AppsGate

“Applications Gateway”

D2.4a Deliverable Report

Intermediate HMI Middleware

Date

August 28, 2014

Abstract

This document describes the first version of the technical specifications of the middleware developed in Task T2.4. The purpose of this middleware is to facilitate the development of novel user-centered services and gesture-based interaction techniques for the AppsGate STB. This middleware is comprised of the *HMI Middleware*, a service-oriented infrastructure that runs on top of OSGi developed by UJF/LIG, and the *iisu* middleware developed by SoftKinetic for multi-location gesture control.

Keywords

Software architecture, specifications, middleware, API, gesture-based User interface, resilience, run-time adaptation.

Document History

Version	Date	Author	Changes
V0.1	23/10/2013	Joelle Coutaz	Initial draft (document template)
V0.2	22/11/2013	Ilse Ravyse	Initial input for the IISU middleware
V0.6	30/11/2013	J. Coutaz, T. Flury	Input for the HMI middleware
V0.9	20/2/2014	I. Ravyse, B. Lorent	Reviewed input for the IISU middleware

Contributors

Name	Email	Organization
Joëlle Coutaz, Thibaud Flury, Alexandre Demeure, Patrick Reignier, German Vega, Jacky Estublier, Jander Nascimento	Nom.prenom@imag.fr	UJF/LIG
Cédric Gérard, Rémy Dautriche, Kouzma Pethoukov, Jean-René Courtois	Nom.prenom@inria.fr	UJF/LIG
Ilse Ravyse, Bertrand Lorent, Elhosain El Fahsi, Kevin Simons, Marius Cetateanu, Laurent Guigues	ira@softkinetic.com	SoftKinetic Software

Table of Contents

List of Figures

Executive Summary

This deliverable describes the first version of the technical specifications of the middleware developed in Task T2.4. The purpose of this middleware is to facilitate the development of novel and resilient user-centered services and gesture-based interaction techniques for the **AppsGate STB** and ecosystem. This middleware is comprised of the ***HMI Middleware***, a service-oriented infrastructure that runs on top of OSGi developed by UJF/LIG, and the ***iisu*** middleware developed by SoftKinetic Software enhanced for multi-location gesture control.

1.Introduction

This deliverable describes the first version of the technical specifications of the middleware developed and/or enhanced significantly in Task T2.4. The purpose of this middleware is to facilitate the development of novel and resilient user-centered services and gesture-based interaction techniques for the **AppsGate STB** and ecosystem. This middleware is comprised of the ***HMI Middleware***, a service-oriented infrastructure that runs on top of OSGi developed by UJF/LIG, and the ***iisu*** middleware developed by SoftKinetic Software for multi-location gesture control. The **HMI Middleware** is described in Section 2 whereas the **enhancements to iisu** are presented in Section **Erreur ! Source du renvoi introuvable.**. Detailed specifications are provided in a suite of annexes: architectural technological integration in Annex 1, specifications of the components of the HMI middleware (Annex 2), specifications of the SPOK AppsGate EUDE (Annex 3), while the API documentation of iisu can be found in D1.3.

2. The HMI Middleware

As shown in Figure 1, the **HMI Middleware** runs on top of the **Android** Dalvik Java Virtual Machine and Android Core libraries. Http is used as the communication protocol between the HMI Middleware and the native Android Applications that run on the AppsGate STB. An End-User Development Environment (EUDE) called SPOK (Simple PrOgramming Kit) has been developed on top of the HMI Middleware to allow end-users (i.e. not developers) to define the behaviour of their AppsGate Smart Home (cf. Deliverable D1.4). The component “End-User GUI” denotes the UI elements that run on interaction devices of the AppsGate ecosystem such as tablets and smartphones. The component “STB-GUI” denotes the UI elements that run on TVs to access media framework-related services.

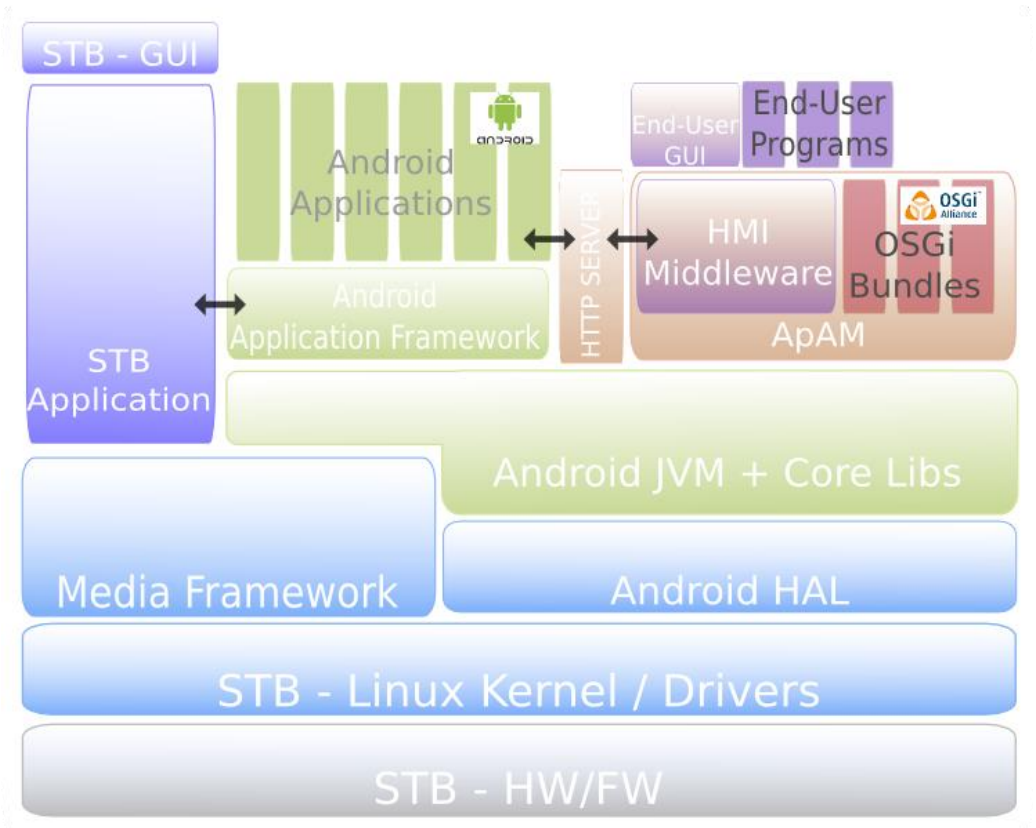


Figure 1. The HMI middleware within the global architecture of the AppsGate STB.

The HMI Middleware is implemented as a mix of **OSGi** and **ApAM** components where ApAM is in turn a middleware that runs on top of OSGi. The principles of this baseline are introduced below in Section 2.1. We then present the overall functional decomposition of the **HMI middleware**, followed by a more detailed description in Sections 2.2 and 2.3.

2.1. Foundations: OSGi and ApAM

2.1.1. OSGi (Open Services Gateway Initiative)

OSGi provides a general-purpose, secure, and managed Java framework that supports the deployment of extensible and downloadable software components known as bundles. OSGi-compliant devices can download and install **OSGi bundles**, and remove them when they are no longer required. The OSGi framework manages the installation and update of bundles in a dynamic and scalable fashion. To achieve this, the framework manages the dependencies between bundles and services. It provides the bundle developer with the resources necessary to take advantage of Java platform independence and dynamic code-loading capability in order to easily develop services for small memory devices that can be deployed at a large scale [OSGI 2012].

The **OSGi framework** is structured into the following functional layers:

- *Bundles*. Bundles are jar components with extra manifest headers (i.e. kind of meta-descriptions including a human readable name, a unique identifier, a version number, the class name to be invoked once the bundle is activated, the java packages that the bundle provides to, and requires from, the external world).
- *Services*.The services layer connects bundles dynamically by offering a publish-find-bind model for plain old Java Interfaces (POJI) or Plain Old Java Objects POJO.
- *Service Registry*. The service registry layer provides API for service management (Service Registration, Service Tracker and Service Reference).
- *Life-Cycle*. The Life-cycle layer provides API for the life cycle management of bundles (install, start, stop, update, and uninstall).
- *Modules*. This layer defines encapsulation and declaration of dependencies (how a bundle imports and exports code).
- *Security*. This cross layer handles the security aspects by limiting bundle functionality to pre-defined capabilities such as access to critical resources.

An AppsGate ecosystem such as a smart home, is a highly distributed dynamic environment. This means that services and processors appear and disappear frequently across a possibly unreliable set of networks and network protocols. Service continuity must be ensured as well as the management of the concurrent execution of numerous applications. OSGi is not sufficient to support all these requirements at runtime. This is where ApAM comes into play.

2.1.2. ApAM (Application Abstract Machine)

A service is dynamic if it can appear and disappear, without notice, at runtime. A dynamic application is built from dynamic services. ApAM is an extensible service platform intended to support the execution of concurrent dynamic applications [Estublier et al. 2012]. ApAM aims at simplifying the design, development and execution of applications that must guarantee service continuity in a moving and unpredictable context, and in the presence of other, potentially competing, applications. ApAM includes the ApAM Core and a number of managers (ApAM plugins) that are designed to collectively support the execution of concurrent dynamic applications.

ApAM machine (or ApAM core)

The **ApAM machine** is based on a component-service metamodel (called **ApAM metamodel**) that supports the definition, execution and management of component-based and service-based applications.

As shown in Figure 2, **Component** is the central concept of the ApAM metamodel. Components can be of three types: specification, implementation and instance.

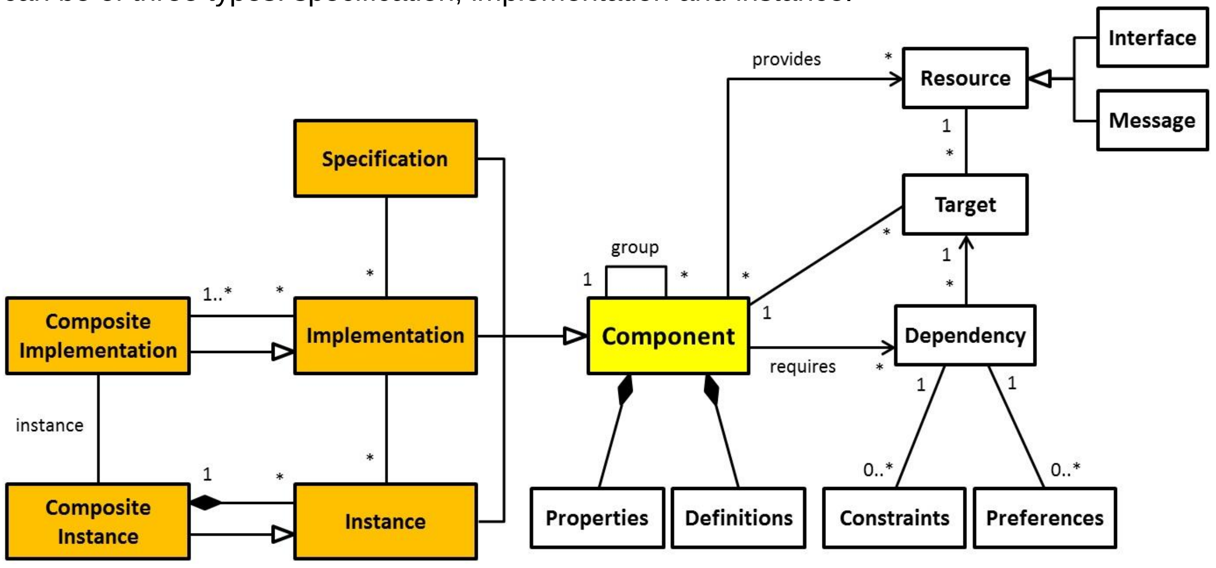


Figure 2.The component ApAM metamodel.

- *Specification*. A specification is a first class object that defines a set of provided and required resources (in the Java sense). Compositions are designed and developed in terms of specifications.

- *Implementation*. An implementation is related to one and only one specification through the "implements" relationship. An implementation is an executable entity (in Java) that implements all the resources defined in its associated specification, and that requires at least the resources required by its associated specification. In practice, an implementation must define a class that implements (in the Java sense) the interfaces of its specification.

- *Instance*. An instance is related to one and only one implementation through the "instanceOf" relationship. An instance is a runtime entity represented in the runtime platform (in our case, OSGi) as a set of Java objects where each object is an instance (in the Java sense) of its associated main class implementation. In the underlying runtime service platform (in our case, OSGi), an instance can be seen as a set of services, where one service corresponds to one resource of the specification. In ApAM, an instance is a Java object.

- *Composite Implementation* (or composite type). A composite implementation, also called composite type, is a special case of implementation. As such, it implements a specification, but in contrast to an atomic implementation, a composite type does not define a main class. A composite type contains implementations and defines the rules that govern their composition and visibility.

- *Composite Instance*. A composite instance, also called composite, is a special case of instance. As such, it is an instance of a composite type. A composite instance contains instances (atomic or composite), including one instance at least of the main implementation of its composite type.

- *Component Groups*. Components are related by the "group-members" relationship. A group specification is a group whose members are implementations, and a group implementation is a group whose members are instances. A group-members relationship establishes a *de facto* inheritance between the group and its members. More precisely, the characteristics of a group (its properties, its provided and required resources) are automatically inherited by its members, as in a class-instance relationship.

- *Component life cycle*. At runtime, an ApAM component has a single state: either it is “existing” (it is therefore, available and active), or non-existing.

ApAM Component Description Model

A component description contains the (meta-)information needed for the java compiler to produce the "right" executable component for the target execution platform (in our case an OSGi bundle), and for the runtime execution platform to enforce its "right" behavior.

ApAM State Model

The **ApAM machine** represents the current state of the supported applications as a model, called ASM (ApAM State Model). ASM is compliant with the ApAM metamodel. The ApAM machine provides an API for navigating and managing the ASM as well as mechanisms for enforcing the declared behavior of applications components. It also provides mechanisms for extending the core functionality of the ApAM machine through the addition of specialized managers.

In ASM, the instances of the concepts of a component are represented as first class objects with a set of attributes (properties) and relationships. The ApAM machine is causally connected to the underlying platform. As a result, actions performed on an ApAM object are transformed into actions performed on the corresponding entities of the platform, and vice-versa.

The ApAM machine automates the creation of "wires" between the components that compose an application. This automatic wire management (creation, destruction, substitution) is performed dynamically in conformity with the applications definition and requirements. This capability supports the automatic creation of dynamic and adaptable architectures.

ApAM system

The behavior of the ApAM system is the result of the collaboration between the ApAM machine and a number of managers. In short, the ApAM machine is responsible for maintaining the Application State Machine (ASM), and for tracing **causality** between the ASM and the underlying platform. It also guarantees the properties that are associated with ASM objects. However, the ApAM machine does not modify the ASM. Modifications are delegated to the **managers** that are currently available. There are two types of managers: the core managers that come with the ApAM core, and the managers that are part of the “extended ApAM”.

ApAM core managers

Three classes of managers are defined:

- Dependency managers that are called when a dependency needs to be resolved.
- Property managers that are called when a property of a component is modified.
- Dynamic managers that are called when a service appears or disappears.

ApAM extensibility is based on managers that are provided by third parties. Two managers, which are closely related to the ApAM core, are provided in the distribution of the ApAM system: ApamMan and DynaMan.

ApamMan is the default dependency manager. The function of ApamMan is to resolve a dependency by looking into the components that are currently running on the platform, and by taking into account the visibility expressed in the application composites. If an implementation is found but no instance is available, ApamMan creates an instance of the selected implementation.

DynaMan is the default dynamic manager. DynaMan interprets the strategies defined in the applications composites, the dependencies when a resolution fails, as well as when a component appears or disappears.

With ApamMan and DynaMan, all information, properties and characteristics defined in the component definition are fully enforced. It is strongly discouraged to re-implement these two managers as this may significantly change the core semantics of ApAM.

Extended ApAM

The standard distribution of ApAM comes with the following basic two managers:

OBRMan is a dependency manager that extends ApAM with dynamic deployment. During a resolution, OBRMan is called if ApamMan has not found the appropriate service (the right service is not currently running on the platform, or is not visible). OBRMan looks for the component in a number of bundle repositories to find out a service that satisfies the dependency requirements; if found, the corresponding bundle is deployed.

