrow ‘Data flow’.

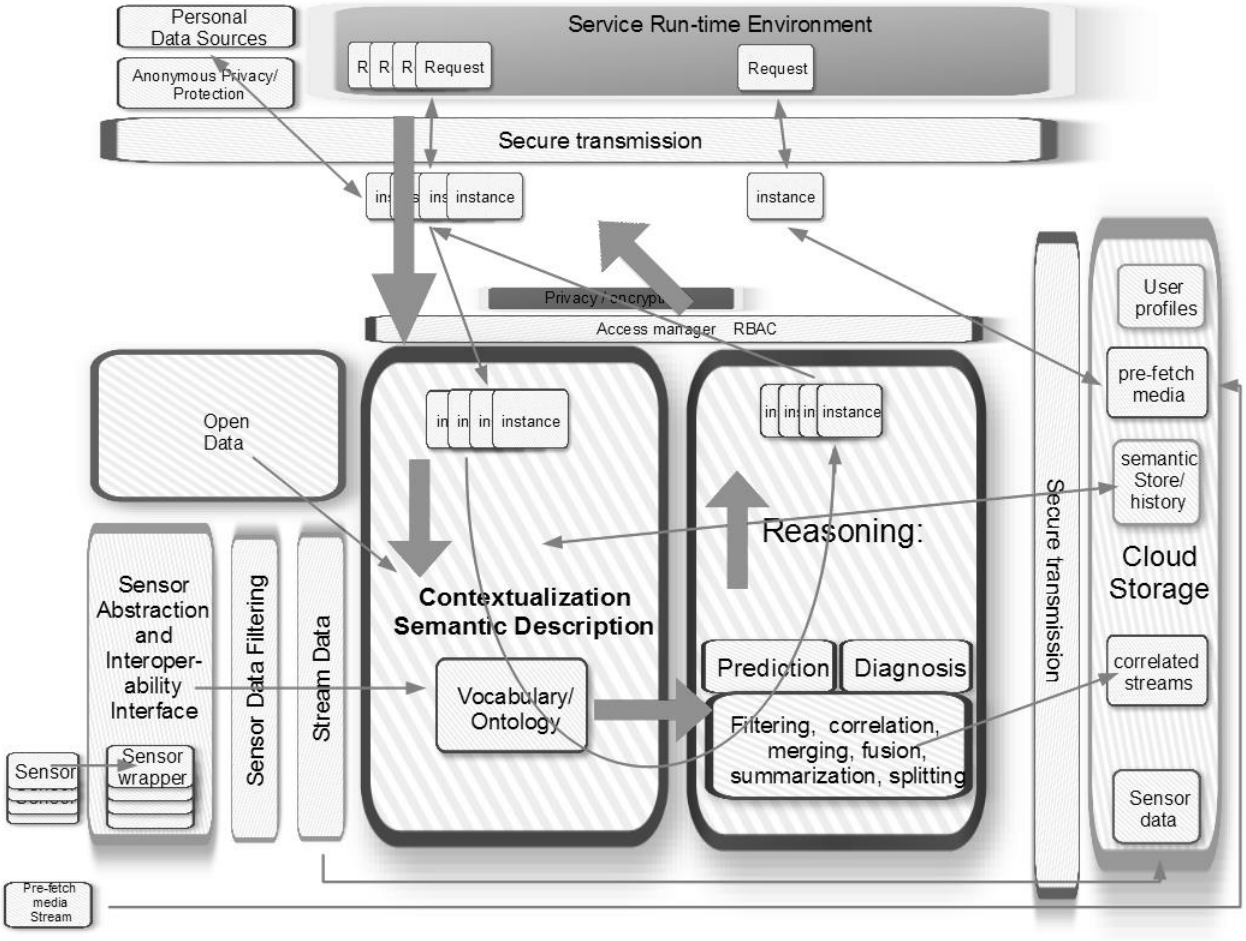


Figure 10: Work package 4 – Overall Architecture

### 4.8.1 Component Definition

This component comprises sub-components which combine and process incoming data from different sources:

- User-Centric data which can include PMA data, vehicle sensor data, user profile data, user mail calendar and other social media content.
- Open Data / Government Data.
- Sensor data which has been processed by the 'Sensor Abstraction and Interoperability Interface'.
- Historical (semantic and sensor) data retrieved from the cloud storage.

#### 4.8.1.1 External Data Pulling

This sub-component will manage data-pulling at specific frequencies from the external data sources for sensors and Open Data. The pull frequencies for specific data sources can be configured based on requesting services requirements. This is included in the section 4.9 Sensor Abstraction and Interoperability Interfaces description.

#### 4.8.1.2 Context

All of the externally pulled data including sensor, open data, user-profile and personal source data is processed by the 'Contextualization' component vocabulary and ontology producing a combined semantic structure. This combined semantic structure is stored in the cloud.

#### 4.8.1.3 Reasoning

The combined semantic structure tree is analysed in the 'Reasoning' component which does Prediction and Diagnosis. This includes such operations as filtering, correlation, merging, fusion, aggregation, summarisation and splitting of data. The outcome and relevant data of the Reasoning component is available to store in the cloud history storage for return via the service to the user.

#### 4.8.1.4 Media Pre-fetching

The facility to do media pre-fetching (T4.5) is included as shown for storage in the cloud.

#### 4.8.1.5 Privacy and Secure Transmission

This component uses secure access and ensures privacy by creating a different request instance for each service request. This also ensures role-based access to cloud storage history and sensor data. Encryption and media stream decompression (T4.5) can be enabled by the user profile settings.

#### 4.8.1.6 Sensor Data Filtering

This component is intended to filter sensor data so that only useful and necessary data is stored in the cloud storage and could be encompassed in section 4.9 (Sensor Wrapper).

### 4.8.2 Interaction of Component

The component requires interaction with the following components:

- Sensor Abstraction and Interoperability Interface (SAII) and its wrapper output data format.
- Sensor data provided from the PMA: It is assumed at this point that the sensor data format will be similar to the SAII data wrapper format above.
- User data to access calendar/social media content: The question of the mechanism to access the username and password for personal data sources needs to be defined as to whether it will be sent per request or stored as part of the user profile. Access to and data collection from these sources could also potentially be executed on the PMA reinforcing the users privacy.
- The Cloud storage infrastructure: The cloud storage APIs will be used to access user profile, semantic and sensor history information and to store updated semantic information and request results.
- The Service Runtime Environment: This component will interact with the Service Runtime Environment to receive user requests, notify the Service Runtime Environment of results and to set data pulling frequencies.

### 4.8.3 Possible Technical Foundations

The 'Contextualization' and 'Reasoning' sub-components will use common web interfaces to exchange data related to incoming service requests and outgoing results/outcomes. These sub-components will also use common web interface to request and store data within the Cloud storage Infrastructure. In addition the 'Contextualization' and Reasoning sub-components will use semantic technologies and store their representation in the Cloud storage.

APIs defined for the 'Sensor Abstraction and Interoperability Interface' will be called by the 'Data Processing' sub-component. In addition the relevant Open Data APIs and stream handling software will be used. The run-time instance, based on a particular service request, may call APIs related to the users' Personal data sources. Access to sensor, cloud and processed data will be managed using a role based mechanism. This will determine which history or other data the service and user have access to.