DistriMan is a dependency manager that extends ApAM with distribution. During a resolution, if ApamMan has not found the appropriate service (the right service is not currently running on the platform, or is not visible), DistriMan can be called (after, or before, or instead of OBRMan). DistriMan searches for a visible ApAM machine on the network on which a convenient service is currently running. If found, a proxy is created from the current machine toward the distant service, and the resolution returns the address of the proxy.

Specialized and domain specific managers

ConflictMan is a dynamic manager for solving access conflicts between two or more applications that request an access to the same exclusive service.

Having presented the principles of the underlying generic middleware (i.e. ApAM in conjunction with OSGi), we now present the functional decomposition of the HMI middleware followed by a more detailed specification.

2.2. Software Architecture of the HMI Middleware

As shown in Figure 3, the HMI middleware for the AppsGate STB is structured as a two distinct sets of software components: the **Core HMI (CHMI) Middleware** – which is domain-independent, and the **Extended HMI (EHMI) Middleware** – whose components are designed for a particular class of application domains such as smart homes.

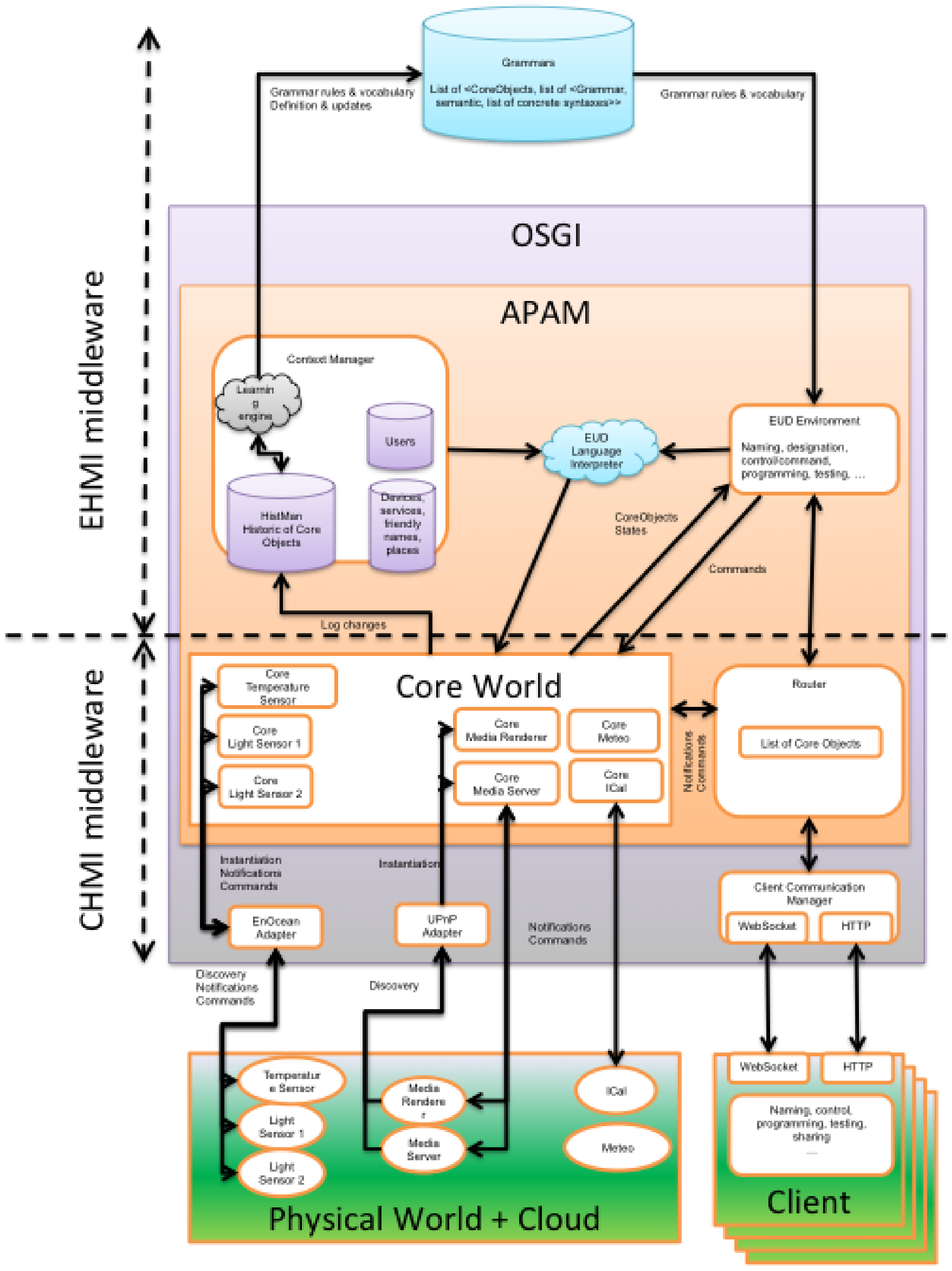


Figure 3. Functional decomposition of the HMI Middleware.
(Arrows denote the direction and the nature of information flow between the components of the architecture).

2.2.1. Core HMI Middleware (CHMI)

The CHMI middleware bridges the gap between the external world and the AppsGate STB world at the appropriate level of abstraction to hide the heterogeneity of the external world. Its functional coverage includes: (1) Technological integration of an heterogeneous set of sensors/actuators network protocols such as EnOcean and UPnP; (2) Uniform representation and management of EnOcean/UPnP/etc. compliant physical devices as well as of third parties services, such as Google calendar available from the cloud; (3) Communication with client applications that run on high-end interaction devices such as SmartPhones, tablets, robots, and smart objects used by end-users to interact with the Smart Home. (4) Automatic support for dynamicity.

(1) Adapters as a means to support technological integration. As shown at the bottom of Figure 3, devices from the **Physical World** are integrated by the way of **adapters.** Adapters are implemented as OSGi bundles (often relying on the available OSGi base drivers). There is one adapter per type of protocol supported by the CHMI middleware. For example, the **EnOcean adapter** and the **UPnP adapter** bridge the gap between the physical devices supported by the EnOcean and UPnP protocols respectively, and the Core World.

(2) "Core world" as a uniform representation of the physical world and of the cloud digital world. **Core World** denotes the representation that the core HMI middleware has about the physical world and of the cloud digital world. This representation is expressed in terms of **Core objects**. Core objects are implemented as ApAM components. For example, in Figure 3, **CoreTemperatureSensor** is an ApAM component that represents a physical temperature sensor handled by one or more physical sensors network protocols (e.g., the EnOcean sensor network protocol). Similarly, **CoreiCal** is the abstract representation of a Calendar service made available in the cloud. Note that the external world includes "things" that are unknown from the HMI middleware. Any **CoreObject** may be reused and enriched as an **ExtendedObject** in the Extended HMI to fit the purpose of the services, such as the Context Manager, provided by the Extended HMI.

(3) An asynchronous client-server architectural style for supporting communication with Human-Smart Home interaction devices. Smartphones and tablets are typical interaction devices. As shown in Figure 3, the **Client Communication Manager**, which is implemented as an OSGi bundle, is the mediating component between the **clients** and the **Router**. Client applications are developed with web technologies for whatever web browsers. The PhoneGap framework is used to generate hybrid mobile applications[1] for iOS, Android, Windows 7, etc. By doing so, we address the heterogeneity of technological spaces of the client platforms. The Client communication manager supports web sockets connectors as well as http. The Router serves as the communication component between all the ApAM components running on the STB.

(4) ApAM mechanisms as a way to support dynamicity automatically. A difficulty in developing applications that use and control the physical world is that this world is dynamic. Dynamic means that the entities of the physical world, thus their representation in the Core World, appear/change/disappear as a result of events that happen in the physical world. ApAM handles these events automatically such that applications are resilient to changes of the physical world.

2.2.2. Extended HMI Middleware (EHMI)

The EHMI middleware is an extensible set of software components designed for facilitating the development of smart environments. For the purpose of the AppsGate project, it includes, but is not limited to, the following components: a context manager and an End-User Development Environment (EUDE) called SPOK (Simple PrOgramming Kit).

[1] In the context of User Interface development, an hybrid application is implemented with a mix of technological spaces including a native SDK (such as the Android SDK) and web technologies (such as HTML5, CSS and Javascript).

Context manager

The **context manager** is structured into multiple levels of abstraction, as illustrated in Figure 4. At the lowest level of abstraction, the sensing layer corresponds to numeric observables produced by the Core World. These observables, which correspond to changes in the Core World, are automatically logged by the ApAM **PropertyHistoryManager**. To determine meaning from numeric observables, the system must perform transformations. The perception layer provides symbolic observables at the appropriate level of abstraction. The situation and context identification layer identifies the current situation and context from observables, and detects conditions for moving between situations and contexts. Services in this layer specify the appropriate entities, roles, and relations for operating within the user's activities. They can be used to predict changes in situation or in context, and thus anticipate needs of various forms (system-centric needs as well as user-centric needs). Finally, the exploitation layer acts as an adapter between the context manager and its "clients". All these can be pre-specified in an ad-hoc manner by developers or may be learned by a **learning engine**. For example, the learning engine may infer rules as well as detect recurrent patterns of human behavior. It may then ask the end-user to specify a name for this pattern (e.g., "party with friends at home"). By so doing, **grammars** used by the EUDE can be dynamically synchronized to reflect the current services and devices available in the smart home.

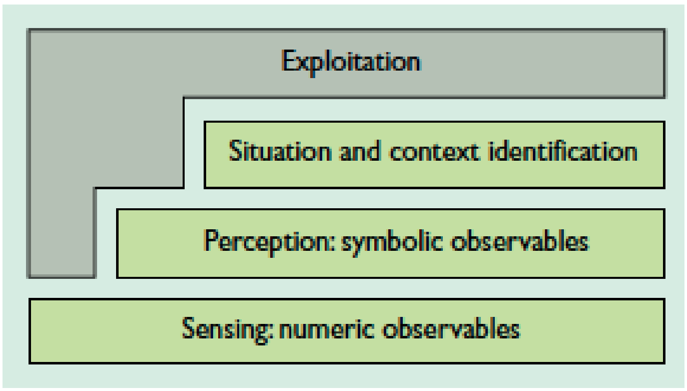

Exploitation
Situation and context identification
Perception: symbolic observables
Sensing: numeric observables

Figure 4. Levels of abstraction of a context manager [From Coutaz et al. 05].

At the time of this writing, the context manager is limited to a **PlaceManager** that allows end-users to create places (e.g., kitchen, reading corner, etc.) and to associate devices and/or services to these places. **PlaceManagerSpec** specifies the services that **PlaceManager** offers such as create place, get places, put device at place.

SPOK, the AppsGate End-User Development Environment

The current version of SPOK includes an editor for editing programs using a pseudo-natural language and an interpreter. A multi-syntax editor as well as additional services such as a debugger and a simulator are expected for the second version.

A *multi-syntax editor* will allow users to build syntactically correct programs using the syntax that is most appropriate to them or by using a combination of them. These syntaxes include pseudo-natural language (i.e. a constrained natural language) and graphical iconic syntax (as exemplified by Scratch [Maloney et al. 2010]). The interaction techniques used to enter programs may be menu-based, free typing, as well as by demonstration in the physical home or by the way of the simulator. The *simulator* is the dual digital representation of the real home. It is intended to serve also as a *debugger* for testing and correcting end-user programs.

Whatever syntax used by end-users, programs are translated into syntactic abstract trees whose leaves reference services provided by the Core HMI and/or by the Extended HMI Middleware. The *interpreter*, executes end-user programs, using the corresponding syntactic abstract trees as input.

A detailed description of the components of SPOK is available in Annex 3.

2.3. Specifications of the Core HMI (CHMI) Middleware

In this section, we present a refinement of the functional decomposition of the Core HMI Middleware in terms of the underlying technology used for implementation (i.e. OSGi and ApAM). Figure 5 shows the global implementational architecture of the CHMI as a mix of OSGi and ApAM components. The detailed specifications of the components of this architecture can be found in Annex 1 and Annex 2.

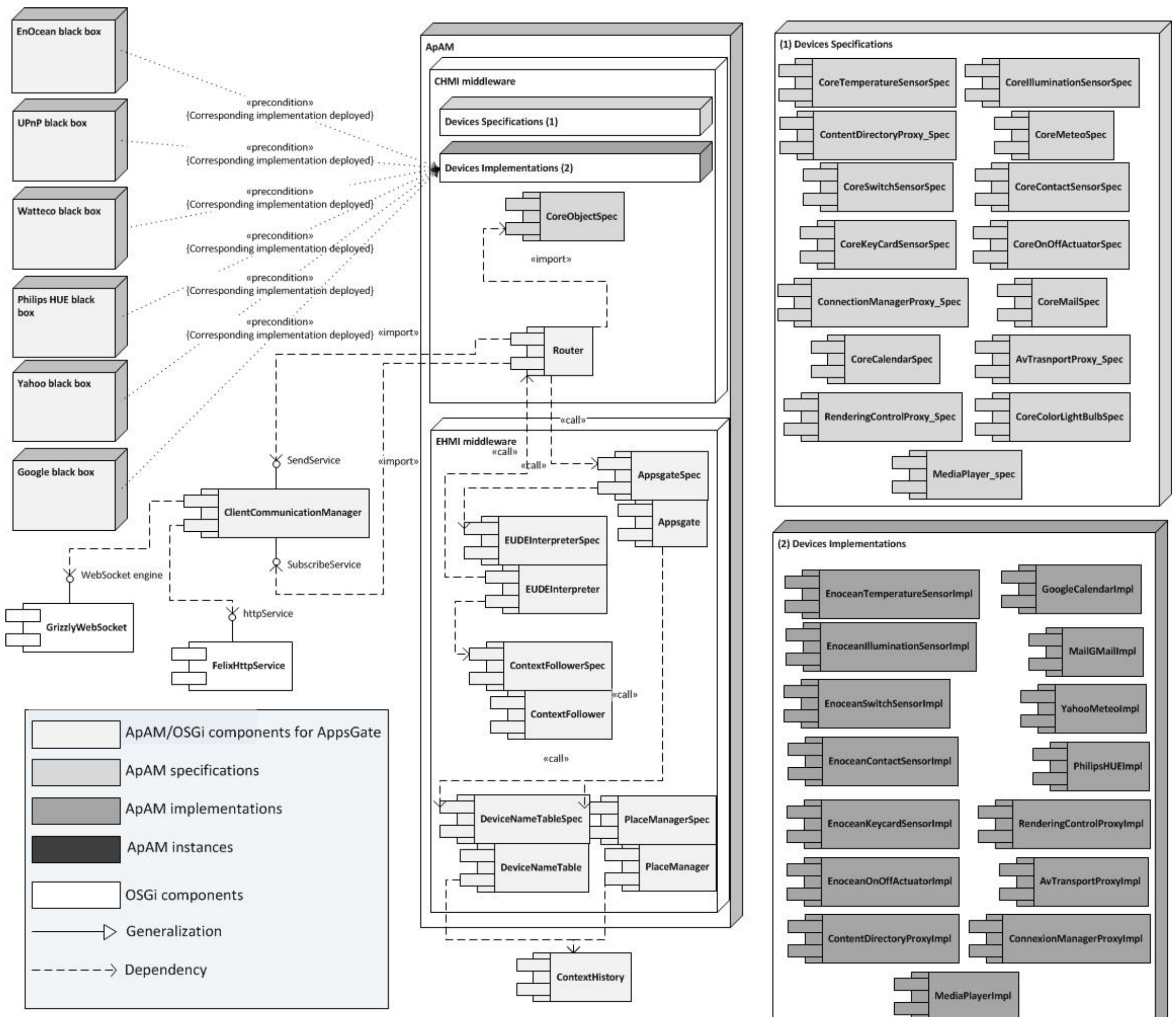


Figure 5. Global architecture of the AppsGate Core HMI Middleware.
(The EnOcean and UPnP black boxes are detailed in Annex 1 and 2. The content of the ApAM boxes labeled Devices Specifications (1) and Devices Implementations (2) are detailed in the two right-most boxes of the figure.)