## 4.9 Sensor Abstraction and Interoperability Interfaces

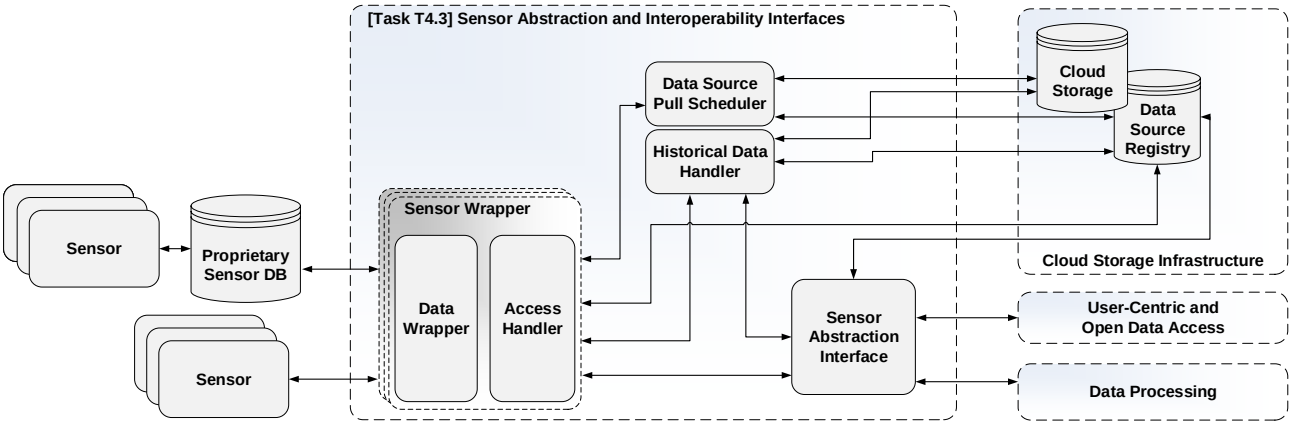


Figure 11: Structure of the Sensor Abstraction and Interoperability Interfaces

### 4.9.1 Component Definition

The sensor abstraction and interoperability component has the role to seamlessly integrate heterogeneous sources of sensor reading and providing corresponding sensor data to SIMPLI-CITY. The component has to be able to allow pulling of data from several sources and at the same moment be able to deal with event based information systems. In general there will be three basic types of data sources. External sensing systems that continuously measure the environment and provide interfaces to gather data on demand, event-based systems that transmit information once an event happens and proprietary databases that provide actual or historical data about specific sensor sources.

Each external data source will be registered to the system with a specific wrapper that transforms the external information into a common data format. Depending on the type of the external data source, the data flow is handled differently. In case of externally stored information, the respective data wrapper is associated to the data source and, each time information is requested, the component requests this information from the external source, transforms it into the common data format and returns this transformed information to the component that has requested it. For external information that is accessible on demand, the access scheme is the same. But in case of event-based information systems, e.g., sensor networks that offer an event bus, the component has to subscribe to these

and, each time an event occurs, it has to store this information for later requests. Also, a wrapper will be used here to transform the event data into the common data format.

To the internal components, a homogenous access scheme will be provided with a homogenous and common data structure. Thus, services can access information about all external data sources in the same way and receive the data in a common format. There will be an abstraction of the type of the requested sensors, thus information can be requested regardless the type of the used sensor, e.g., traffic flow in a specific location can be requested regardless whether the used sensor is infrared, ultrasonic, or an induction loop.

### 4.9.2 Interaction of the Component

The component will provide an interface and allow the provision and exchange of data. The component will receive data and deliver from/to the following data sources:

- Interaction with data processing components: Data will be provided in a common data format in a single data structure as single sensor readings to allow further data processing. The data processing component requests for sensor readings and receives the raw sensor data for all requested sensors, but in a common data format and structure.
- Interaction with cloud storage infrastructure: The component will store information that is only accessible at specific points in time, e.g., an event is triggered each time information changes in the cloud storage infrastructure. If such data is requested by the system, the component is able to retrieve this information from the cloud storage infrastructure at any point in time. This will allow on-demand access to all available information sources.

### 4.9.3 Possible Technical Foundations

The component will provide APIs to access sensor data and return requested information in a common data format. The component will be able to subscribe to event bus systems to gather event based sensor information.

The component will use APIs (e.g. based on REST) to store and request sensor readings in the cloud based storage infrastructure.

# 5 PMA Components

This section describes all components that are located at the mobile device or within the vehicle environment. All components together are forming the personal mobility assistant which is the main contact point between SIMPLI-CITY and the end user. End users will use this mobile device to access all SIMPLI-CITY functionality by making use of apps. Apps may be discovered and installed via the marketplace. They communicate with the server-side components of the project using the application runtime environment which will handle all communication via the service runtime environment.

## 5.1 Application Runtime Environment

### 5.1.1 Component Definition

The consortium strongly believes that the success of SIMPLI-CITY will depend on the acceptance of end users. This success will trigger demands for integrating SIMPLI-CITY into vehicles as it will provide vehicle manufacturers with a chance to deliver added value to their customers. A central element for user acceptance is the extendibility of SIMPLI-CITY. The project is not a closed information system but it rather allows developers to add own services and apps to it.

- *Apps* are targeting end users and will be available via the application marketplace allowing an installation of apps directly from inside the PMA. This concept is identical to successful marketplaces in the mobile world such as the Apple App Store or the Google Play market. Users will be able to find apps, e.g. by category or by keyword. They may then view details about an app such as the description and they may look at feedback from other users. Users may receive apps with just one click on their PMA and the PMA will download and install the app for making it usable immediately.
- In contrast to apps, *Services* are targeting other developers and will be available via the web based service marketplace as described in section 4.5. Apps will handle all communication with the user and they will make use of services from the server side to perform logical calculations, provide data or access external data sources. As such, services may be used by developers to create new apps as they build the foundation of each app. Services may be the base for multiple apps. For example, a car parking service may deliver free parking lots and may be used by many different apps at the same time.

#### 5.1.1.1 Lightweight App and runtime Concept

The very heart of SIMPLI-CITY is to provide a next generation service platform for building the road user information system of the future. The concept of SIMPLI-CITY foresees that apps are lightweight by design. This means that the main logic should be performed by services. Services will be located at the server side and are managed by the service runtime environment. They are consumed by apps running in the mobile device – the PMA.

The following examples show the relationship between apps and services:

- Example I: Parking Lots. An app that allows user to find free parking lots for the car will not host a database of parking lots at the local PMA storage. Instead, it will invoke a service at the server side, which returns free parking lots to the app based on the current GPS location of the user.
- Example II: Car-Pooling. Apps may publish route information and time schedules to a service if the user marks them as “public”. This allows the server side to find other users on the same route and to provide recommendations for car-pooling possibilities. Services will send notifications to their users informing them about car-pooling requests.
- Example III: Music App. An app that allows car drivers to listen to their favorite music would not host the music on the PMA. Instead it would make use of a service to get access to the music library of the user. All music would be streamed and therefore be provided by the server side with the exception of prefetching and caching media information as described in section 4.2.

Essentially, this concept means that the apps which are running on the PMA mainly handle the user interaction and the input and output of data while most of the logic is performed on the server side. This way, apps will be kept lightweight as they only contain a minimum of logic and data.

This concept has many advantages: For example, it allows the PMA to be equipped with a small amount of CPU power per app as all complex calculations would be performed by services. It also allows the PMA to only provide a limited amount of local storage space as all “big” data chunks are stored by the cloud storage. Finally, it allows SIMPLI-CITY to ensure that all data is up-to-date because its directly coming from the internet based service which can be updated at any time and which may directly access external data sources.

Apps are executed on the mobile device and the PMA will make use of the local operating system to execute them. The application runtime environment will act as an environment for those apps providing apps with common SIMPLI-CITY functionality. More precisely, the application runtime environment will provide the functionalities described in the following subsections.

#### 5.1.1.2 Coupling of Apps and the Multimodal UI

The application runtime environment of SIMPLI-CITY will act as the main interface for apps. It will build a bridge between the apps and the multimodal user interface described in section 5.4. This means that apps may fire events that will trigger the dialogue interface - if the dialogue interface deems it to be appropriate - to make a call to the application in order to retrieve certain data that it will read out to the user. Users will use the application runtime environment to launch specific features of the dialogue interface such as speech recognition or text-to-speech functionalities.

#### 5.1.1.3 Local App Registry and Inter-App Communication

While the app marketplace will allow users to discover, download and install apps, their management is performed by the application runtime environment. This means that the application runtime environment will keep a list (“registry”) of installed apps, similar to the service registry on the server side. This local app registry will allow other apps to detect which apps are installed and to exchange messages with each other. This functionality is

required as it allows apps to check if dependencies are fulfilled. For example, an app for parking lots may make use of a navigation app if such an app is installed.

For this purpose, the application runtime environment will provide a method for allowing apps to send (XML-) messages to each other using a local message bus. This allows a fast inter-app communication without exchanging messages via the server side.

#### 5.1.1.4 Server Side Communication

SIMPLI-CITY will handle all communication between apps and services via a unified interface of the application runtime environment. This means that apps will make use of an easy to use API from the application runtime environment to invoke services and to receive their response. The application runtime environment will then make a call to the service runtime environment which will deliver the message to the corresponding receiver and which will return the results to the application runtime environment. The application runtime environment will then forward all results to the original app.

This concept allows a central communication which makes it easier to ensure security and privacy issues and which allows a more efficient handling of the communication facilities as the network communication needs to be implemented only once at a central location.

#### 5.1.1.5 Local Data Management

As described in section 4.7, the application runtime environment will provide a simplistic data storage facility for local data. This local storage will be limited to storing simple key-value like data and will be very limited in terms of space. It may be used by apps to store local data that is either not meant to be stored on the cloud storage (e.g. credit card information) or that needs to be available very quickly and can therefore not be stored in the cloud for latency reasons. Data may also include some local user profile information related to the settings of the PMA itself.

#### 5.1.1.6 Error Handling

Finally, the application runtime environment will perform error handling for the PMA. This mainly includes two types of errors: Firstly, the application runtime environment will handle communication errors which may appear when the connection goes down. In this case, apps will receive a structured error message and may decide to discard the message or to request that the message is delivered by the application runtime environment at a later moment in time.

Secondly, application errors will be handled. The application runtime environment will be able to catch errors of apps and to avoid that a crashing app impacts the PMA as a whole. An app that crashes will be reported to the developer so that crashes may be analyzed based on crash reports (including stack traces).

#### 5.1.1.7 Push Notifications

SIMPLI-CITY will allow services with the option to inform users about events. For example, a service may send out a push notification to all users which are using an event planning app to inform them that an event has been cancelled.

This push notification system of SIMPLI-CITY will consist of two parts:

1. A server side interface for sending push notifications to an app. This interface is acting as a restful “push-service” on the server side. The service may be invoked by any of the SIMPLI-CITY services whenever they want to notify an app or a specific user. It will care about the delivery to the runtime environment of the PMA.

2. A receiver at the application runtime environment. The runtime environment of the PMA will filter the push message and deliver it to the corresponding app. For this purpose, the push notification will contain information about the target app. The application runtime environment will then check its local registry and use the inter-app communication to send a message to the app.

Please note: Usage of push notifications is optional. Developers may use this service but they may also choose to only provide normal pull mechanisms. If developers only want to support pull then they will provide a simple endpoint to invoke the service. If they want to support push then they can invoke the push service above whenever an event occurs.

### 5.1.2 Interaction of the Component

This component will provide an API for direct access. It will receive and deliver data from/to the following data sources:

- Apps. Obviously, the application runtime environment will interact with apps running on the PMA. This means that the application runtime environment will provide apps with methods to perform the tasks described above. This includes the usage of the local storage and of the inter-app communication.
- Application Marketplace. The application runtime environment will keep a registry of installed apps. For this purpose, the application runtime environment will interact with the application marketplace for installing and uninstalling apps on the PMA.
- Multimodal User Interface. SIMPLI-CITY provides text-to-speech and advanced speech recognition facilities for interacting with road users. As such, the application runtime environment will invoke the T6.1 and T6.2 components.
- Service Runtime Environment. The main task of the application runtime environment will be to handle the communication between apps and services. For this purpose, the application runtime environment will provide an API which apps may use to communicate with the server side. The application runtime environment will take those requests, wrap them into a communication envelope and deliver them to the service runtime environment, which will route the call to the corresponding service instance.
- Push Notification Service. The application runtime environment will provide the handling of push notifications. It will allow the push notification service to send messages targeting a specific app. After receiving the push notification, the application runtime environment will forward the message to the corresponding app.

### 5.1.3 Possible Technical Foundations

Technically, the application runtime environment will be realized as a native application inside the PMA. During the discussions of the consortium it has turned out that Android may probably be chosen as the base platform for SIMPLI-CITY although this needs to be confirmed during the functional and technical specification discussions.

In this case, the application runtime environment would be realized as a native Android app with the possibility to invoke its own application marketplace and to manage all SIMPLI-CITY apps within the java runtime environment.

The API provided by the application runtime environment to the apps will be provided as direct API methods in terms of delivering libraries for app developers which they can use to invoke the corresponding functionalities.

The communication between the application runtime environment and the service runtime environment will be performed using an open protocol such as JSON. This will allow an efficient and platform independent way of communicating between the PMA and the server side. The communication itself will always be performed with a secure HTTPS connection as discussed in section 7.

## 5.2 Application Marketplace

Please note: The application marketplace is interconnected with the service marketplace in terms of the review process and the developer communication side. There are many overlaps between both marketplaces. The overlapping parts are only briefly described in this section in order to avoid repetition. Because of this, it's recommended to read the service marketplace description in section 4.5 in conjunction with this section.

### 5.2.1 Component Definition

A core of SIMPLI-CITY is to provide users with an easy to use way of adding new functionalities to their mobile device – the PMA. These functionalities will be provided by apps. Apps may be installed by users directly from the PMA and they may be uninstalled at any time in case that a user doesn't need an app any longer. SIMPLI-CITY will provide a so called app marketplace which allows users to find new apps, e.g. by browsing the marketplace by category or by using a search. This concept of SIMPLI-CITY is comparable to well-known app markets of Apple (AppStore) and Google (Play Market).

In addition to the user side, the SIMPLI-CITY application marketplace will allow developers to add their apps to the marketplace. This will provide them with an easy to use way to publish apps and to generate income. Developers may specify the price of an app (which might even be 'free') and users may buy them for immediate use on their device.

### 5.2.2 Developer Side: Submission and Review

Developers may use the app marketplace for providing SIMPLI-CITY compliant apps. Those are apps that run inside the application runtime environment of SIMPLI-CITY and may be consumed by end users via the PMA.

For providing those apps, the application marketplace will provide a web application for developers. This web application will be usable from any modern web browser and allow developers to register and to login for managing the apps that they are using or providing.

This structure means that the application marketplace will essentially consist of two different elements: A web based interface for app developers helping them to publish, modify and update apps and a PMA version for end users helping them to discover and install apps to their device.

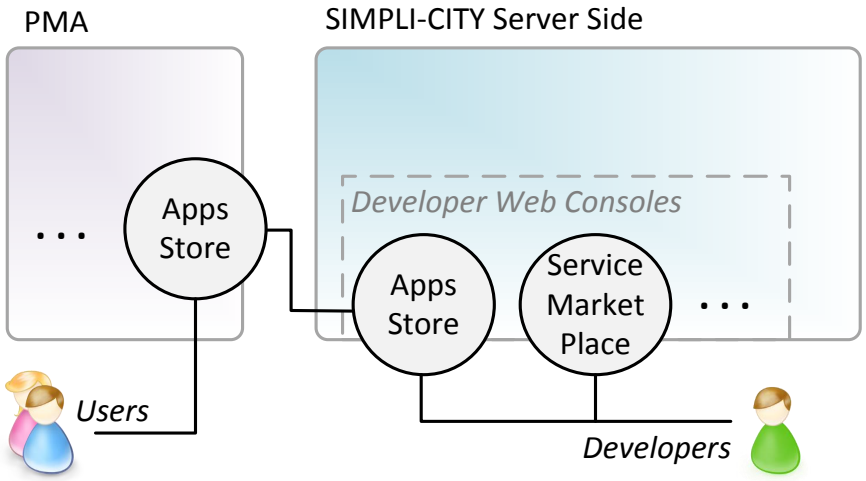


Figure 12 : Two areas of the application marketplace

Developers may submit their apps to the marketplace. Once an app has been submitted it will be marked as “under review” for developers. At this moment in time, the app will not be listed in the marketplace for consumers. Instead, the SIMPLI-CITY marketplace team will be notified about the new app submission and will perform a manual testing of it. For allowing this, developers will be able to specify testing notes to describe all requirements and dependencies for app reviewers such as license keys or user credentials. The goal of this review process is to ensure a high quality of SIMPLI-CITY and its apps.