As described in Section 2.2 above, the CHMI Middleware includes the following three functions:

1. Technological integration of physical devices (e.g., sensors, actuators, media renderer and server) handled by specific protocols such as EnOcean and UPnP.
2. Abstraction of physical devices as well as abstraction of services from the cloud into a uniform representation to constitute the Core World.
3. Communication with client applications running on devices such as smartphones and tablets.

Technological integration. As discussed in the previous section, technological integration of physical devices is performed by the way of adapters (e.g., EnOcean Adapter, UPnP Adapter). When it comes to implementation, these adapters are proxies implemented as OSGi bundles. The detailed implementations used for EnOcean and UPnP compliant devices are presented in Annex 1 and Annex 2.

Abstraction of devices and services into a uniform representation. As discussed in the previous section, any visible physical device from the physical world has its abstract counterpart in the Core World. For example, a physical Temperature Sensor, whether it be supported by EnOcean or UPnP, becomes a CoreTemperatureSensor component in the Core World. When it comes to implementation in terms of the ApAM component metamodel, the CoreTemperatureSensor of Figure 3 is reified as a **CoreTemperatureSensor Specification**, and one (or several) **CoreTemperatureSensor Implementations** that are compliant with this specification (see Figure 5), and at runtime, instances of these implementations. More precisely, an ApAM **<device> implementation** is a specialization of (inherits from) an ApAM **<device> specification** and of the **CoreObject specification**. The latter provides a uniform view for communication. As for the API, we mimic UPnP APIs, when they exist, as this is the case for Multimedia services.

The **router** is a kind of message dispatcher between the ApAM components of the HMI middleware as well as with the clients of the HMI middleware. It is implemented as an ApAM component. Typically, the router forwards the notifications received from instances of CoreObjectSpecs to the **Client communication manager** (for example, to inform the end-user of a state change in the Core World) and vice versa. It also serves as a communicator with components of the Extended HMI middleware such as the context manager and the EUDE.

As discussed above, the **Client Communication manager** is an OSGi bundle that bridges the gap between the code running on interaction devices (e.g., smartphones) and the HMI middleware. It supports two communication protocols, Web sockets and http, each one supported by a specific OSGi bundle (**GrizzlyWebSocket** and **FelixhttpService**). Web sockets are used for applications whose source code such as App source code, is installed on the interaction device.

2.4. Future Work

2.4.1. Lower stacks of the middleware

Building a middleware is challenging as it has to accommodate a diversity of heterogeneous technologies. In addition, it is an ungrateful task as users only notice it when it fails! Strong foundations and reliability are key requirements before using a middleware and building on top of it. The **Core HMI Middleware** brings this promise with clear specifications and API. The whole CHMI is now reliable at runtime and compatible with Android and the AppsGate STB.

Although CHMI is compatible with Android and the AppsGate STB**, Integration with Android** is not fully achieved. Some work must be undertaken so that the HMI middleware (1) benefits from the functionalities provided by the Android APIs and libraries, and (2) interacts with the **Android Application Framework** used by regular end-user applications purchased on the Android application store.

As part of the CHMI Middleware, UJF/LIG is now developing and integrating **Fuchsia**. Fuchsia is an extensible tool for integrating and managing external communication protocols within OSGi. Most of modern applications require multiple sources of information such as web services, databases, and ubiquitous devices. The integration of these services becomes a major issue when dealing with a multitude of them. The **Core World Specifications** brings some abstraction in order to solve this issue for developers. Until now, protocol integration was done on a case-per-case basis in an ad-hoc fashion. This is where Fuchsia plays a key role, by providing the same way of dealing with different technologies, creating a single entry-point for all protocols supported. In addition, Fuchsia improves the **testability and the aggregation of different protocols** on the same platform. As protocols become supported by Fuchsia, they automatically follow the same structure (same framework), thus **enhancing the maintainability and testability while reducing the learning curve for different protocols.**

Integrating new technologies will be easier with Fuschia. The project will take this opportunity to accommodate new sensors and actuators for the home, extending the range of possible scenarios. Ubiquitous computing emphasizes the need of physical devices in these early developments. As the challenge is now addressed, the project will also tackle the integration of **Web services**. For the moment, only weather, mail and calendar are supported. This remains poor when considering the new era of cloud computing. It is now time to bring the digital world to the home, providing social networking, daily news, sharing pictures, video or other media, and much more.

Technicolor, as a partner of the project, promotes **Qeo**. Qeo is a communication framework designed to realize full interoperability between consumer devices. Integration of this framework into the Core HMI Middleware is the next step to bring interoperability within the project. This should allow the partners to share the devices and services across AppsGate clusters.

2.4.2. Upper stack of the HMI Middleware and End-User Experience

The Extended HMI middleware will be improved in relation to the needs of the Smart Home application described in Deliverable D1.4. The following key issues must be addressed:

- *With regard to the SPOK editor*: support for multiple syntaxes and interaction techniques to allow end-users to develop programs in a flexible and easy manner.
- *With regard to the SPOK interpreter*: support for semantic robustness so that users cannot build programs that are inconsistent.
- *Development of a simulator* that acts as a dual representation of the actual smart home as well as a debugger for testing and correcting end-users' programs.
- *Development of a meta-UI* for the home to allow end-users to live safely and happily in their home (e.g., programming, installing&replacing devices and services, as well as understanding and controlling the state of their smart home). The challenge is to reach simplicity for an increasingly complex environment.

3.Enhancements to the IISU Middleware

3.1.Foundations

SoftKinetic Software intends to enable the AppsGate platform with gesture-based interaction by using multiple depth sense cameras with the iisu® HMI Middleware The iisu software development kit is a complete platform for gesture application development and deployment in immersive and intuitive applications. Its specifications are available in the AppsGate Deliverable 1.3. In summary, its user tracking functionality captures multiple user's movements in real-time with volumetric information and full body skeleton extraction. Furthermore, iisu identifies the main user and computes statistical data, including user's height, body and torso orientations, center of mass, chest and pelvis direction, etc. In close interaction (CI, at 15cm to 1m away from the camera), iisu also provides 2 hand 10 finger tracking information with automatic detection of the hands. At the start of this project, iisu was available on windows OS only, while the AppsGate platform is Linux-based.
In this document we describe the preparatory work that was required before the iisu® HMI Middleware may be reworked to a multi-camera full body analysis library on the AppsGate Software stack. Namely, how to keep an internal debugging visualization tool available for the developers of iisu, and how to handle the porting to Android. Later, the deliverable 2.4b will report on the implementation of the requirements presented in this document.

3.2.Software Architecture (Functional Decomposition)

As shown in **Erreur ! Source du renvoi introuvable.**, the core technology module of iisu is comprised of the algorithms for long and short range depth image analysis, while the Kernel organizes the processes into smaller units that exchange events, data and parameters, and a generic API (named ITL, iisu Template Library) sends and receives these for usage in the tools for developers (easii, wrappers, bridges, toolbox, interaction designer). We refer to the developer's guide, online at the support section of www.softkinetic.com as mentioned in D1.3, for more detailed architectural specifications on the iisu API and its usage by developers of gesture-based applications. During the deployment by our clients of previous releases of iisu, we received requests to use only the long range, or only the short range depth image analysis. Unfortunately, when only a portion of the results from the interaction image analysis is needed, the drawback of the current heavy dependencies between the Kernel, ITL and core technologies architecture is that complexity and high compilation time are unavoidable. Hence, the work of SoftKinetic Software in the AppsGate project has primarily focused on exploring options to break up the iisu core technology into small library blocks while keeping an internal debugging visualization tool. The requirements for this significant enhancement are explained in the next paragraph **Erreur ! Source du renvoi introuvable.**. On the other hand, preliminary to the Linux-based AppsGate platform integration, some bricks of the iisu core technology have been compiled on Android using NDK. This is detailed in paragraph **Erreur ! Source du renvoi introuvable.**.

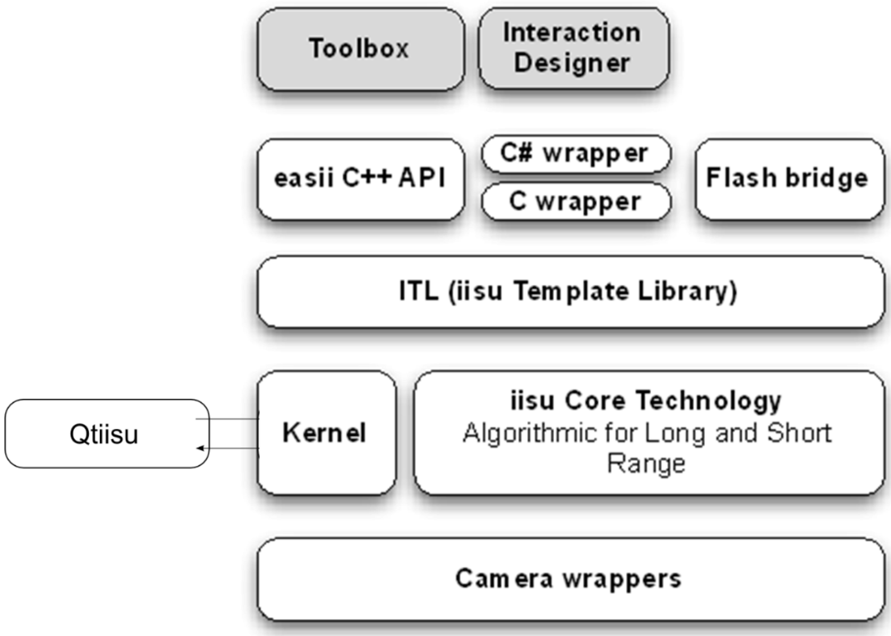


Figure 6. Softkinetic iisu current middleware architecture, and the coupling to Qtiisu.

3.2.1.Challenges in designing specialized gesture middleware based on iisu.

Softkinetic has investigated the architecture of its iisu middleware for use with multiple cameras that typically capture 3D data at a different range. This fine-grained division naturally has led to specifying specialized APIs that each solve part of the problems, such as a long range API for the DS311 product or a short range interaction API for the DS325 product. The down side to this approach is that the lack of a unique API makes it more complex to build visualization tools that are mandatory for debugging the core technology. There are many problems when it comes to build a unified rendering for a heterogeneous set of libraries. The main issue is the lack of access to the internals of libraries. Most libraries expose only the data and parameters that are relevant to the user whereas for the library developer (the person who needs to ensure that all functionalities of the software – in this case iisu - are operational), all the internal structures are of interest. Because the core technology of iisu represents the algorithms that work on the 3D data of a scene captured by a *DepthSense* camera (specifications also in D1.3), and because the results of the

algorithm application can be best represented as 3D data as well, the debugging of iisu must be handled via visualization of 3D structures such as 3D points, lines, surfaces, representing user masks, skeletons, hand positions, etc. based on the internal structures in its API(s).
Qtiisu, the current internal debugging visualization tool of the current released *iisu* architecture, as illustrated in **Erreur ! Source du renvoi introuvable.**, was built to address the debugging problem on iisu and did so by having *iisu* expose additional API internal data only for the visualization tool. This tight coupling, shown in **Erreur ! Source du renvoi introuvable.**, was possible because the internal structure of iisu allowed *Qtiisu* to browse through the internals of the library. When working with an arbitrary library such data structure, to organize coupling of the internals, does not always exist and if it does it will most likely not have the same API.
In addition, *Qtiisu* had many other issues (outside the scope of the AppsGate project) that needed to be addressed, so the decoupling of the rendering (to be organized in a front-end framework) from the input data access (to be organized in a back-end framework) was a natural evolution point for this debugging visualization tool.

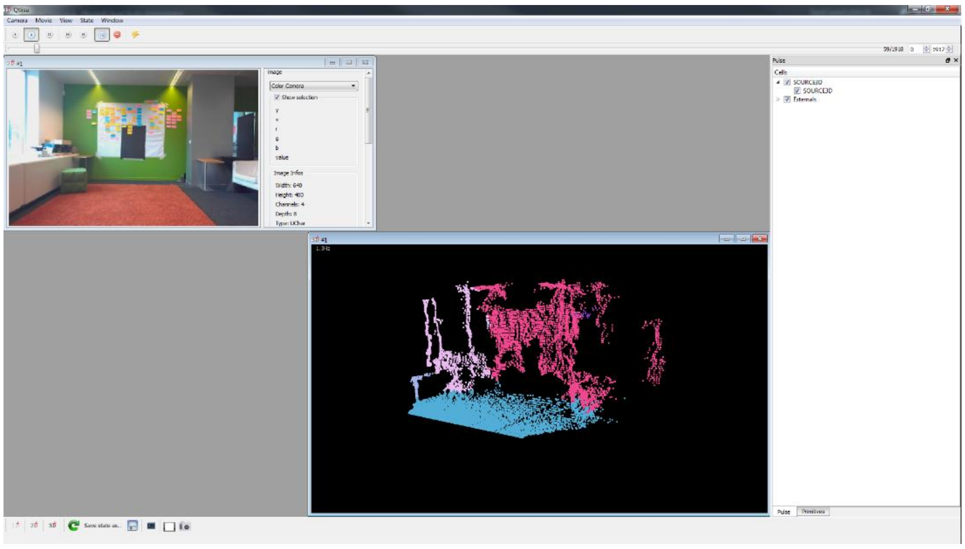


Figure 7. *Qtiisu*, the internal debugging visualization tool illustrated on a long range depth scene

The proposed changes of the library (Kernel) thus required to redesign the visualization tool and its pipeline. Given this, we made an assessment of the requirements that the new way of debugging needs to satisfy and how well already existing tools can help with that. Here is a list of the challenges:

- *Provide a clean and easy means to disable the data exposition parameter:*

  Qtiisu relies on the internal data structures of the iisu Core technology. As explained before, when iisu is to be split into specialized API, the internal data structures that are used now might not be accessible anymore. Indeed, a developer of a processing library should not bother to put all internals in an exposed structure if these do not contribute to its function, especially because this introduces additional compilation time. In general, exposing all internal structures is not a good practice.
- *Decouple library and primitive rendering code:*

  In its current state, Qtiisu does not support scripting nor coding of 3D structure data. It only contains the primitive rendering. Consequently, the code to create a primitive and 'position' it in 3D space must be written inside ITL, resulting in tight coupling between the ITL library and the Qtiisu renderer. Ideally, the ITL library should only worry about what data must be visible and not as what kind of primitive should be shown. The way data is shown must be performed by the front end in an easily changeable (iterative) way. The developer should not have to modify the library code to visualize its exposed content.
- *Provide support for multi-threaded and asynchronous library:*

  The specialized libraries could be designed to run in multi-threaded and/or asynchronous mode, and Qtiisu has no build-in support for this. Once again, it relies on ability of the library to expose its data properly, as the current iisu release does. When a data structure becomes the visualization responsible, it is also there that thread safe mechanism should be provided. For the use with an asynchronously designed library, the rendering tool must provide ways to synchronize library data output to keep the viewed data consistent.

- *Multiplatform support:*

  Both frameworks (back-end and front-end) must work on the usual platforms (e.g., Windows OS, Linux) or at least it should be easy enough to port.
- *Provide support for remote debugging:*

  Many platforms cannot accommodate to run the visualization front-end (Qtiisu was for use on windows OS only) and thus the back-end framework must provide remote access to run the specialized API on a target platform while rendering its results somewhere else. Qtiisu does not have such networking capability. It relies solely on the design of iisu which was not built to move internal/private data across the network. This is a huge limitation because nowadays iisu is continuously updated to be embedded on many resource-constrained devices with various OS. Within the Appsgate project, we note the port to Android on the STB.
- *Easy to use and efficient front end:*

  The user interface of Qtiisu is minimalistic and not very efficient. On top of this, it is not very evolutive and requires substantial effort to improve.
- *Multi language access:*

  The rendering front-end framework must be able to work with iterative small changes in the API code without the need for continuously updating the internal structures. Nowadays, Qtiisu acts as a passive visualization tool, only capable of rendering the result of the iisu algorithms by running the compiled iisu release on a 3D data input movie. As noted above, it is therefore critical to have the primitive creation and positioning code in the rendering tool, and not in the processing API containing the core technology with all the algorithms. For efficient debugging, it is also critical for that code to be written in a language with no (or very low) compile time. Depending on the needs, some languages are better suited than others (e.g., scientific plotting is easier in Python than in most other languages). Qtiisu does not provide any such flexibility and requires each change to be done by recompiling the ITL library, even just for changing the color of a rendered point. This makes it hard to create small packaged libraries that still would have self-contained primitive creation and positioning based on parts of the current release of iisu.