Once, an app has been reviewed, it may be accepted or rejected and feedback will be provided to the app developer. Apps that have been rejected may be resubmitted by developers and apps that have been accepted will may be marked as “published” in order to be listed in the marketplace for installation.

App consumers may also provide feedback about an entry which is described in the figure below. However, consumers may also flag an app as inappropriate or damaged. In this case, the review process will be restarted allowing the SIMPLI-CITY team to take apps offline if necessary.

The following image shows the review process graphically:

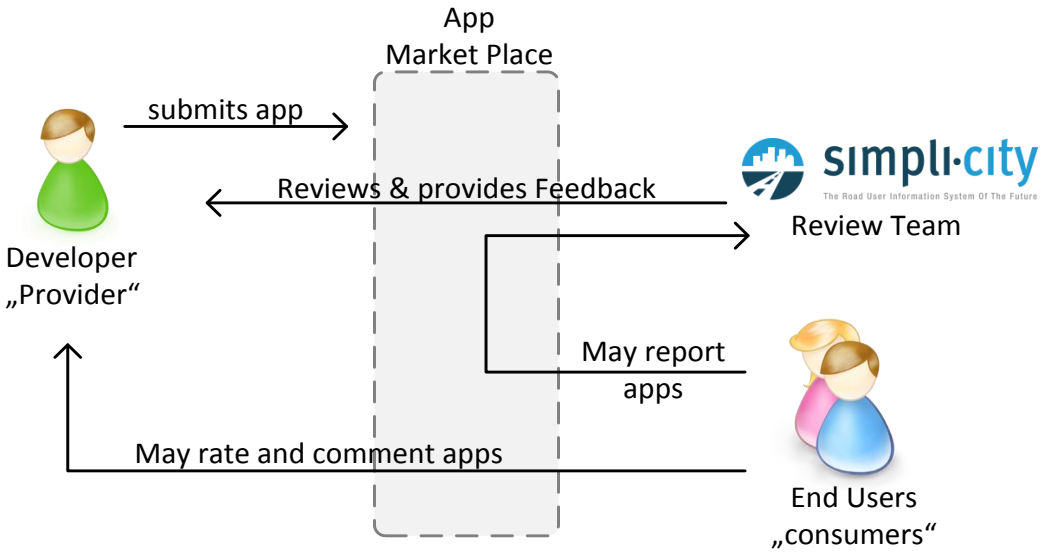


Figure 13 : The SIMPLI-CITY review process for apps

#### 5.2.2.1 Developer Side: App Preparation

The process described in the last section raises the question on how apps will be submitted in detail. For this purpose, app developers will be able to use a submission form on the web UI. This submission form will allow them to describe their apps and to upload all files that are needed for the deployment. Those files include a technical installation description in form of a technical manifest file and will be created in the application design studio (T6.4).

#### 5.2.2.2 Developer Side: App Lifecycle

App developers may upload new app versions at any time and they may put apps on hold by deactivating them. This process will then remove apps from the list of available apps in the marketplace. Whenever a new version of an app is uploaded, users will be notified in their app market on the PMA. The PMA will then allow them to download the new version of the app.

#### 5.2.2.3 Consumers: App usage

Once, apps have been published to the marketplace and marked as “published”, they are ready to be used. As such, the application marketplace will provide a list of apps which allows users to quickly find apps based on categories and also based on keyword searches.

Once an app has been found, consumers may click a simple install button to download and install the app. The PMA will then handle the full installation process and make the app available for immediate usage.

#### 5.2.2.4 Developers & Consumers: Developer Communication

Consumers may be able to rate app and to provide feedback in terms of comments. This will allow other developers to judge the quality and usefulness of an app based on other users’ comments. App marketplaces such as the Apple AppStore or the Google Play

Market have shown that this can lead to valuable information and will also give an indicator about the reliability of an app.

App developers will be able to use the web based version of the marketplace to view the ratings and comments of their apps and they will be able to respond to a comment, e.g. by helping a consumer with a specific problem.

#### 5.2.2.5 Developers & Consumers: Business Aspects

During the prototype implementation of SIMPLI-CITY apps will always be free of charge. However, once SIMPLI-CITY leaves the prototype stage, it will support commercial, and non-commercial apps. This impacts three different elements:

- App developers will be able to select a payment model for their apps. This might be ‘free to use’, ‘pay-per-use’, ‘monthly payment’ or ‘one-time payment’.
- App consumers will be able to purchase an app and to use their favourite payment method (PayPal, credit card, debit, etc.) to pay for an app.
- The application marketplace will ensure that apps can only be downloaded if the user has an active subscription to use it.

### 5.2.3 Interaction of the Component

The component will receive data and deliver from/to the following elements:

- Cloud Storage. The marketplace will make use of the cloud storage to store the setup files of apps in it. Additionally, descriptions and other marketplace material will be stored in the cloud storage component.
- Application Runtime Environment. The application marketplace will be closely connected to the application runtime environment and especially to the internal app registry. This includes the installation of apps and the information about commands that are supported by an app.

### 5.2.4 Possible Technical Foundations

From a technical perspective, the developer side of this component will be realized as a web application providing a web based UI for app developers. It will be targeting technical experts (developers) and will therefore be able to make use of domain specific languages and expressions. Nevertheless, the UI should be user friendly and easy to use. In order to maximize the usability of the web application, the marketplace should be usable from any modern web browser. Since tablets are becoming more and more important, it should be avoided to use outdated technologies such as Adobe Flash for the realization of the marketplace.

In addition to the developer UI, an app marketplace will be developed for end users. This will consist of a native app running on the PMA. It will therefore be implemented in the language of the operating system (e.g. Java in case of Android). The marketplace should have an easy to use UI as it is targeting non-experts.

## 5.3 Dialogue Interface

### 5.3.1 Component Definition

This component is the user interface layer of SIMPLI-CITY, taking user input in the form of utterances managing the need for further user input, and transforming them into application calls. The result of the application call is then fed back to the user, using speech. User input is collected using an Automatic Speech Recognition (ASR) subcomponent. The speech input is interpreted to dialogue "moves" (dialogue acts, such as *ask*, *answer*, *and request*) with some semantic content. The dialogue component can also be triggered by application activity, and can fetch input data from the application instead of asking the user if suitable.

The dialogue moves are forwarded to the central subcomponent Dialogue Move Engine (DME). The DME contains a description of the dialogue context, including information about recent utterances, recent questions, active apps etc., and makes decisions about the system's next move based on this information. The next move can be to "do nothing", to "ask a question", to "answer a question" or to "carry out an app call". Any resulting dialogue move output from the DME will be sent to a generation subcomponent, which will generate GUI contents and an utterance, respectively. The final step in the iteration is to do the utterance, which is done using a Text-To-Speech (TTS) component. A Turn Manager subcomponent is used to manage the right and opportunity to speak during the dialogue (to make an utterance in a dialogue is often referred to as a "turn"). The Turn Manager distributes the turn between the user and the system.

There can be several applications active in the DME at once, and applications can be dynamically loaded and unloaded. Parts of the application data (see Section 5.3.1.6) can be seen as a kind of "Manifest", describing the capabilities of the application to the DME, extending the parsing grammar (and possibly the recognition grammar) in order to make the newly added functionality available in the PMA. The ambiguity that arises when two or more applications can handle the same utterances are resolved primarily in the DME. There is also a possibility for a developer of an app to define in what geographical and temporal space the app is valid, thus keeping the ambiguity at manageable level.

#### 5.3.1.1 DME: Dialogue Capabilities

The dialogue move engine to be used in this component allows for a number of different dialogue behaviours, all founded in studies of human dialogue. Furthermore, these behaviours are all built into the DME, and do not need to be implemented in the individual SIMPLI-CITY applications. The following features are supported:

- Task switching: When performing one task, the system can answer questions about other issues or perform other tasks.
- Grounding: When unsure of user intention or contents of a certain utterance, the system communicates this to the user.
- Accommodation: The user is free to give partial information or give more information than the system asked for. For instance, the question "Do you prefer to go by bus or by train?" can be answered with "By train tomorrow night". Questions about what the user would like to do can be answered by giving other relevant information instead: For instance the question "How can I help you?" can be answered by saying "To Paris, by train".

#### 5.3.1.2 Interpretation and Generation

The interpreter and generator are built around the Grammatical Framework (GF) grammar formalism (www.grammaticalframework.org). The interpretation is done as a translation from natural language (e.g. English) into a semantic language, defining dialogue moves over a subset of first-order logic. Conversely, generation is done as a translation from the dialogue move language into natural language. The interpretation and generation grammars are defined jointly in order to coordinate the language use between the system utterances and the user, so that the system utterances describe to the user what kind of utterances are expected.

Currently, the interpretation grammar can only handle input from a grammar-based ASR. In order to being able to interpret data coming from dictation ASRs (such as the Android ASR, Nuance DragonMobile ASR etc.) we will build a new interpretation engine capable of handling more unrestricted utterances.

#### 5.3.1.3 TTS and ASR

The TTS and ASR engines are third party products. Android has a built-in dictation ASR as well as a TTS, available for application developers. Nuance has also made its VoCon recogniser and SDK available for Android at commercial conditions, making it possible to use both grammar based and dictation based recognisers in parallel. It is an open question which of the ASRs will be chosen for this project. There is a high-quality TTS engine built into Android, but there are also commercial options available. The TTS and ASR units will be wrapped into suitable modules for communicating with the rest of the PMA.

#### 5.3.1.4 Push-To-Talk or Push-To-Initiate

It is common to equip dialogue systems with a push-to-talk (PTT) button. The user, wanting the attention of the system, presses the PTT button in order to make the system listen to her or him. After the ASR recognised the utterance, the system releases the PTT button, and the recognition is processed by the system. The ASR will remain inactive until the next time that the PTT is pressed. The purpose is to avoid mistaking utterances directed to other persons and noises as utterances directed to the system.

An alternative is to use a Push-To-Initiate (PTI) button instead. The PTI button is pressed in order to initiate a dialogue, and the system and the user take turn speaking until the dialogue is finished. In a situation with a user under high workload and other people talking in the car, it may be better to let the user decide when he or she should speak by using a PTT solution. A PTI session means that the user must focus on the task during a longer period of time.

The choice of PTT or PTI is dependent on the expected sound environment of the system, as well as limitations in the platform.

#### 5.3.1.5 Turn Manager

The turn manager distributes the turn (the right and obligation to speak) between the user and the system. It takes into account the state of the TTS engine, the PTT state, the events coming from the application and service layers as well as the state of the DME. It

manages interruptions and distributes the appropriate pieces of information to the appropriate subcomponents.

#### 5.3.1.6 Application Data

The dialogue interface component needs access to data in order to make its decisions properly. The data consists of:

- Domain data: The application logic, describing what pieces of information are needed from the user and from the application in order to carry out some action or answer some questions.
- Application data: The data of the application, sensor data etc.
- Ontology data: What actions, questions, individuals etc. are available in the world of a particular application.
- Grammar data: What utterances are used to describe the world of an application.

### 5.3.2 Interaction of the Component

The component interacts with the following components:

- T6.3 Application Runtime Environment: The Dialogue component will interact with the Application Runtime Environment in order to get information about what Apps are available, to load App resources (including the four kinds of data mentioned above in section 5.3.1.6).
- T6.4 Application design studio: The application design studio will generate the data described in section 5.3.1.6.

### 5.3.3 Possible Technical Foundations

The dialogue interface will run on the actual device (smartphone, tablet, head unit, PC, etc.), with some components running on a dialogue management server.

#### 5.3.3.1 Server components

The server will host the DME, the interpretation and generation components and the Turn Manager.

#### 5.3.3.2 Client components

The ASR and TTS components will be wrappers around services, which may run on the device or in the cloud. The ASR will be contained in a wrapper. If an on-board ASR will be used (such as pocketSphinx, Nuance VoCon), the ASR will not require a network connection. If cloud-based ASR (Nuance DragonMobile, Googles ASR for Android) is used, a network connection will obviously be needed.

#### 5.3.3.3 Client/Server communication

The communication between the dialogue management server and the client components will take place using a JSON-based protocol over websockets.

## 5.4 Multimodal User Interface

### 5.4.1 Component Definition

The Multimodal User interface is based on the Dialogue interface described in section 5.3, but is extended with a Graphical User Interface. The Graphical User Interface is treated as another input/output modality, on the same level as voice input/output. This means that there is a component for interpreting Graphical User Interface events into dialogue moves just as there is one for interpreting user utterances (selecting a music track in the Graphical User Interface results in an *answer* move). In the same way the system moves are generated into both speech and graphics/text.

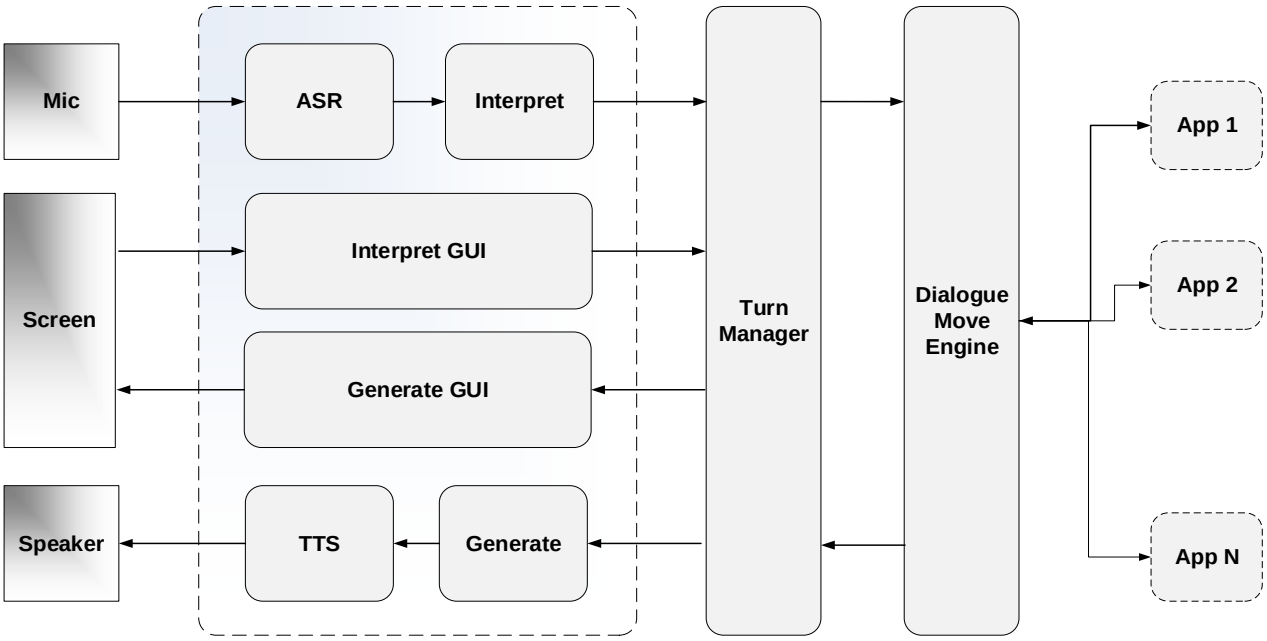


Figure 14: Multimodal User Interface

#### 5.4.1.1 DME: Multi-Modal Capabilities

The dialogue move engine supports multimodality. When the system asks a question it looks up possible answer alternatives to display to the user (according to a specification in the application description). Answer, feedback and report moves are accompanied by dialogue context information which makes it possible for the Graphical User Interface generation component to render appropriate views (headers, lists etc.) for the Graphical User Interface.