Given these challenges, we are building a new way of debugging, called *Milda*. It is composed of the following:
The back-end framework: is a C++ library whose goal is to allow the specialized API to export any data it wishes to share. Data can be exposed as read or write, and the library can also place hooks in the code to allow a scripting front-end to override the behavior of the code found there. This last feature supports rapid prototyping (for small iterative changes) with immediate result in the rendering tool's interface with a language that does not require any significant compilation time and thus supports iteration very quickly before committing a final version in the specialized API.
The data abstraction layer: is a layer that exposes the backend across platform/language. Thrift [thrift.apache.org] is used to this end. Here is a quote from the apache web site of thrift:
"The Apache Thrift software framework, for scalable cross-language services development, combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages."
As seen in this description, thrift allows us to map our data to any language and provides RPC for communication.
The front-end framework: for which the tool used will depend on the needs of the debugging. Scripting can be done using either Python [www.python.org] for quick testing (to handle the hooks in the code) or to make scientific views (plot, matrix, …). A 3D rendering engine front-end can be

Unity [unity3D.com]. By using existing tools, we take away the development and maintenance constraint on the front-end itself and can focus on building packages to extend the existing behavior and editor of Unity to support our needs (e.g., playback control, custom assets, data readers, image viewers, …). It is of course possible to plug any other front-end and all of them can run together on the same specialized API via the back-end and the data abstraction layer.

3.2.2.Challenges in enabling iisu on the STB Android stack

The STB software stack will bring TV zapper and gesture functionality to Android, or vice versa, depending on customer's preference. ST has carried out the architecture work to integrate Android along the legacy STB stack (SDK2), and has given 2-day training (14th and 15th of May, 2013) to allow the AppsGate partners to get familiar with the STB android stack. SoftKinetic signed the license agreements and an engineer of the embedded team followed the training, based upon which the work as described next was done.

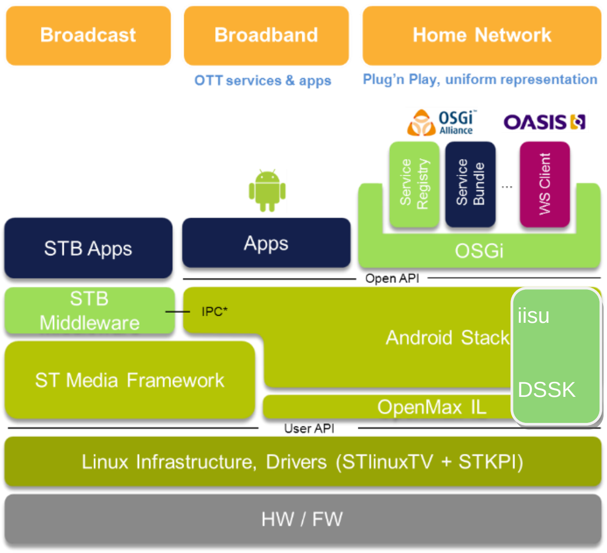
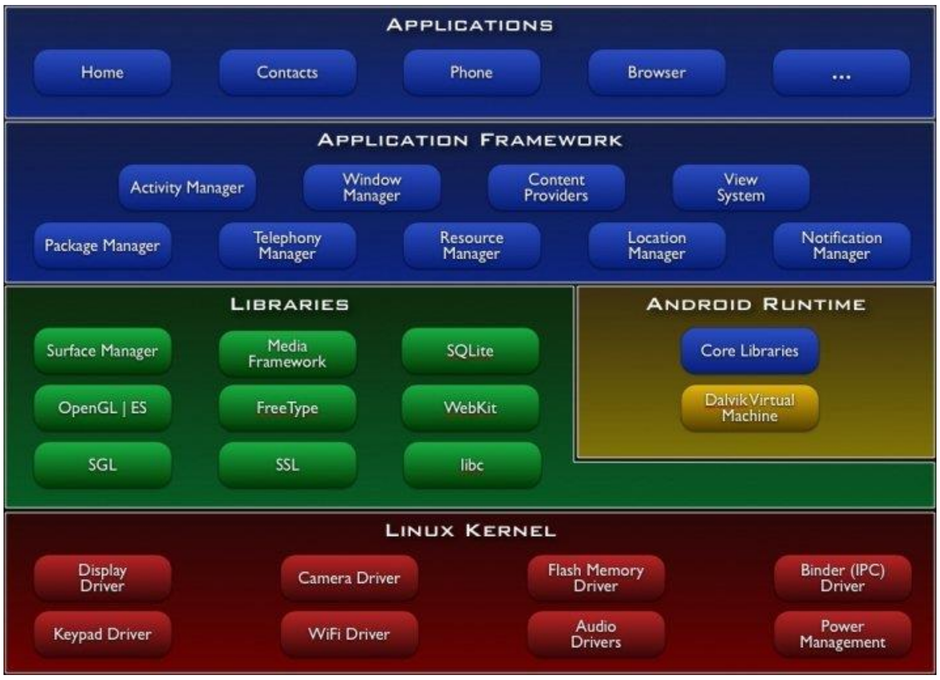


Figure 8. Location of the iisu middleware in the STB software stack

Android is a completely new operating system, sharing a lot of common ground with linux systems, while at the same time being completely different in some aspects. The main difference is that Android applications are, for the most, written in Java, and are executed within a sandboxed virtual machine called Dalvik. However, Android allows the use of native code inside applications and libraries. This is mainly targeted at performance critical applications (such as games, image processing softwares, …), native code being obviously more effective. The provided native tool chains do however lack support for some core functionalities of the C and C++ languages, which is not without causing issues when working on a huge code base such as *iisu*.
Iisu on Android is a native library, but it can perfectly well be used from any standard Java-based Android application through the use of JNI, Java Native Interface. Iisu is not provided as java API. This process being quite complicated to setup at first, we created a few samples to illustrate the correct use of JNI with *iisu*. We also created some graphical samples for demonstration and quick test purpose. For the set-top box use case, the focus has been on making the long range skeleton analysis, as specified in D1.3, available on Android.
Iisu on Android is tightly tied to our *DepthSenseSDK* camera driver: only DepthSense cameras are supported. The *DepthSenseSDK* user-space library has thus been ported as well, which required a lot of modifications to comply with Android low-level interfaces that differ quite a lot compared to linux and Windows systems.

The Android layout is explained on the following figure. In this figure, the *iisu* and the *DepthSenseSDK* libraries would reside in the 'Libraries stack'. *Iisu* is queried by applications or libraries from the 'application framework' and 'applications' in the stack (through the use of JNI, or directly for purely native applications), and retrieves 3D data from *DepthSenseSDK*. The *DepthSenseSDK* library is then talking to the 'Linux kernel', to get data from the hardware.



The Android stack

3.3.Future Work

3.3.1.Specifications of Milda

SoftKinetic Software described the challenges for the architecture of iisu to handle multiple cameras efficiently: we have decided to focus on the solution that brings market value, decreases the workload for QA and release, and decreases compilation time while debugging.
The proposal is to build a per camera product based on common technology bricks as an alternative to one abstraction layer handling the input of all cameras. This may indeed allow quicker product releases, and may also facilitate the deployment on platforms that request analysis of multiple cameras' data.

A library per iisu dedicated depth range - or in practice dedicated Depthsense camera - , taking input from a set of technology bricks, as displayed in **Erreur ! Source du renvoi introuvable.**, is the most promising.

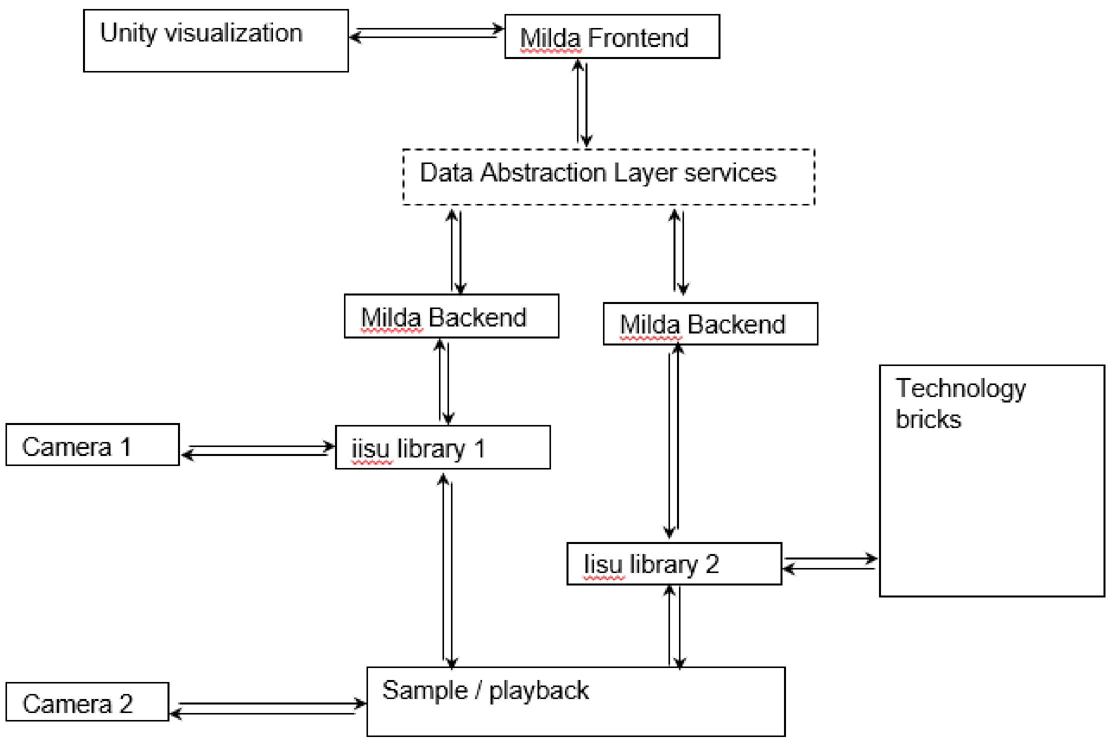


Figure 9. Proposed architecture showing the coupling between the cameras, iisu and Milda

Within the further work in T2.4, the link of *Milda* to the new proposed architecture will be laid out. This architecture is a solution for challenges described above, namely:

- *Provide a clean and easy means to disable data exposition parameter:*

  Data is exposed using the milda back-end and all expositions are encapsulated in MACROs that can be disabled. The most common operations are one line MACROs.
- *Decouple library and primitive rendering code:*

  There is no primitive creation and position code anymore in the ITL library. Only the core 3D processing algorithms' data should be exposed. All the primitive rendering is performed in a front-end.
- *Provide support for multi-threaded and asynchronous library:*

  The back-end is thread-safe and provides synchronization mechanism so that any specialized API provider can use those to expose data in a consistent manner.
- *Multiplatform support:*

  The backend is standard C++, the main frontends are Unity and Python. Windows and Mac are hence natively supported. Unity does not support Linux but can target it: the manipulation of the 3D data cannot be done in the provided editor of the engine on Linux, but small unity application can be trivially built to run on Linux. Given that everything is exposed through thrift, all data are accessible through many languages on most platforms, so other alternative to Unity can also be used.
- *Provide support for remote debugging:*

  By using thrift as a glue between the back-end and the front-end we can seamlessly work on a remote library.
- *Easy to use and efficient front end:*

  The main front-end will be Unity. This allows us to benefit from all the improvements done already in the Unity engine (in performance and in user interface). We just concentrate on developing a package to improve Unity for our needs (custom windows, prebuilt data viewer, …)

•*Multi language access:*

Again, thanks to the thrift middleware the content exposed through Milda is accessible in many languages and the front-ends can run simultaneously (e.g., viewing the skeleton in Unity while plotting some camera noise function in Python).

This proposed change on the library level requires a redesign of the internal visualization tool and pipeline ‘Qtiisu’. **Erreur ! Source du renvoi introuvable.** shows the diagram of the proposed Milda framework, satisfying the requirements identified in Phase 1 of the project, in use with any C++ library.

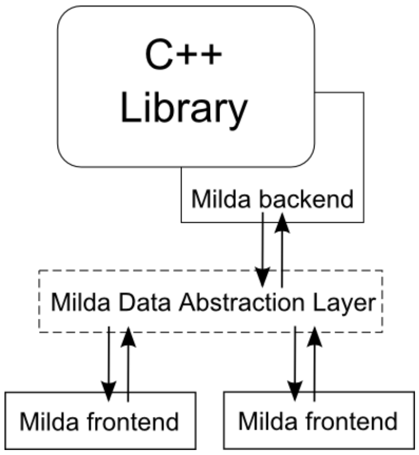


Figure 10. The Milda framework.

While creating the front-end solutions, we will keep on using Qtiisu as a front-end for Milda. This is of course not perfect given the previously expressed limitations of Qtiisu, but it will allow the independent implementation of the back-end framework and data abstraction layer before having to develop a complete front-end in Unity.
This effort will be completed during the project. In addition, the important refactoring of the architecture to maintain clean technological bricks must be initiated.

3.3.2.Specifications of iisu (close range) on Android

Our first efforts have focused on bringing the long range experience to the Android platform, thus providing support for the long range DS311 camera and the long range pipeline of iisu. The next step will logically be to do the same for our close-range solutions. This will thus consist of adding support for the close range cameras inside DepthSenseSDK, with the optimizations that this will require, the resolution provided by these cameras being higher, hence heavier processing. Then, we will port our close range library to be able to provide meaningful results to the user based on these close-range 3D points. This will also require heavy optimization of the algorithms using both the CPU and the GPU to be able to sustain the camera’s frame rate and higher resolution. An example for using iisu on Android for the Application developer on the AppsGate STB will be provided in the Annex of 2.4b later.

3.4.Conclusion

SoftKinetic Software has reviewed the challenges in the architecture to use multiple cameras, and will go forward with this in the Phase 2 of the project.

4. References

[Estublier et al. 2012] Estublier, J., Vega, G. "Reconciling Components and Services: The Apam Component-Service Platform", *IEEE SCC 2012 - International Conference on Service Computing*, Jun 2012, Honolulu, HI, United States.

[Maloney et al. 2010] Maloney, J., Resnik, M., Rusk, N. Silverman, B., Eastmond, E. " The Scratch Programming Language and Environment", *ACM Transactions on Computing Education*, 10(4), 2010, 1-15.

[OSGi 2012] OSGI Alliance, "OSGi Core Release 5 Specifications", *http://www.osgi.org*, March 2012.

[UPnP 2002] Ritchie, J. , Kuehnel, T. "UPnP AV Architecture:1", *http://upnp.org*, UPnP Forum, Standards: Device Control Protocols, June 25, 2002.

5. Annex 1: Technological integration in the CHMI Middleware

5.1. EnOcean detailed architecture

Ubikit, developed by Partner Immotronic on top of OSGi, provides the basics for EnOcean integration. As shown at the top left corner of Figure 11, Ubikit includes three OSGi bundles to manage the EnOcean radio network protocol (**RXTX Serial**), to maintain a representation of the physical world (**pem-EnOcean**), and to provide a uniform entry point for its "clients" (**Ubikit**).