#### 5.4.1.2 Multimodal generation and interpretation

Output from the system needs to be displayed on the screen as well as spoken to the user in a coordinated manner. The DME will select dialogue moves to be realised. The dialogue moves will be realised in different ways by the Graphical User Interface and speech generation components, and the realisation will need to be coordinated by some coordination component. If the generation components and the GUI/ TTS modules run on different machines, there is a need for a communication protocol in order to communicate the realisation information and the coordination information between the realisation components.

Input must be collected from the user. When the Graphical User Interface content is generated, it will be represented with a proper graphic solution in such a way that the GUI component can provide the right interpretation of tapping, clicking and in other ways manipulating the screen content to the DME according to the available ergonomics and usability guidelines. GUI should also be compliant with automotive requirements for in car use, in tem of dimensions and complexity of the displayed information.

### 5.4.2 Interaction of the Component

The interaction of this component does not differ from the interaction of the Dialogue Interface. See section 5.3.2.

### 5.4.3 Possible Technical Foundations

The Graphical User Interface will be a client component, receiving XML or JSON descriptions of what the Graphical User Interface should render. Applications should also be allowed to get access to part of the display in order to display application-specific graphics, etc.

Such application-specific graphics must fulfil the basic contract described above: it should in some way be a rendering of the view description (the application must thus have the opportunity to access this information), and the manipulation of the graphics should result in a transfer of the appropriate information to the DME. As long as the contract is honoured, (parts of) the GUI can be provided by the application.

## 5.5 PMA-based Sensor Abstraction

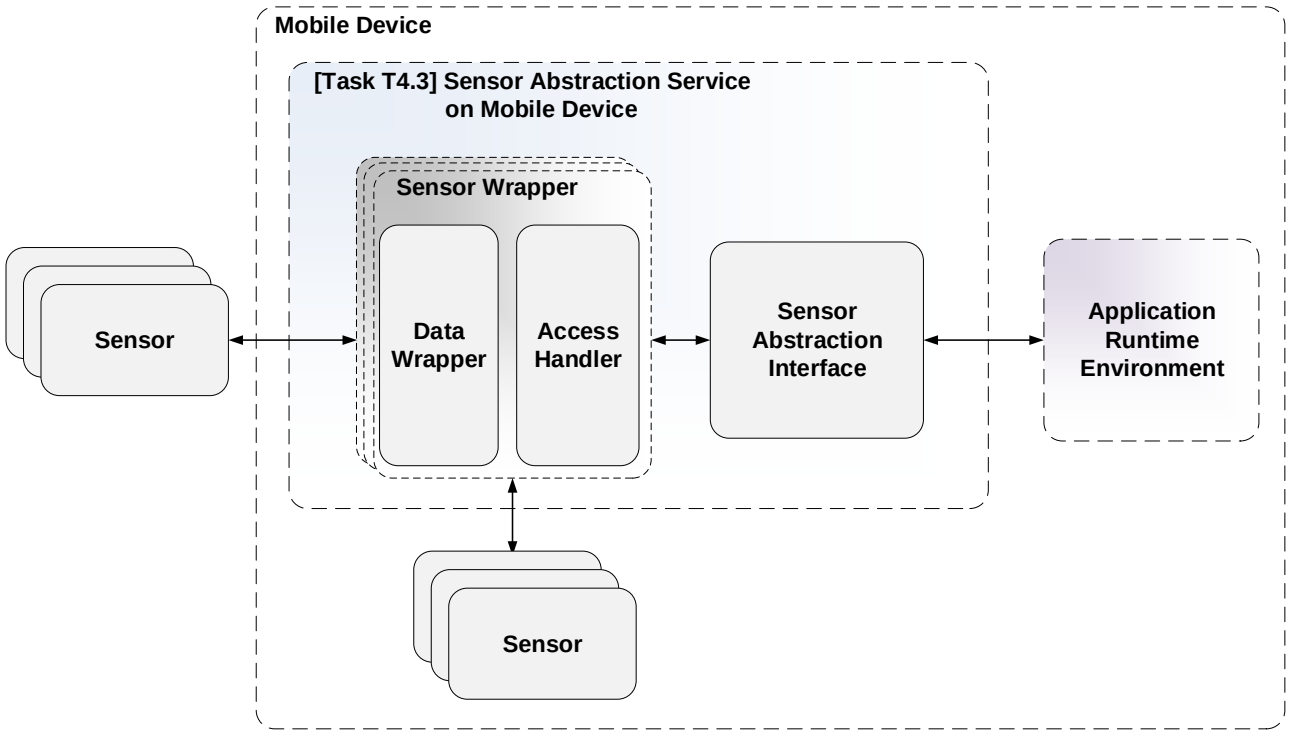


Figure 15: Structure of the Sensor Abstraction on the mobile device

### 5.5.1 Component Definitions

This component is the local extension to the sensor abstraction and interoperability interfaces described in Section4.9. In contrast to the previously mentioned component, this component is a service running locally on the mobile device. This service offers a consistent access to all available local sensors, both built-in in the mobile device itself and in the vehicle, as well. The available sensors are interfaced and the corresponding sensor readings are translated by respective wrappers into the common data format. Thus, sensor readings of the mobile device, e.g., acceleration data, and sensor readings of the vehicle, e.g., lambda oxygen sensor data, can be accessed in a common way and data format.

### 5.5.2 Interaction of the component

The component will provide an interface and allow the provision and exchange of local sensor data. The component will receive data and deliver from/to the following components:

- Application Runtime Environment: Data will be provided in a common data structure in a common data format as single sensor readings. The component will run as service layer on the Personal Mobility Device. Applications will directly request for local sensor readings and receive the raw sensor data for all requested sensors, but in a common data format and structure. Further processing will then be done on application level.
- Personal Mobility Device: The component will access the locally built-in sensors of the device to offer sensor readings to the other components.
- Local Sensors, e.g., car sensors: The component will provide access to the car sensors to offer sensor readings to the other components.

### 5.5.3 Possible Technical Foundations

The component will provide APIs to access sensor data and return requested information in a common data format.

# 6 Developer Support

The consortium strongly believes that SIMPLI-CITY will only be successful if it offers added value for end users. This added value mainly comes from a range of different functionalities that SIMPLI-CITY will provide and which will be extendable via new apps and services. The provision of new apps and services via the SIMPLI-CITY marketplaces will lead to an ecosystem which allows developers to generate income from app and service sales and which allows end users to benefit from new developments in a continuous way. This aspect is comparable to the success of the Apple App Store or the Google Play market. In order to achieve it, SIMPLI-CITY has to provide support for developers in terms of tools, APIs, examples and guidelines. This section describes those elements which are purely targeting developers of apps and services.

## 6.1 Application Design Studio

### 6.1.1 Component Definition

The Application Design Studio offers an appropriate step-by-step procedure to app development, assisting the app developer during the entire development process. It delivers everything that a developer needs to prepare an app for the usage within SIMPLI-CITY.

#### 6.1.1.1 Guidelines, Best Practices and HowTos

Among other things, the studio will provide a set of guidelines and HowTos to developers. This will define guides on how apps should be developed. Descriptions will be provided helping developers to ensure that their app is compliant to the multimodal UI and for ensuring a holistic look & feel for the best possible user experience.

Additionally, those documents will describe the typical development process of SIMPLI-CITY apps and provide developers with a handbook on how to develop apps in a step-by-step manner.

#### 6.1.1.2 Examples and Open Source Apps

In addition to the documents above, the studio will also come with a set of examples. Those examples will be source code snippets to show developers how to solve specific tasks in SIMPLI-CITY. Additionally, example apps will be provided which will be released under a non-infecting open source license. Those apps may be used as a kick-starter for developing new SIMPLI-CITY apps for the PMA.

#### 6.1.1.3 Manifest Creation

The studio includes support for the definition of an app manifest file. A manifest file is used to specify the runtime environment requirements and to specify properties of apps such as the app icon or the dependencies to other apps. The resulting manifest file will be included along with the app when it is submitted.

Additionally, the manifest will define commands that an app can react to. This list of commands will be used by the T6.1 and T6.2 components and also by the T6.3 Mobile Application Runtime Environment in order to launch an app. The manifest will therefore be

the main step in the app design and deployment process. The commands listed in it will form the connection between the user and the application runtime environment. For example, a statement like “find the next parking space” will be used to later allow the application runtime environment to find out which application to launch as soon as the user has pressed the “Speak With The PMA” (SWP) button and has spoken those commands.

#### 6.1.1.4 Bundling

Once a manifest file has been created, the studio will support the compilation of an app bundle. An app bundle contains the app itself and manifest file and also all other files that are needed for an app (images, etc.). It may be compared to a JAR file in Java or to a WAR file in web developments. The bundle will be used as an input to the application marketplace and will essentially be a deployable version of the SIMPLI-CITY app.

### 6.1.2 Interaction of the Component

The component will receive data and deliver from/to the Application Marketplace. The component will indirectly interact with the application marketplace because it will create a manifest file which users will bundle together with their app when submitting an app to the marketplace. As such, this component and the Application Marketplace component need to have the same understanding of the manifest file.

### 6.1.3 Possible Technical Foundations

From a technical viewpoint, the application design studio will provide a set of documents, guidelines and deliverables to app developers. Those documents will be delivered as normal ZIP files and may be downloaded by any app developers.

Additionally, the task will deliver a development environment for creating new apps. This environment will be based on the popular Eclipse platform which allow the reuse of many important functionalities such as syntax highlighting. The studio will extend the eclipse platform by integrating the developer documentation and by covering the manifest description as outlined above. Additionally it will provide direct links to the application marketplace for developers.

Finally, the design studio will contain the possibility to view crash reports in order to analyze the current stability of an app.

Please note: The selection of Eclipse as a base platform is preliminary and based on the assumption that Android will be used as a core for the PMA. If another platform is chosen in the functional or technical specifications, then Eclipse might be replaced by a different base.

## 6.2 Service Development API

### 6.2.1 Component Definition

The Service Development API is a component aimed at third party developers that allows them to create and configure their own services on the SIMPLI-CITY platform. These services will be later used within end-user applications and will be responsible for the provision of the external data needed during their execution.

There are three different types of services:

- Data services. These services act as wrappers for external data sources.
- Backend services. These services provide more complex functionality like the combination of different data services and the inclusion of context-related information.
- External services. These services are hosted by an external third party, and integrated in SIMPLI-CITY by means of proxies.

Both backend and external services can be consumed by end-user apps.

#### 6.2.1.1 Developer interfaces

The API will be provided as a restful interface by means of web services, where all the functionalities needed by developers in order to manage services will be covered. In the Functional and Technical Specifications (D3.2.1 and D3.2.2) it is to be defined if the so-called WS-* stack or REST-based services will be used for the description and execution of services.

Moreover, some functionality can also be provided by means of a web application tool. It will be based on a front-end web interface formed by easy menus that will permit to create and manage services. This web application will be usable from any modern web browser.

#### 6.2.1.2 Data services management

Data services refer to the access to external data sources. Data sources will be provided by external data providers, and integrated into the system by means of the different components implemented in WP4, following a Mobility-related Data as a Service approach.

The data service creation process includes the definition of the following different parameters associated with the service. Some of them may include:

- Name and description of the service.
- URL of the data source and the format of the data.
- A Service Level Agreement, defining the QoS requirements that the service has to fulfil.
- Push or pull mode of accessing the data. The push mode means that the external data provider will post information when there is a relevant update of the data, whereas pull mode means that the system is responsible for requesting data to the external data source.
- Frequency of update of the data source. In the pull mode this parameter is used to specify the frequency of requests to the data source.
- Storage of information. In order to specify if the cloud database should store all the historical information of the data sources or only the updated values.

Once the service is created it will be registered in the Service Registry.

The Service Development API will also provide the following data service management features:

- Data service modification. The service developer will be able to modify and update the information of the service and the data source configuration, and also to add new versions of the service. It also permits to delete a service from the registry.
- Data service discovery. This functionality will permit to search for previously created data services.
- Data service visualization. The developer will be able to visualize the information of a service published, including the parameters describing the service and the current values of the data source.
- Data service SLA monitoring. Permits to monitor the QoS aspects of the SLA associated to the service.

#### 6.2.1.3 Backend services management

Backend services are the ones consumed by end-user applications. They can be formed by a composition of different data services and include context-related information so that the output of the service execution varies depending on this context, e.g. the location of the user.

Backend services management functionalities include:

- Backend service creation. It allows the definition of the service parameters, the configuration of the different data services that compose the service, and the inclusion of context-related information.
- Backend service modification. The service developer will be able to modify and delete a previously created service.
- Backend service discovery. In order to search for previously created backend services.

### 6.2.2 External services management

The Service Development will also permit the management of external services, which are the ones that are hosted by third parties. The management of these services is similar to the ones created using the SIMPLI-CITY platform from the service developer point of view. As explained in section 4.1.1.5, external services will be supported in the Service Runtime Environment by means of proxy services, which will permit to technically treat services as if they were originally published within the SIMPLI-CITY platform. The Service Development API will provide functionalities to service developers in order to create and manage external services in a similar way as the backend services.

#### 6.2.2.1 Documentation

The Service Development API will assist developers during the entire process of service creation. It will provide tutorials and guides explaining step by step how to use the API and the web interface in order to create and configure a new data or backend service.

It will also provide some examples of service configuration, as the configuration of the data source to be used by the data service or the composition of data services when creating a backend service.

### 6.2.3 Interaction of the Component

The Service Development API interacts with the following components:

- Service Registry. The Service Development API interacts with the Service Registry in order to store the information of a new service during the creation process, to modify or update the parameters of a service, and to search for previously created services.
- Service Runtime Environment. Service developers will be able to execute a service in order to visualize the current values of the data service.
- Service Monitoring. In order to monitor the QoS aspects of a service according to its SLA.

### 6.2.4 Possible Technical Foundations

The Service Development API will be implemented using e.g. JAVA J2EE technologies, and the front-end web services technologies. In the Functional and Technical Specifications it is to be defined if the so-called WS-* stack or REST-based services will be used for the description and execution of services.

The web application tool will be provided as a user friendly and easy to use web based UI. The frontend interface will be developed using standard such as HTML and CSS.

The different interfaces of the API will be provided using standard restful services over HTTP 1.1. The services will be provided using standard XML or JSON data formats.

# 7 Security and Privacy Aspects

When handling sensitive data from the user, it should be ensured that this data cannot be seen by third party applications or services without the users' explicit consent. This will be achieved by providing metadata every time user data is communicated, in which the sensitivity and therefore the required permissions are stored. When querying external data sources, handing the data over to services or storing the data in the cloud, the user must have consented to have personal data used for the specific app or service. When the data is processed, the resulting data should be marked sensitive if it contains sensitive data from the input.

The metadata shall be stored in the data as an extra key, e.g. in the root of a JSON object. For the internal security and privacy measures of the SIMPLI-CITY platform to work, all network traffic will be done over secure HTTPS connections. Additionally, all applications (on the PMA) and services (server side) will be digitally signed and the authenticity of these signatures shall be verified at run- or install time.