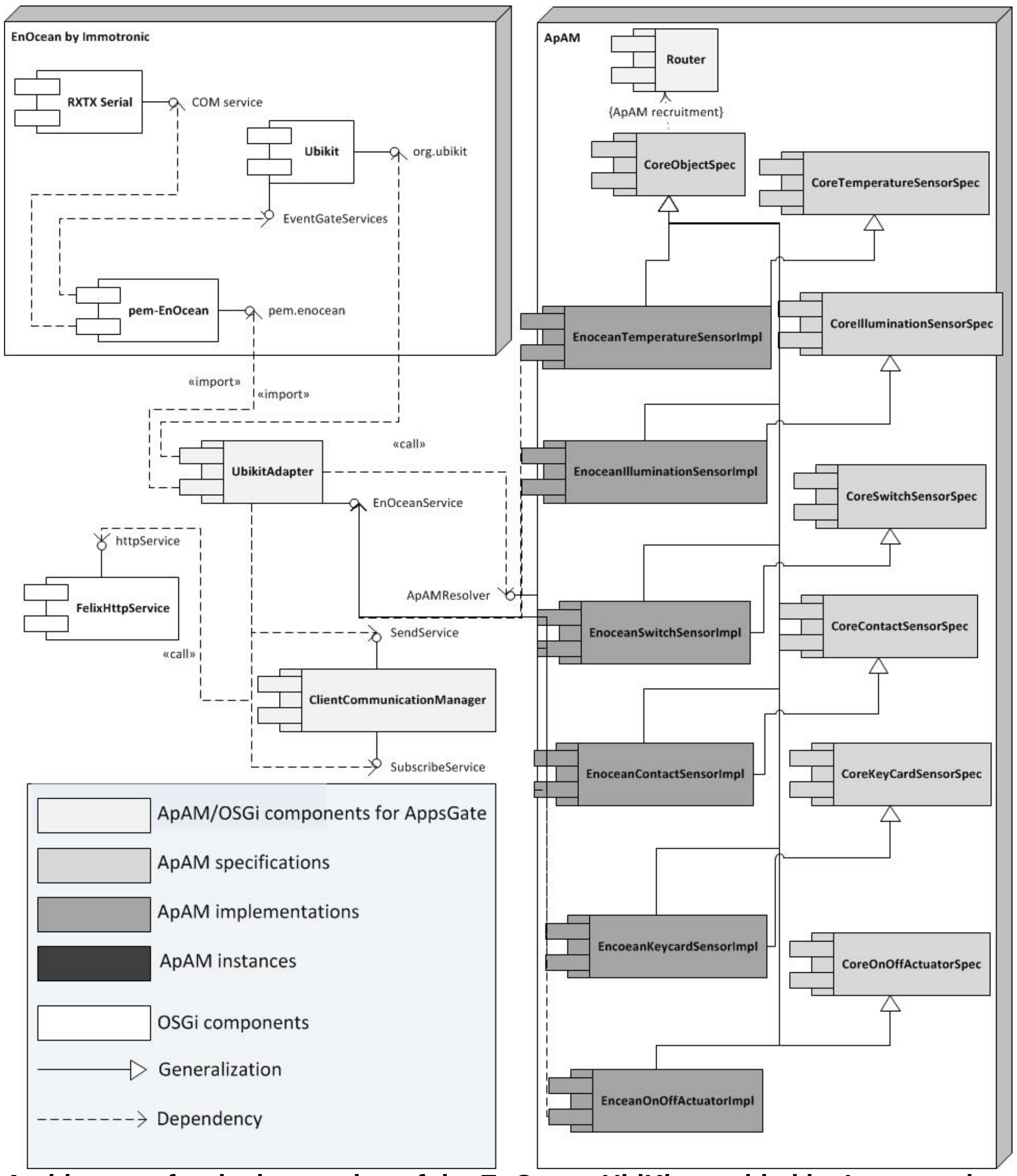


Figure 11. Architecture for the integration of the EnOcean UbiKit provided by Immotronic.

As discussed above, the OSGi **UbikitAdapter** component serves as the adapter between Ubikit and the ApAM Core World. UbikitAdapter is also connected to the **client communication manager** to offer a web configuration GUI that allows developers to pair EnOcean sensors and actuators. The configuration mode is initiated with the configuration GUI provided by UbikitAdapter.

At this time of the writing, six sensor/actuator implementations have been developed in conformity with the six sensors/actuators specifications: temperature, illumination, switch, contact, and keycard sensors, and the on/off actuator. As shown in Figure 11, these implementations (e.g., EnOceanTemperatureSensorImpl, EnOceanIlluminationSensorImpl, etc.) inherit from CoreObjectSpec so that the **Router** can reference them in a unified way for message passing as well as from their sensor-specific specification (e.g., CoreTemperatureSensorSpec, CoreIlluminationSensorSpec, etc.). They are connected to UbikitAdapter through one single service. They are instantiated by UbikitAdapter when a new physical sensor/actuator is paired by the end-user with the ApAMResolver static field. Sensor/actuator instances send/receive EnOcean radio telegrams through the Ubikit event system.

5.2.Media Services detailed architecture

UPnP Media Services

The integration of UPnP devices is based on the **UPnP Base Driver** from the OSGi distribution (see top left corner of Figure 12). UPnP Base Driver offers a discovery service and registers all UPnP devices and services discovered in the local area network in the OSGi registry. **UPnP Adapter** serves as an adapter between those discovered UPnP devices and services and the ApAM Components. Drawing from the standardized description of UPnP AV/DLNA compliant devices [UPnP 2002] the **UPnP Adapter** retrieves **Media Services** and calls the **UPnPProxyGenerator** to create corresponding proxies for the devices and services discovered. Those proxies are provided as ApAM Components and offer functional media streaming and rendering capabilities.

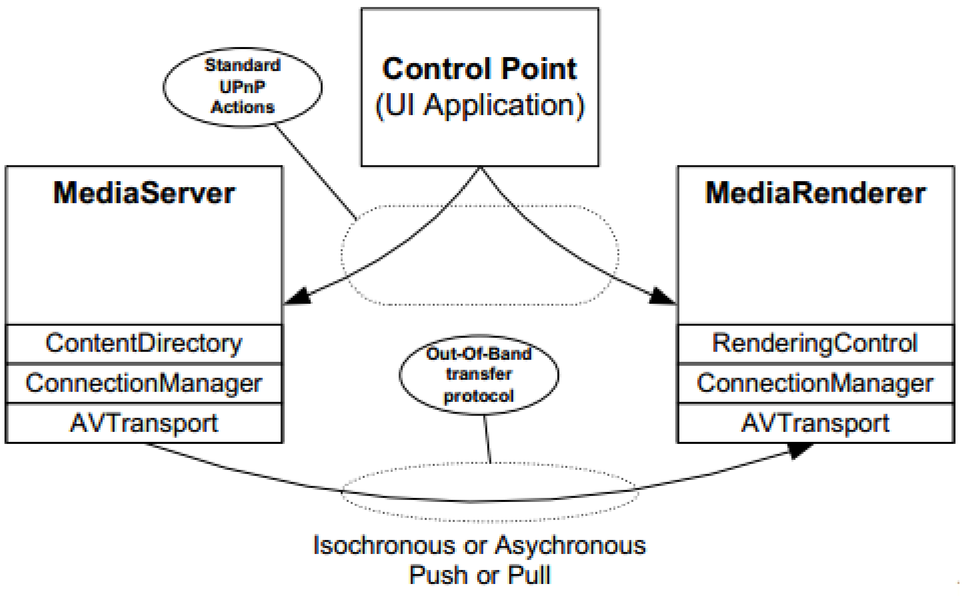


Figure 12. UPnP AV Playback Architecture.

- Each instance of **MediaRendererProxyImpl** device is responsible for rendering media to the end-user (on Set-Top-Box, smartphone, tablets, etc.). It provides **RenderingControlProxyImpl**, **ConnectionManagerProxyImpl** and **AVTransportProxyImpl** services.

- Each instance of **MediaServerProxyImpl** device is responsible for selecting and streaming media content (e.g., MediaLibraries, NAS, DVR). It provides **ContentDirectoryProxyImpl**, **ConnectionManagerProxyImpl** and **AVTransportProxyImpl** services.

Each of these services is compliant with UPnP standards but also inherits from **CoreObjectSpec**. Therefore, they can be referenced and used by the **Router** in a unified way.

High-level Media Service Integration

The UPnP AV specification provides powerful functionalities but configuration is quite tricky for end-users that just want to enjoy their digital contents. High-level abstractions of these devices are provided by the **MediaPlayerFactory**. For each **MediaRendererProxyImpl** discovered, an **UPnPMediaPlayer** component is created. This ApAM implementation inherits from **CoreMediaPlayer** specification. The **CoreMediaPlayer** offers the expected methods such as play/pause/stop, and set volume, hiding the underlying calls and complexity of the actual UPnP services (see Figure 13).

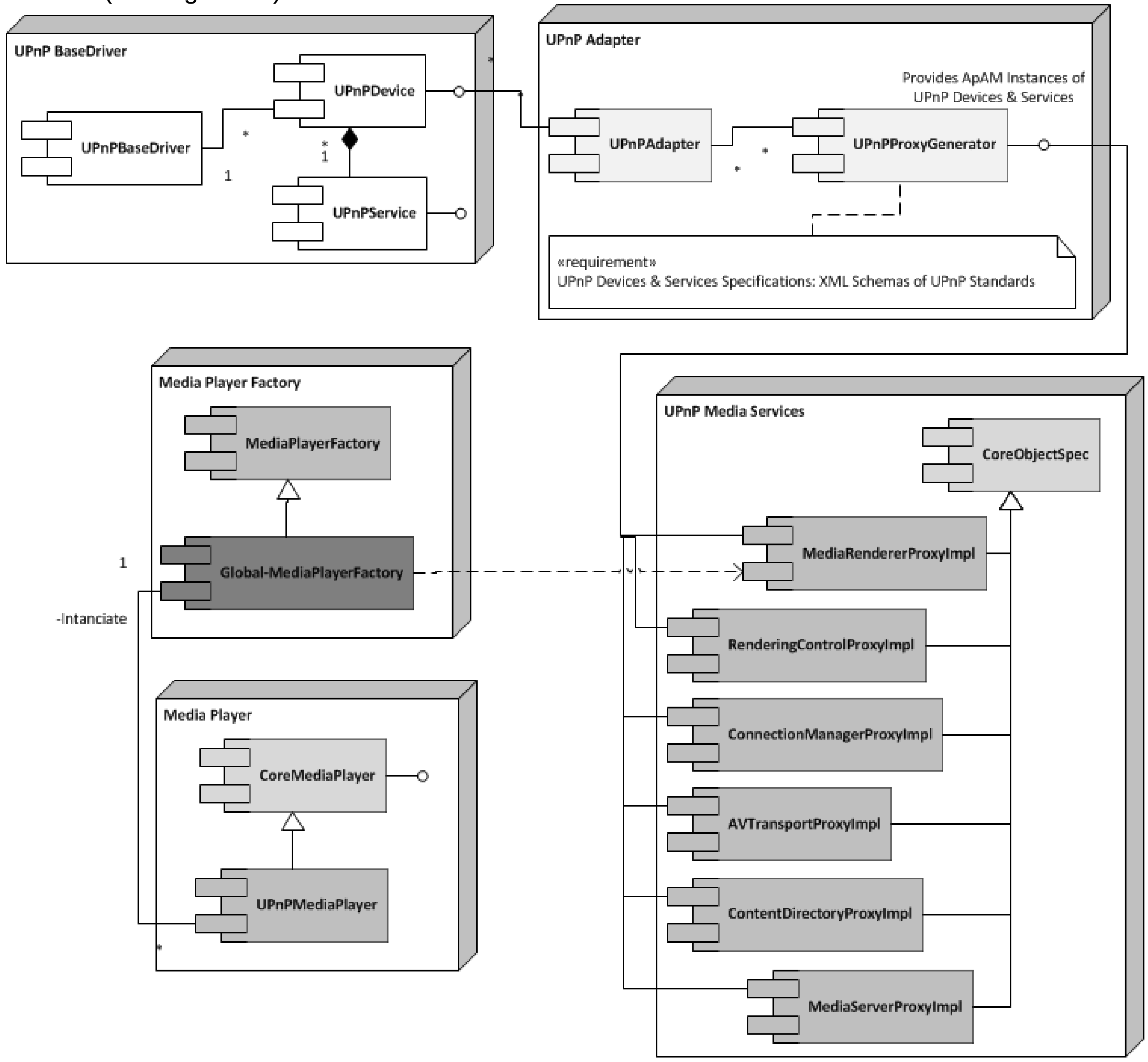


Figure 13: Architecture for the integration of UPnP devices and services.

5.3. Web services detailed architecture

The integration of Web services is based on APIs that are specific to the service providers used, such as Google and Yahoo!. According to the principles presented in Section 2, there is an adapter per Web service provider (e.g., **GoogleAdapter** in Figure 14). This adapter provides a service per Web service integrated in the HMI Middleware. For example, **GoogleAdapter** offers services: **GoogleCalendarAdapter** (see Figure 14) and other services from Google when their will be intergrated. A Web service provider adapter (such as GoogleAdapter) manages users' account connection if needed, and reifies the technical access to the services it provides (REST, RSS, etc.). According to the ApAM metamodel, an ApAM implementation of a Web service is a proxy of the remote Web service and uses OSGi mechanisms to communicate with the Web service provider adapter. An ApAM web service implementation (such as **GoogleCalendarImpl**) implements an ApAM specification (such as **CoreCalendarSpec**) that describes the web service abstraction and the notification messages it can trigger.

At this time of the writing, Google mail, Google calendar and Yahoo! Weather forecast, are integrated in the Core HMI Middleware.

The integration of Google services (Figure 14) is based on Google v2 API and uses Google data representation for the java objects handled in the GoogleAdapter. There may be several implementations for CoreMailSpec and CoreCalendarSpec, each one corresponding to a specific mail or calendar provider. The GoogleCalendarImpl is instantiated for each calendar for each user with a Google account synchronized with the AppsGate HMI Middleware. The MailGMailImpl is instantiated for each Gmail user account and uses the IMAP standard API directly (as a result, it does not use the GoogleAdapter).

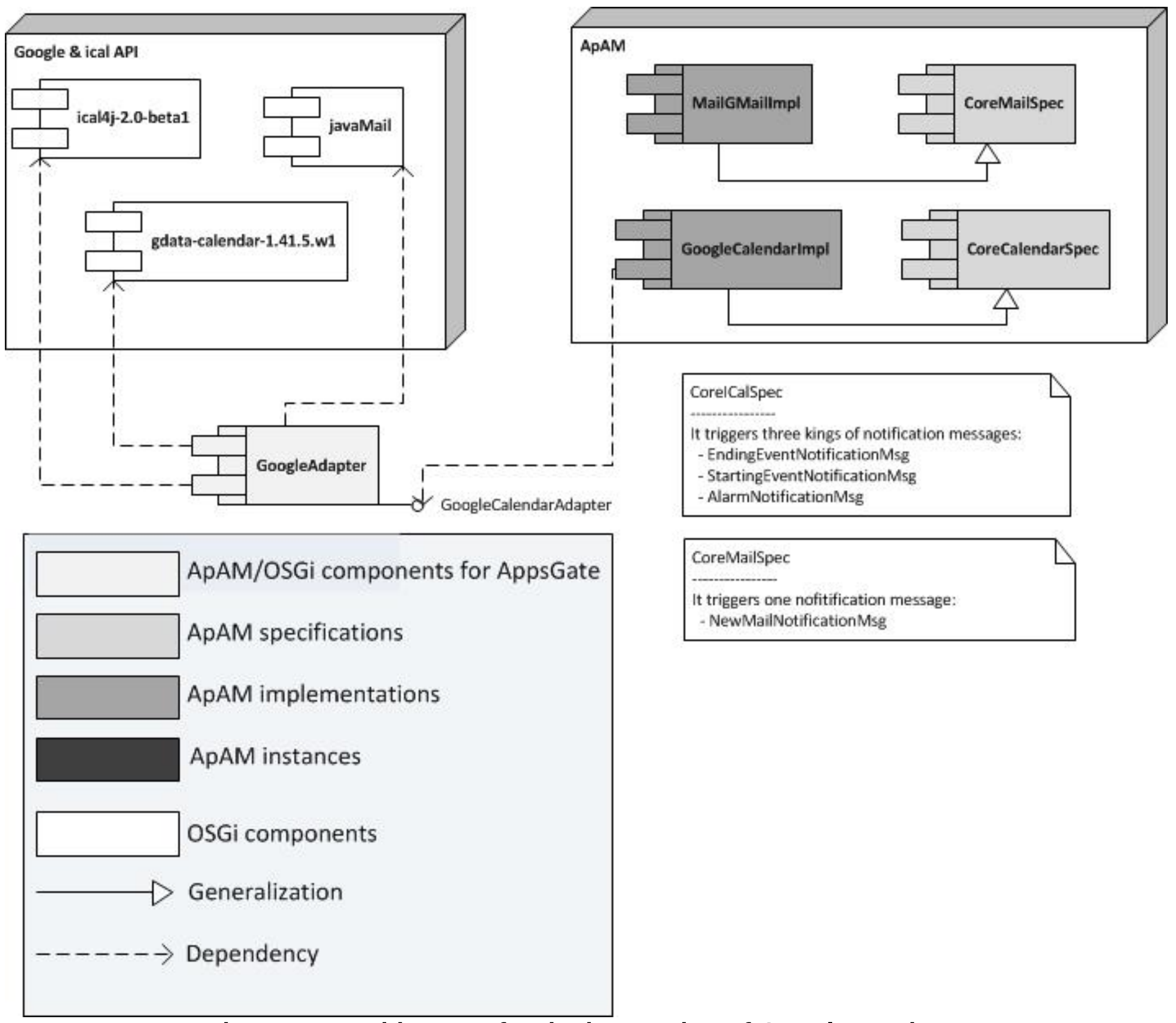


Figure 14: Architecture for the integration of Google services.

The integration of the Yahoo! weather forecast service is based on RSS stream to get weather forecast information (see Figure 15). This is the easiest way to get weather forecast information with the identification of a Yahoo account. The **YahooWeatherImpl** is a singleton that synchronizes Yahoo! weather forecast with the HMI Middleware. **CoreWeatherServiceSpec** describes the weather forecast service abstraction (getTemperature, getWeatherTrend, etc.) along with the ApAM notification messages.

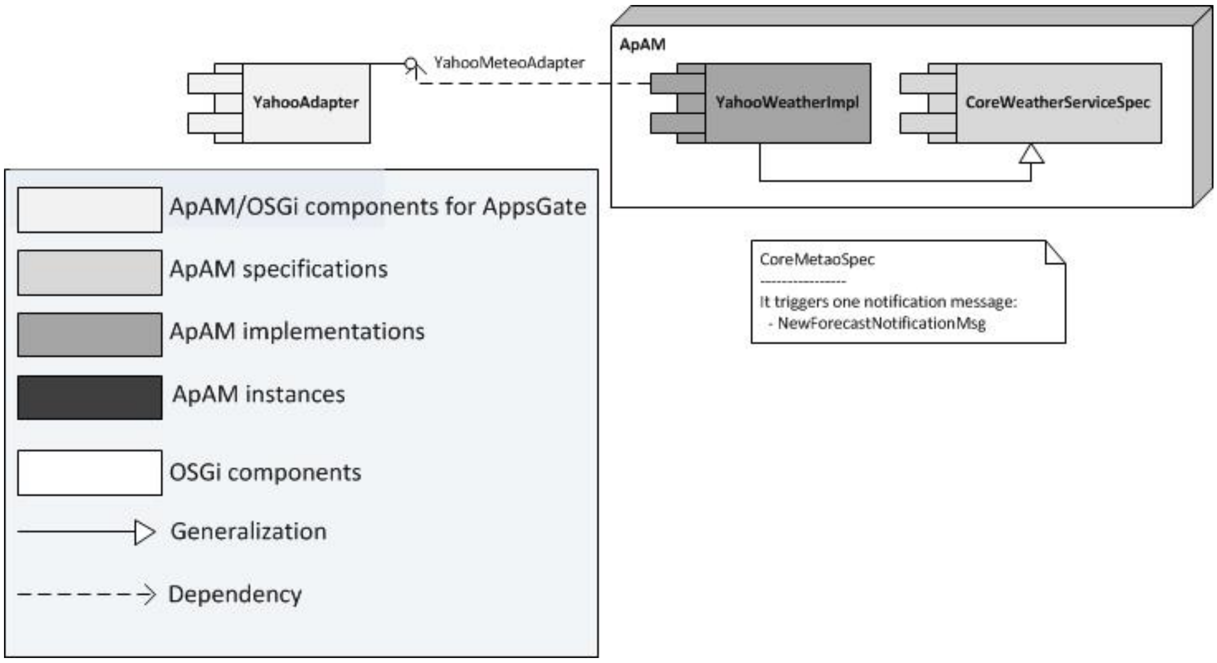


Figure 15: Architecture for the integration of Yahoo! Services.

5.4. Philips HUE lights detailed architecture

The integration of the Philips Hue lights (cf. Figure 16) is based on the REST API provided by the Philips HUE bridge. This bridge hosts a web server to receive commands as HTTP requests. Because we need the IP address of the bridge, the **PhilipsHUEAdapter** uses Cybergarage UPnP API to discover the bridge over the local area network. This adapter gets the list of PhilipsHUE lights from the bridge and, for each light, instantiates a **PhilipsHUEImpl**. The PhilipsHUEAdapter abstracts the REST API with java interface to manage the HUE network and provides a generic management for lights.

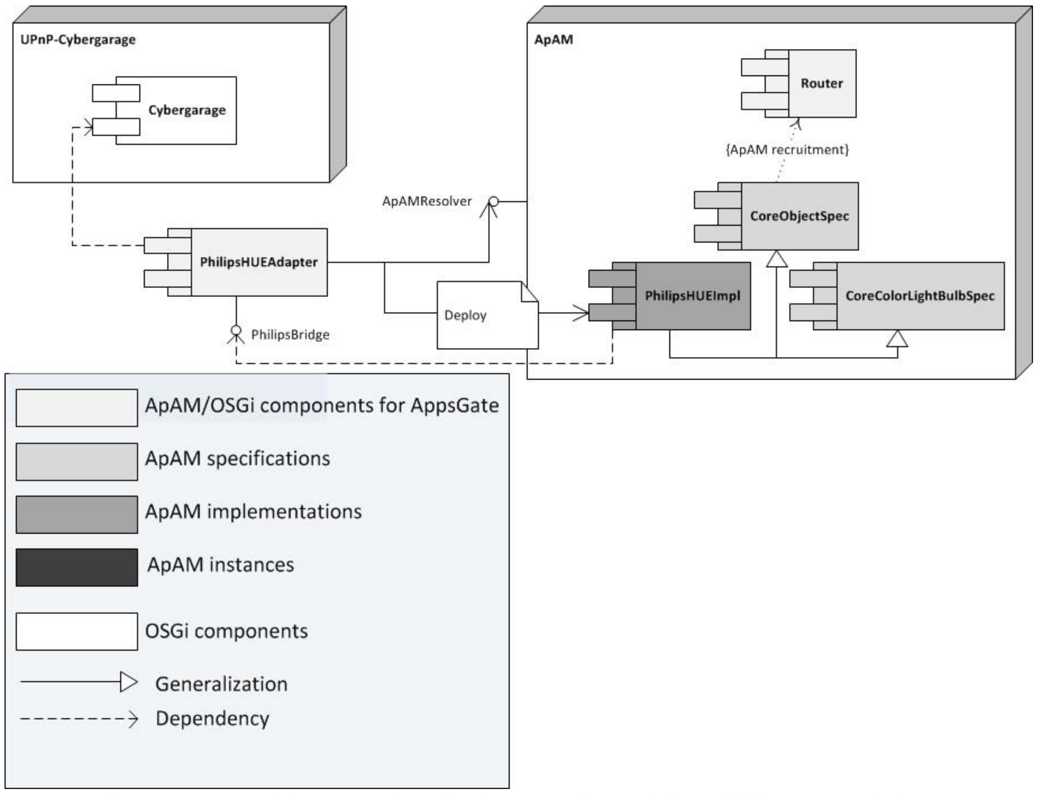


Figure 16: Architecture for the integration of the Philips HUE lights.

A **PhilipsHUEImpl** offers a java abstraction of the REST API for specific light control. In addition, it implements the **CoreColorLightBulbSpec**, which in turn is the interface for end-user to control light colors.

5.5. Watteco detailed architecture

The integration of Watteco IP sensor technology (see Figure 17) is based on a small native C executable provided by Watteco. This executable creates a network interface within the Linux operating system. This interface is a software link between the Watteco radio dongle which is the border router between the IPV6 sensor network and the Watteco Adapter for AppsGate. The border router hosts a small web server that lists all Watteco IPv6 devices detected and all routes to reach each sensor. The adapter instantiates the detected sensors with the corresponding implementation. With the Watteco technology, user can set up the behavior of its sensors (notification rate, accurancy, etc.) manually. Actually, the adapter sets up a default configuration for each sensor when instantiated.

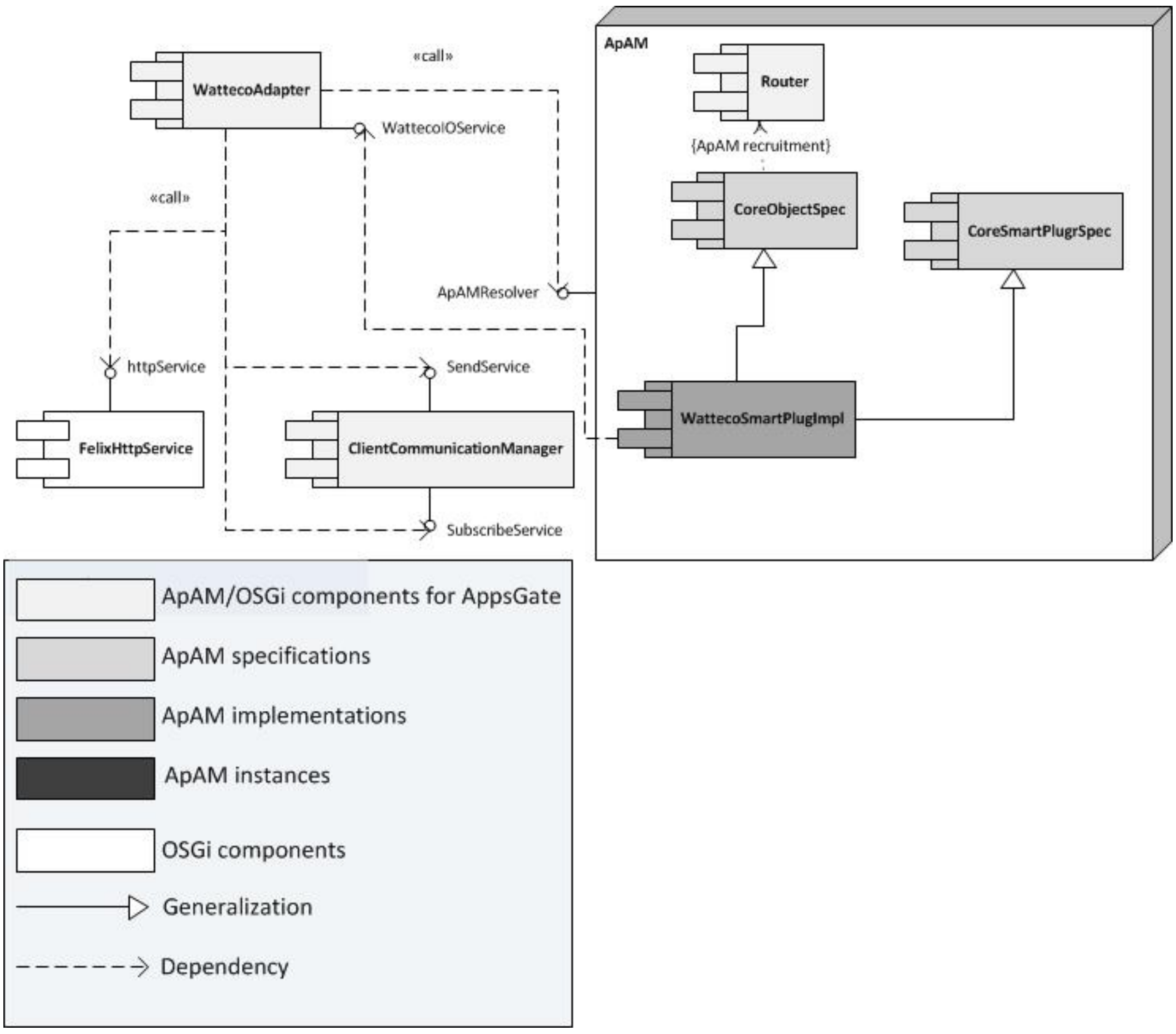


Figure 17: Architecture for the Watteco sensor technology integration.

6. Annex 2: Detailed Specification of the Components of CHMI

The **Core HMI middleware** is comprised of four types of components:

- the Core ApAM components that represent **Physical World devices and services**
- the Core ApAM components used for **Communication**,
- the Core **Appsgate OSGi bundles,**
- **External OSGi bundles** that are generic and reusable bundles developed by the OSGi community.

6.1. Core ApAM Components for Physical World devices and services

According to the ApAM metamodel, an ApAM component is decomposed into three classes: Specifications, Implementations and Instances.

6.1.1. Specifications

Specifications represent abstraction of devices or services. For instance, the CoreTemperatureSensorSpec is a specification that abstracts the concept of temperature sensor. Any service or device that provides temperature information has to implement this specification. Below is the list of the currently existing specifications along with the links to the corresponding source files (mostly java interfaces or abstract classes).

These components are defined as follows:

Component Name	Description	Programmatic Interfaces
CoreObjectSpec	Abstracts generic physical devices and services into a unique Java interface. In addition, it specifies the message format for notifications.	CoreObjectSpec.java NotificationMsg.java

Sensor specifications:

Component Name	Description	Programmatic Interfaces
CoreContactSensor Spec	The Java interface provides sensor contact status on demand. It is also possible to be notified when the sensor status changes through an ApAM message.	CoreContactSensorSpec.java ContactNotificationMsg.java
CoreIlluminationSensor Spec	The java interface provides current illumination on demand.	CoreLuminositySensorSpec.java IlluminationNotificationMsg.java
CoreKeyCardSensor Spec	The Java interface provides sensor card status on demand.	CoreKeyCardSensorSpec.java KeyCardNotificationMsg.java
CoreSwitchSensor Spec	The Java interface provides the last switch number on demand.	CoreSwitchSensorSpec.java SwitchNotificationMsg.java
CoreTemperatureSens or Spec	The Java interface provides temperature on demand.	CoreTemperatureSensorSpec.java TemperatureNotificationMsg.java
CoreUndefinedSensor Spec	Abstracts away unknown sensors. The java interface allows the user to see sensors that are not yet implemented or sensors with ambiguous pairing telegrams. This is the case for some sensors that	CoreUndefinedSensorSpec.java

	do not comply with the standard.	

Actuator specifications:

Component Name	Description	Programmatic Interfaces
CoreOnOffActuatorSpec	This specification is used for all devices that can be turned on or turned off.	CoreOnOffActuatorSpec.java OnOffActuatorNotificationMsg.java
CoreColorLightBulbSpec	This specification is used for remote controlled lamps that can change their color.	CoreColorLightSpec.java ColorLightNotificationMsg.java
CoreSmartPlugSpec	This specification is used for remote controlled switchable socket and gets the consumption.	CoreSmartPlugSpec.java SmartPlugNotificationMsg.java

Media Services specifications:

Component Name	Description	Programmatic Interfaces
CoreMediaPlayerSpec	Abstracts a Media player. This specification is used for remote controlled media player services.	MediaPlayerSpec.java

WebServices specifications:

Component Name	Description	Programmatic Interfaces
CoreCalendarSpec	This specification is used for synchronized every calendar services with AppsGate.	CoreCalendarSpec.java AlarmNotificationMsg.java EndingEventNotificationMsg.java StartingEventNotificationMsg.java
CoreWeatherServiceSpec	This specification is used for synchronized every weather services with AppsGate.	CoreWeatherServiceSpec.java WeatherUpdateNotificationMsg.java
CoreMailSpec	This specification is used for synchronized every mail services with AppsGate through IMAP protocol.	Mail.java ApamMessage.java

Other services specifications:

Component Name	Description	Programmatic Interfaces
CoreClockSpec	Abstracts the time. This specification is used for control the system time.	CoreClockSpec.java ClockAlarmNotificationMsg.java ClockSetNotificationMsg.java FlowRateNotificationMsg.java

6.1.2. Implementations

Implementations are compliant with their corresponding specifications but are adapted to the specific target technology (e.g., EnOcean, Watteco, UPnP, etc.). There may be multiple implementations for a given specification and there may be several instances of an implementation. Three target technologies are currently supported for now: EnOcean and Watteco for sensors and actuators network, and UPnP for multimedia devices.

EnOcean implementation

EnOcean ApAM implementations below correspond to the EnOcean sensors and actuators that are currently supported by the HMI middleware. Each implementation inherits from its corresponding specification class as well as from the CoreObjectSpec class so that the **Router** can reference them in a uniform manner.