## 7.1 Vehicle & PMA

### 7.1.1 Local Storage

Security of the local storage will be handled by the operating system. Given that the user has free access to the PMA and the device is not required to be tamper proof, there are no security measures that can guarantee the confidentiality and integrity of the data if the device has been tampered with, save for encrypting the entirety of the data with a user supplied encryption key which would be too inconvenient.

With the same reasoning, storage of any credentials should be handled by using the OS supplied password storage. These may include credentials for applications installed with the Apps Store and the bare necessary authentication data needed to make purchases in the Apps Store.

### 7.1.2 Application Runtime Environment

To protect the user from installing malicious apps in SIMPLI-CITY that circumvent or otherwise render useless the internal security and privacy measures, the Application Runtime Environment should verify the digital signature of each app that is installed by the Apps Store. For this purpose, a key pair shall be generated of which the public key will be stored on the PMA. When the app is published, it will be digitally signed by hashing it and encrypting the resulting hash with the private key. The resulting signature is stored on the PMA when installing the app. When starting the PMA, the signatures are verified by again generating a hash for each app and comparing the result with the decrypted signature.

### 7.1.3 Apps Store

The PMA Apps Store will use a secure connection with the server side Apps Store, and store the necessary data for the ARE to verify the applications at runtime. Alternatively, the Apps Store could do the verification at install time.

When installing an app, the user shall be presented with a list of the required permissions of the app, including which (types of) services need access to the data. Here, a distinction

could be made between coarse and exact data, for example a rough location to find nearby parking lots, or an exact location to find a parking place while on the lot. This list will be stored in the app itself, and when providing the app with data, this list will be used to verify that the app is only allowed access to data that the user has consented to. When an app needs to query a service, the data provided in the query is checked against this list to again ensure that the service may use the data.

## 7.2 Server-Side

### 7.2.1 Service Runtime Environment

The principles of a white-list of permissions to leverage a users' sensitive data also applies to the Service Runtime Environment. Services that are queried for data may only receive sensitive user data if the user has agreed that this particular service or type of service may be provided with it. Here, the same basic architecture will be applied as on the PMA: Services are digitally signed during publishing and the signatures are verified on run- or install time. The Service Registry will provide these signatures and the lists of permissions that the services require to function, and any app that uses these services will automatically require these permissions as well.

When combining user data with open or third party data sources or otherwise processing user data, the resulting data (that may be stored or used in queries) should be tagged with the resulting sensitivity of the data. For example, when user centric data such as a calendar is used to augment data after a query, the resulting data set should be tagged as containing calendar data, whether coarse or exact. The same principle applies when a service is personalized based on context, where the result of a query to find facilities in a given area might have to be tagged as containing the coarse location of a user.

### 7.2.2 Cloud Based Information Structure

When data is stored in or retrieved from the cloud, any and all meta-tags concerning sensitivity and permissions should be respected. The responsibility for this lies with the components that store the data into the cloud.

The security of the data in the cloud is provided by the hosting environment, as long as only authenticated connections from SIMPLI-CITY components are allowed. The PMA nor external data sources shall have direct access to the cloud storage.

## 7.3 External Data Sources

External data sources will most likely be queried mainly through services installed in the Service Runtime Environment. The Service Runtime Environment is responsible for not leaking sensitive data to services and therefore to external data sources that are not allowed to use it. For data sources that are queried periodically, no user data will be supplied, nor is it necessary.

# 8 Potential Risks and Limitations

The SIMPLI-CITY project is aiming to create a complex, yet flexible system with a lot of different components and requirements. The architecture has carefully been designed in order to match with those requirements and in order to provide a solid base for the upcoming deliverables and the prototype implementations. However, as a matter of fact, all architectures have certain risks which are derived from their structures and from the way that the components are designed or connected.

Since SIMPLI-CITY is a complex project with many different aspects, the architecture has been widely discussed and analysed by the consortium. Nevertheless, there are certain risks which will be highlighted in this section. The consortium is fully aware of those risks and will carefully monitor them in the course of WP3 and of course during WP1. The following sections show those risks and they also describe how the consortium handles them during the project.

## 8.1 Risk I: Connectivity

SIMPLI-CITY is an innovative and forward-looking project which essentially builds the next generation service based platform for road user information systems. The PMA of SIMPLI-CITY allows users to install new apps into the device by downloading it from the marketplace. Apps usually consist of two parts: A local app logic which runs on the mobile device and a set of services which are invoked from the service runtime environment via the application runtime environment. This service based approach represents the heart of the SIMPLI-CITY approach and its architecture. It makes the SIMPLI-CITY approach flexible and scalable and allows a handling of all critical logic elements on the service side. However, in reality this approach requires an active communication between the PMA and the service runtime environment. As such, the architecture assumes an always-on connection of the PMA to the internet.

This assumption may be seen as a risk of the architecture by design because as of today, the connections are not always available within vehicles, e.g. when driving on the street with a car. However, the SIMPLI-CITY team sees SIMPLI-CITY as a road user information system of the future meaning that the project itself is dedicated towards the future situation. The project will release prototypes during the course of the project and will probably bring a first version to the market around 2-3 years after project ending, i.e. around 2017. Because of that, the project team strongly believes that connectivity will not be a problem by the time of market introduction. The project team even believes that technologies such as LTE will be fully deployed and available by this time.

## 8.2 Risk II: Scalability

SIMPLI-CITY has been designed with scalability in mind. However, it should be noted that SIMPLI-CITY is not a project that is fully dedicated towards scalability. As such, the project architecture was not chosen to be optimal for scalability aspects even though it addresses it up to a certain extend. This means that SIMPLI-CITY does not foresee a mechanism for providing message queuing as it would be provided by an Enterprise Service Bus.

Nevertheless, SIMPLI-CITY provides scalable components for all core elements. For example, the storage system of SIMPLI-CITY is fully based on a cloud foundation and will therefore project an elastic data storage solution that will grow with the demands of

SIMPLI-CITY. The service based architecture will also allow a strong scalability of the system as services may easily be replicated and duplicated. As such, all critical systems of SIMPLI-CITY will be scalable by design.

## 8.3 Risk III: Legal Aspects

The architecture of SIMPLI-CITY has been chosen based on technical and user requirements as identified in D2.3. Because of the nature of the project, the architecture has been driven by technical / ICT partners and not by legal aspects. This obviously means that the architecture does not keep in mind any legal aspects. For example, it may be possible that some services are not allowed to be used in certain countries. For example, some countries may not allow the usage of a speed control warning app, which might be offered in the app marketplace of SIMPLI-CITY. At the moment, it is not foreseen that SIMPLI-CITY would be able to handle those legal differences – at least not from an architectural / component viewpoint.

# 9 Conclusion

Within this deliverable, the global architecture of SIMPLI-CITY has been described in a high-level way. This global architecture has been used to define components and their interconnection. It is based on the main components defined in the Description of Work but it has taken the current development of the project and especially the findings of D2.1 and D2.3 to define a set of components at a much more detailed level than it has been possible in the Description of Work. As such, some components have been detailed or even split into several sub-components.

In addition to the component breakdown and description, each component has outlined its interaction with other components, apps and services. For this purpose, a description of the interaction has been given completed by a short section of fundamental technology choices.

In the next step, details will now have to be defined in the functional specification (D3.2.1), especially outlining detailed functionalities e.g. in case of system failures or in terms of defining the functional scope of each component in an even more detailed way than it was possible in this document. Afterwards, the technical implementation will have to be specified in D3.2.2.

Finally, it should be noted that this document is not to be considered static. Of course the global architectural choices are not expected to change during the course of the project. However, some of the details outlined in the component descriptions may change as new technologies are developed. In those cases, if any of the following deliverables will force a revisit of the architecture the changes will be recorded in an extra deliverable or in the SIMPLI-CITY wiki, depending on the degree of changes.