Component Name	Description	Java main class
CoreEnoceanContactSensorImpl	Implementation for the CoreContactSensor specification. Instances of this implementation notify that the contact status has changed by sending an ApAM message. This means that something has been opened or closed.	EnoceanContactSensorImpl.java
CoreEnoceanLumSensorImpl	Implementation for the illumination sensor specification. It notifies that illumination has changed significantly. EnOcean illumination sensors send illumination events regularly.	EnoceanLuminositySensorImpl.java
CoreEnoceanKeyCardSensorImpl	The EnOcean key card sensor implementation allows clients to get the status of key card, i.e. whether a card is inserted or not. Clients will also be notified when a card has been inserted or taken out.	EnoceanKeyCardSensorImpl.java
CoreEnoceanSwitchSensorImpl	EnOcean switch sensors can be made up of one or two switch buttons. It can be a wall mounted switch or a mobile remote switch. This implementation allows clients to be notified when an end-user has pressed the same button twice, only one notification is sent. Notification can indicate which button is pressed and accordingly if it pressed on or off (bottom or top).	EnoceanSwitchSensorImpl.java
CoreEnoceanTempSensorImpl	EnOcean temperature sensor sends an event when the temperature has changed significantly (about 5 Celsius degrees) and every 10 minutes.	EnoceanTemperatureSensorImpl.java
CoreEnoceanOnOffActuatorImpl	This implementation allows end-users to pair every physical On/Off EnOcean actuator with a logical representation to remote control it. It will also send TurnOn or TurnOff events to the EnOcean proxy.	EnoceanOnOffAcuatorImpl.java
CoreEnoceanUndefinedSensorImpl	This implementation does not correspond to any EnOcean device profile. It is designed to have a representation for unsupported EnOcean device profiles. It offers methods to get	EnoceanUndefinedSensorImpl.java

	generic information about this kind of devices. It will be used in debug mode to indicate which device is not correctly auto-deployed or not yet implemented	

Media Services implementation

All UPnP devices and services are available as CoreObjectSpec. The components are dynamically created from the UPnP standard specifications (provided by the UPnP forum as XML schemas) and provide convenient programmatic handles. The device and each service are reified as separate components. Matching between the physical apparatus, the corresponding UPnP device and the UPnP services is done using the UpnP **Unique Device Name (UDN)**. According to specification of UPnP AV [UPnP 2002], several proxies are created as ApAM Implementation. The underlying mechanisms allow easy adaptation to newer specifications of UPnP AV Components.

Component Name	Description	Java main class
Content Directory Service (Media Server)	Enumerates the digital media contents that a device can provide to the home network. It can provide detailed information about each Content Item. This information includes properties such as its name, artist, date created, size, etc. Additionally, the information describes the transfer protocols and data formats supported.	ContentDirectoryProxyImpl.java
Rendering Control Service (Media Renderer)	Configures the rendering of the current media. For example the brightness, the contrast or the RGB color set. The rendering control process includes audio rendering with volume, balance or equalizer settings.	RenderingControlProxyImpl.java
Connection Manager Service (Media Server) (Media Renderer)	Prepare and manage multiple connections between them. It abstracts away the various kinds of streaming as well as the technology used for connection with the physical media.	ConnectionManagerProxyImpl.java
AV Transport Service (Media Server) (Media Renderer)	Control media streams (Play, Pause, etc.). It is always related to Connection Manager, which describes the A/V connection setup procedures, as well as to the Content Directory, which provides meta-information about the stored resources.	AVTransportProxyImpl.java

As these ApAM Components solely implement the **CoreObjectSpec** specification, all business logic is dedicated to the Implementation. A few services are provided to hide UPnP complexity.

Component Name	Description	Java main class
Media Player Factory	Discovers all MediaRenderer on the platform. For each of these, it creates a CoreMediaPlayer Instance. As a singleton design pattern, an unique ApAM instance named Global-MediaPlayerFactory is running at the time.	MediaPlayerFactory.java
Core Media Player	This service is an abstraction of MediaRenderer. The UPnP MediaPlayer is an Implementation of the CoreMediaPlayer specification. The MediaPlayers describes functions as play, pause, stop and set/get volume.	MediaPlayer.java MediaPlayerAdapter.java

Web Services implementation

The ApAM implementations for the Web services below correspond to the web services that are currently supported by the HMI Middleware. Web services are used in two ways: to allow the HMI middleware to gather information from users' account as for Google calendar, and to provide users with their personal information.

Component Name	Description	Java main class
Calendar Implementation	AppsGate representation of Google remote calendars. Each implementation instance is synchronized with the corresponding calendar maintained by Google in the cloud.	GoogleCalendarImpl.java
Mail Implementation	AppsGate representation of Google remote mail accounts. Each implementation instance is synchronized with the corresponding gmail account maintained by Google in the cloud.	GMailImpl.java
Weather forecast Implementation	AppsGate representation of the weather forecast provided by Yahoo!. An instance of this implementation provides the weather forecast for the three coming days.	YahooMeteoImpl.java

Philips HUE lights implementation

Component Name	Description	Java main class
Philips HUE Implementation	Specifically designed to control lights remotely using the Philips colored light bulb technology.	PhilipsHUEImpl.java

Watteco sensors implementation

Component Name	Description	Java main class
CoreWattecoSmartPlugImpl	Software integration of Watteco SmartPlugs that are also sensors and actuators. They return the active consumption of devices plugged in and they can be turned off/on remotely	WattecoSmartPlugImpl.java SmartPlugValue.java SmartPlugServices.java

Utility services implementation

Component Name	Description	Java main class
ConfigurableCoreClockImpl	Implementation of the CoreClock, allows modification of current time, control time flow…	ConfigurableCoreClockImpl.java

6.2. Core ApAM Component(s) for Communication

6.2.1. Router

As discussed above, the **Router** is in charge of routing messages between the internal ApAM components of the HMI middleware, as well as between the remote clients and the HMI middleware. In order to communicate with remote clients, method calls have to be as generic as possible. For doing so, a command from a remote client is converted by the router into generic Java calls for the corresponding ApAM instance. This transformation is based on the introspection mechanism provided by Java. The router uses the ApAM reference mechanism to get all instances that implement the AbstractObject specification.

Component Name	Description	Programmatic handles
RouterApAM (Implementation)	The router is composed of two classes: **RouterImpl** and **RouterCommandListener**. RouterImpl gets a communication service from the **Client Communication Manager** and subscribes to all possible incoming messages. Then it routes each message to the appropriate recipient. RouterCommandListener.java is the description of the listener that is connected to the **Client Communication Manager**.	RouterImpl.java RouterCommandListener.java

6.2.2. Appsgate

This component is the entry point for clients to access the features provided by the HMI middleware (also called the AppsGate server). As a result, the Appsgate component allows clients to call the methods provided by the EHMI managers (i.e., PlaceManager, DeviceName Table and EUDE Interpreter). As Router supports generic calls to the core objects of the Core HMI Middleware, similarly the AppsGate component is called by the Router using the Java introspection mechanism. AppsGate, which is an ApAM component, is represented by two bundles: a specification and an implementation.

Component Name	Description	Programmatic handles
AppsGateSpec (Specification)	This specification is the java interface that describes the methods that can be called by third parties.	AppsGateSpec.java

AppsGateImpl (Implementation)	Implements AppsGateSpec described above. The single instance of this class is also an UPnP device. It uses the UPnP discovery protocol to allow clients to discover the AppsGate server in the local area network. It offers UPnP services that allow clients to get the port and the URL of the web socket connection as well as to get the URL of the HTML welcome page.	Appsgate.java

6.2.3. Client CommunicationManager

ClientCommunicationManager is used to establish connections with remote clients through web sockets. It offers subscription and message sending services. This bundle encapsulates two groups of classes: Main class and Services.

Main class	The **ClientCommunicationManager** class manages the web socket engine registration as well as the reception of messages coming from all connected remote clients.	ClientCommunicationManager.java
Services	**SendWebsocketService** is the interface that offers methods to send messages to all connected remote clients or to a specific one. **AddListenerService** allows components to get a subscription service in order to be notified when a message comes in. Any component can subscribe for configuration messages, command messages or to both with listeners subscription.	SendWebsocketsService.java AddListenerService.java CommandListener.java ConfigListener.java

6.3. Core AppsGate OSGi bundles

The Core AppsGate OSGi bundles are pure OSGi/iPojo components. Either they are technological connectors or they offer services to manage specific resources like databases, web socket connections, web server resources, etc.

6.3.1. Ubikit Adapter

This is the EnOcean Ubikit adapter. It bridges the gap between EnOcean Ubikit and the ApAM Core World. This bundle encapsulates three groups of classes.

Main classes	UbikitAdapter class is the main class of the EnOcean connector. This class inits all the resources needed, gets the Ubikit services and, finally subscribes to a communication service. This class deploys an HTML GUI to handle the EnOcean pairing phase. The EnOceanProfiles class is the link between formal EnOcean profiles and their corresponding ApAM implementations. The EnOceanConfigListener class is the listener associated to the communication service to get configuration commands from the web GUI.	UbikitAdapter.java EnOceanProfiles.java EnOceanConfigListener.java

Services	This group includes two services. The EnOceanPairingService allows any other component to change the Ubikit pairing mode. The EnOceanService is used to interact with saved Ubikit items for loading sensor configurations. It is also used to trigger actions for actuators.	EnOceanPairingService.java EnOceanService.java
Events	These classes are listeners for Ubikit specific events. They are used to get events from the EnOcean wireless sensor network or to send events to Ubikit services.	ContactEvent.java KeyCardEvent.java LumEvent.java MotionEvent.java PairingModeEvent.java SetPointEvent.java SwitchEvent.java TempEvent.java

6.3.2. Watteco Adapter

The Watteco Adapter bridges the gap between IPv6 network interface for Watteco sensor network and Appsgate CHMI.

Main classes	The **WattecoAdapter** class searches for a Watteco dongle and launches the network interface supported by the native Watteco configuration executable. It instantiates all Watteco implementations that correspond to the sensors discovered in the network. The **BorderRouter** class wraps the serial communication between **WattecoAdapter** and the Watteco USB dongle.	WattecoAdapter.java BorderRouter.java WattecoConfigListener.java
Services	The **WattecoDiscoveryService** allows any other component or user to get new sensors from the border router. The **WattecoIOService** is used to interact with sensors generically. It is also used to trigger actions for actuators, send configuration to sensors or retrieved data. Finally the **WattecoTunSlipManagment** is used to follow SLIP tunnel state, to start it or stop it according to system needs.	WattecoDiscoveryService.java WattecoIOService.java WattecoTunSlipManagement.java

6.3.3. MediaServices

This bundle contains a library of classes to be used at runtime by the UPnPAdapter. It contains the generated UPnP Media Service and Devices generated from the UPnP Specifications (provided by the UPnP Forum as XML Schemas). These components are proxies that expose all the capabilities provided by the real UPnP stuff.

Main classes	This set of classes is responsible for generating the java code and the associated xml descriptor.	DeviceProxyGenerator.java MetadataGenerator.java

Schemas		AVTransport.1.xml ConnectionManager.1.xml ContentDirectory.1.xml RenderingControl.1.xml
Service classes (generated)	For each of the schemas, a java class is generated, along with an xml descriptor of the ApAM implementation. This component is relative to the **CoreObjectSpec** specification.	AVTransportProxyImpl ConnectionManagerProxyImpl ContentDirectoryProxyImpl RenderingControlProxyImpl
Device classes (generated)	According to these services, two other implementations are created, also relative to the **CoreObjectSpec** specification.	MediaRendererProxyImpl MediaServerProxyImpl

6.3.4. UPnP Adapter

The ApAM_UPnP_Discovery bundle is in charge of (1) getting the UPnP devices that are published by the UPnP base driver and registered in the OSGi registry and (2) instantiating the proper ApAM implementation from the MediaServices bundle.

Main class	Listens for UPnPDevice service discovery events (from UPnP base driver) and creates the APAM proxy for all hosted services in the device.	ProxyDiscovery.java

6.3.5. GoogleAdapter

This is the Google proxy adaptor. It bridges the gap between the Google cloud services and the AppsGate Core World. This bundle encapsulates two groups of classes.

Main class	Manages the connections with gmail account credentials to the google REST service. It is used to encapsulate REST requests and responses as java objects.	GoogleAdapter.java
Services	CalendarAdapter service is a java interface that offers methods to send REST requests from AppsGate to the Google calendar REST service.	GoogleCalendarAdapter.java

6.3.6. PhilipsHUEAdapter

This is the Philips HUE adapter for Philips REST API. It is designed to find a Philips HUE bridge over the UPnP discovery protocol and encapsulates REST commands as Java method calls.

Main class	Manages the communication between the AppsGate PhilipsHUE implementation and the physical light bulbs. This class creates an instance of PhilipsHUE implementation each time a new light is detected in the local network.	PhilipsHUEAdapter.java PhilipsBridgeUPnPFinder.java
Services	Exposes the methods to use the features offered by the Philips HUE Adapter. All Philips HUE implementations use this interface to call the	PhilipsHUEServices.java

	methods provided by the Philips bridge.	

6.4. External OSGi bundles

The external OSGi bundles, either are provided by the AppsGate partners (e.g., the Ubikit bundles), or belong to the Open Software community. In other words, UJF/LIG is not the project owner, nor the project manager of these bundles but depends on them.

Bundle Name	Description
Grizzly httpservice bundle 2.2.8	This bundle provides the web socket engine and services over Felix httpservices.
Mongo java driver 2.10.1	This bundle provides services to manage a Mongo database through a Java driver. It is used by the ApAM PropertyHistoryManager to log state changes of the Core World.
Felix http bundle 2.2.0	This bundle provides a full http server and all services to deploy web resources, servlets or web socket engine.
Felix upnp basedriver 0.8.0 Felix upnp extra 0.4.0 Felix upnp devicegen util 0.1.0	These bundles offer the UPnP base driver and services to get UPnP devices and services interface dependency.
Pem enocean 1.6.0 rxtx4osgi 1.0.2 Ubikit 1.9.0	These Ubikit bundles are provided by Immotronic to support EnOcean compliant devices.

7.Annex 3: Specifications of SPOK, the AppsGate EUDE

In its current version, SPOK includes an editor and an interpreter. Whereas the interpreter is a component of the Extended HMI Middleware, the editor is a client of the AppsGate server. Whereas the AppsGate server runs (or will run) on an AppsGate Set-Top-Box (STB) platform, AppsGate clients are supposed to run on Windows, Mac OS X, and GNU/Linux as well as on mobile operating systems, typically iOS and Android. At the time of this writing, AppsGate clients run on iOS and Android. These clients can also be accessed by most web browsers, but without device-specific capabilities.

An AppsGate client is developed in HTML5 and JavaScript and uses the following libraries:
-**RequireJS** (http://requirejs.org). RequireJS is a JavaScript file and module loader that optimizes memory management by limiting loading to the modules that are needed.
-**Backbone.js** (http://backbonejs.org). Backbone.js is used to structure the AppsGate client in a similar fashion to the MVC architectural design pattern so as to achieve separation of concerns, therefore minimizing dependencies between the AppsGate client components.
-**Underscore** (http://underscorejs.org). Underscore is used as a *utility-belt* library. For instance, Underscore enriches the standard JavaScript array with methods to sort, filter or search array elements. It comes as a dependency of Backbone.js.
-**jQuery** (http://jquery.com). jQuery is used to manage DOM manipulations, animations and event bindings. It comes as a dependency of Backbone.js.
-**Bootstrap** (http://getbootstrap.com). Bootstrap is a web front-end framework that provides UI components. It also facilitates the implementation of responsive layouts as required for AppsGate since the targeted client platforms may have very different screen resolution and size.
-**PEG.js** (http://pegjs.majda.cz). PEG.js, formally Parser Generator for JavaScript, is a library used to generate the parser for the End-User Programming editor.
-**Moment.js** (http://momentjs.com). Moment.js is a library to handle and format dates.

-**i18next** (http://i18next.com). i18next is a library to support internationalization. It allows an AppsGate client to be translated into several languages. At the moment of this writing, French and English are supported.
-**PhoneGap** (http://phonegap.com). PhoneGap is used to create hybrid web AppsGate clients for mobile platforms (smartphones and tablets). It enhances the standard JavaScript API with the capacities that are specific to the mobile device used and supports most mobile Operating Systems (e.g., Android, iOS, Windows Phone, Blackberry).

PhoneGap is a web framework to build hybrid mobile AppsGate clients using standard web technologies. In this context, a *hybrid AppsGate client* is an AppsGate client that mixes native capacities of the target platform with web components executed in a web renderer. PhoneGap provides JavaScript API that provides the developer with access to the capacities of the platform such as camera, compass, and notifications. To compile the AppsGate client for a specific target, the developer creates a PhoneGap project and embeds the source code before compiling

The next two sections respectively present an overview of the global architecture of an AppsGate client since the SPOK editor is a client of the AppsGate server followed by a detailed description of the communication mechanisms on which the SPOK editor relies. We then present the two components of SPOK: the editor and the interperpreter.

7.1. Architecture Overview

As shown in Figure 18, an AppsGate client is structured as four layers: the network layer called Communicator, the event dispatcher (Dispatcher), the AppsGate client router AppRouter, and a set of modules where one module corresponds to the classes and functions that are necessary to support user-centered domain-dependent concepts. There are currently three such concepts: devices, places, and programs.

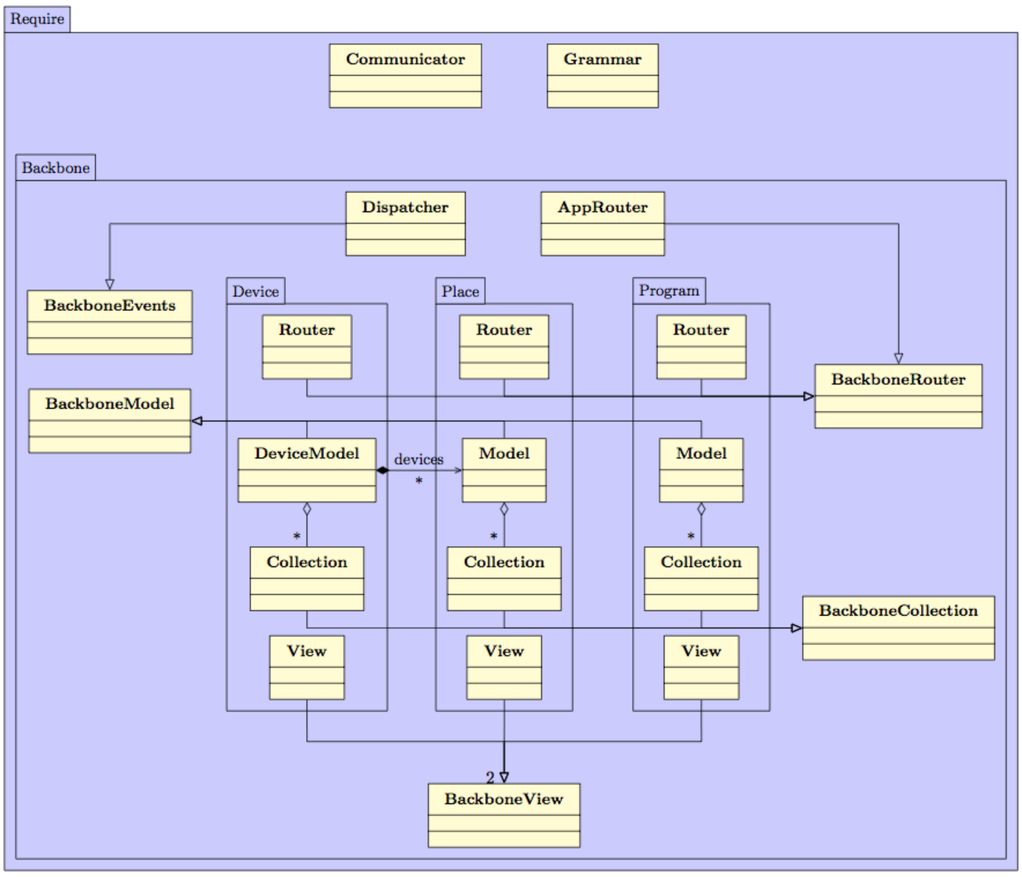


Figure 18. Representation of the global architecture of an AppsGate client.

7.1.1. Network layer Communication: Communicator

Communicator handles the communication between an AppsGate client and the AppsGate server. It is based on WebSockets and provides the AppsGate client with high-level methods to communicate with the server using the JSON serialization format. To send a message to the server, Communicator supplies the client with a method that formats data according to the AppsGate client-server protocol. When a message is received from the server, Communicator transforms the JSON message into the appropriate event by the way of Dispatcher, the event dispatcher.

Implementation: communicator.js

7.1.2. The Event dispatcher: Dispatcher

Dispatcher, which is based on *Backbone.Events*, provides a flexible mechanism to support communication between the components of an AppsGate client. It triggers AppsGate client-wise events. Any component of the AppsGate client can subscribe to any event triggered by the event dispatcher. Typically, when Communicator receives updates from the AppsGate server, it uses Dispatcher to trigger events identified by the component's Ids slot specified in the event. As Dispatcher allows any component to subscribe to events whose Id matches its Id, only the relevant component receives the information.
Dispatcher is used as well in the Start sequence as described in section 4.1, to ensure that all needed data is received before the AppsGate client can run

Implementation: app.js:102

7.1.3.The AppsGate client Router: AppRouter

AppRouter stores the state of the AppsGate client and manages the event subscriptions expressed by Views (more on Views, below). AppRouter also handles the navigation between the different components through a set of routes that match the client-side pages by triggering the rendering of the correct page depending on the component the user sets the focus on. To achieve this, when changing the focus (i.e. when switching from the current page to a new one), AppRouter performs the following actions:
-Remove the View and all the events that were bound to that View
-Replace the View with the new one
-Trigger the View rendering, attaching the events and making it current

Implementation: app.js:15

7.1.4. Modules

A module is a logical component of the AppsGate client that models and represents "user-centered domain-dependent" concepts. At the time of this writing, three modules are available: Device, Place and Program.

A module is compliant with the Backbone MVC-like architectural pattern as well as with the RequireJS Asynchronous Model Definition (AMD) pattern. Adopting this organization has several advantages. First, it ensures low coupling between the functional core of the "user-centered domain-dependent" concepts and their concrete representations made available to users. Second, memory resource is optimized since modules are loaded and unloaded dynamically according to the status of the AppsGate client. This feature is important for mobile devices that have limited memory resource.

In Backbone, a Model corresponds to the data of a "user-centered domain-dependent" concept, for example, that of a Device. A Collection is an ordered set of Models. For each Module *X*, a Collection is instantiated to contain the Models that correspond to the same class *X* of entities detected/handled by the AppsGate server. For example, the Device module includes one Collection instance named *Device.Collection* that contains *n Device.Model* if *n* devices have been detected by the AppsGate server. A View is a concrete representation of the state of a Model that is made available to the end-user for observation and manipulation. For example, in the case of the Device module, this representation is an HTML5 document generated by the *Device.View* using the *render* method. This method uses predefined HTML5 templates to generate this document.

Communication within and across modules relies on Backbone events. When a Model is updated, the Model and the Collection it belongs to, fire a *change event*. By doing so, a state change can be triggered at different levels of granularity (i.e. at the collection level, and/or at the model level individually). A *Device.View* listens to events that occur from the Model it is interested in, as well as from UI elements of the generated HTML5 document, such as buttons and inputs. Modifications of the document are then translated by the View into modifications of the Model it corresponds to.

In order to synchronize the state of a Module with the state maintained by the AppsGate server, relevant Backbone methods are called by a Model when it is modified, such as *save*() or *destroy*() which in turn fire a Backbone *sync event*. The Model then synchronizes with the server, updating the objects represented by the Model on the AppsGate server side. As a result, all the specific logic to communicate with the AppsGate server is implemented in Models. This logic also fires a Backbone.js *change event* that can be caught by the corresponding Views and/or Collection.

Device module: device.js
Place module: place.js
Program module: program.js

7.2. Communication details

This section details event routing within an AppsGate client and with the AppsGate server.

7.2.1.Start sequence

Figure 19 shows the operations performed when launching an AppsGate client. When started, the AppsGate client displays a splash screen to prevent the user from using the AppsGate client before it is ready. Then, it sends three requests to the AppsGate server to retrieve the devices, places and programs that currently exist.

The AppsGate server returns the three lists. (NB: the responses are asynchronous and can be received in any order). On receiving a list, the corresponding Models are instantiated and group into the Collection that correspond to their type (either Device, Places, or Programs).

Once devices, places and programs are initialized, the AppsGate client is ready for use, and the splash screen is removed.

7.2.2. Devices, places and programs notification

The sequence diagram in Figure 20 shows how a device-related notification from the AppsGate server is processed by an AppsGate client. In this example, the AppsGate server notifies the AppsGate client that the name of the device *id* has the value *value*.

As explained above, the notification is expressed as a JSON message received by the Communicator via a WebSocket. The Communicator translates the message into a Backbone event that is then received by the Dispatcher. The *id* element of the event allows the Dispatcher to identify the Model concerned by the notification. The Model updates its state accordingly (here, sets the new name), uses the Backbone *change* event to express its state change. Its Collection as well as all the Views that represent the Model are notified of an update through the Backbone *change event*.

Figure 22, and Figure 23, respectively illustrates the communication between the AppsGate client components when creating, updating and removing a place.

The sequence diagrams of Figure 24, Figure 25, and Figure 26, respectively illustrates the communication between the AppsGate client components when creating, updating and removing a program.

7.3. The End-User Development Environment: SPOK

SPOK (Simple PrOgramming Kit) is the AppsGate End-User Development Environment (EUDE). Ideally, an EUDE includes the functions that are necessary and sufficient for developing, testing, debugging, reusing, sharing, and maintaining programs. The particularity of a EUDE is that end-users are not computer science professionals. Their goal is to develop programs to simplify their daily activities with the help of the technology, not to learn programming and software engineering techniques.

At the time of this writing, SPOK is comprised of an editor for building and modifying end-user programs, and of an interpreter for running end-user programs. The interpreter runs on the AppsGate server whereas the editor runs on a client end-user device. These are presented next. We start with the grammar as it is shared by the editor and the interpreter.

7.3.1. The SPOK Grammar

The language used by end-users to express their programs is a pseudo-natural language using the rule-based programming paradigm. The left hand side of a rule is composed of events and conditions, and the right hand side specifies the actions to be taken when the left hand-side is true or becomes true. A program may include several rules that can be executed either in parallel or sequentially.

The BNF of the grammar currently implemented is the following:

```
<nodeProgram>::=<programName> Param ( <seqParameters> ) Written By <username>[ for <username> ]
Definition <seqDefinitions> Body <seqRules>
<seqParameters>::= <parameter> { , <parameter> }
<parameter>::= <type> <varName>
<seqDefinitions>::= <definition> { , <definition> }
<definition>::= <varName> = <selector>
<seqRules>::= <seqAndRules> { <opThenRule> <seqAndRules> }
<seqAndRules>::= <rule> { <opAndRule> <rule> }
<rule>::= <nodeWhen> | <nodeIf> | <nodeAction> | ( <seqRules> )
<nodeWhen>::= when ( <seqEvents> [ <opEventBool> <expBool> ] ) then <seqRules>
<nodeIf>::= if ( <expBool> ) then <seqRules> else <seqRules>
<nodeAction>::= <actionProgram> | actions on devices, web services, etc.
<seqEvents>::= <event> { <opEvent> <event> }
<event>::= start of a program | end of a program |
activation of a program |
disactivation of a program |
events on devices and services
<expBool>::= <seqAndBool> { <opOrBool> <seqAndBool> }
<seqAndBool>::= <relation> { <opAndBool> <relation> }
<relation>::= <expression> { <opComparison> <expression> }
<opThenRule>::= then
<opAndRule>::= and
<opEvent>::= and
<opOrBool>::= or
<opAndBool>::= and
```

```
<opAdd>::= + | -
<opMult>::= * | /
```

7.3.2. The SPOK Interpreter

The SPOK interpreter is an ApAM component that runs on the AppsGate server. It manages a list of end-user programs and exposes the methods to add, remove, update, start and stop end-user programs.

The interpreter manages end-user programs using a hash map where the key is the end-user program *id* and the value is the abstract syntax tree (AST) of the end-user program. ASTs are stored in the JSON text format as the result of the compilation performed by the editor on the client side. At the time of this writing, the interpreter supposes that ASTs are semantically correct (i.e. at this point, no semantic error checking is performed).

The root node of an AST is a node whose type is NodeProgram. The children of a NodeProgram are of type NodeSeqRules. The types of the children of a NodeSeqRules include NodeIf, NodeWhen, etc., as well as NodeSeqAndRules. The children of a NodeSeqAndRules node are processed in parallel whereas the children of a NodeSeqRules are processed sequentially.

More precisely, a node can be one of the following types:

- *NodeProgram* denotes the root node of an AST.
- *NodeSeqRules* children are processed sequentially.
- *NodeWhen* processes the NodeSeqRules child on the occurrence of the specific events described by the NodeSeqEvent child.
- *NodeAction*: is a call to a device (switch a lamp on or off), or a call to another program.
- *NodeIf* evaluates the NodeExpBool, then, if NodeExpBool returns *'true'*, processes the first left most NodeSeqRules child or, if the NodeExpBool returns *'false'*, processes the right most NodeSeqRules child (if it exists).

Interpreter specification: EUDE_InterpreterSpec.java
Interpreter implementation: EUDE_InterpreterImpl.java

In the rest of this section we detail the JSON representation for the types of nodes used to represent SPOK Abstract Syntax Trees.

JSON format of a NodeProgram

```
{
"id": 'id',
"runningState": 'runningState',
"userInputSource": 'userInputSource',
"source": { source },
"programName": 'name',
"author": 'author',
"target": 'target',
"daemon": 'daemon'
}
```

Where:

- *id* is the program ID
- *runningState* is the state of the program. Its value is set by the interpreter.
- *userInputSource* is the HTML code used to express the concrete syntax of the program as seen and manipulated by the end-user. It is stored by the interpreter to avoid regeneration by the editor on the client side.
- *source* is the AST of the program. It is a JSON object.
- *author* is the user's name who has created the program. Not used at the time of this writing.
- *target* is the name of the targeted user. Not used at the time of this writing.

- *daemon* defines the strategy that the interpreter needs to apply on program termination. If the value of this field is *'true'* then the interpreter launches a new execution of the program (the program loops). Otherwise, the program ends.

Implementation: NodeProgram.java

JSON format of a NodeAction

```
{
"targetType": 'targetType',
"targetId": 'id',
"methodName": 'methodName',
"args: [ args ]
}
```

Where:

- *targetType* defines the type of the object on which the method call will be performed. Possible values are *program* and *device.*
- *targetId* is the ID of the targeted object. It can be a device ID or a program ID.
- *methodName* is the name of the method to be called on the object.
- *Args* is an array of arguments to pass to the method. Each argument has the form *{ type : 'type', value : 'value' }*. If the method does not take any argument, the array is empty.

Implementation: NodeAction.java

JSON format of a NodeSeqRules

The JSON representation of a *NodeSeqRules* is a JSON array where each entry is a *NodeSeqAndRules*. At the time of this writing, in the concrete syntax, each entry of this array is separated by the keyword *then*.
During interpretation, a *NodeSeqRules* launches the nodes *NodeSeqAndRules* sequentially. This ensures the priority of the *and* operator (which expresses parallelism) between rules over the *then*.

Implementation: NodeSeqRules.java

JSON format of a NodeSeqAndRules

The JSON representation of a *NodeSeqAndRules* is a JSON array. Each entry of the array corresponds to a node.
At the interpretation, a *NodeSeqAndRules* launches the interpretation of all its children in parallel. In the current concrete syntax, it shows as a suite of rules separated by the keyword *and*.

Implementation: NodeSeqAndRules.java

JSON format of a NodeIf

```
{
"type": 'NodeIf',
"expBool": expBoolJSON,
"seqRulesTrue": seqRulesTrue,
"seqRulesFalse": seqRulesFalse
}
```

Where:

- *Type* has always the value *'NodeIf'*. Used for the instantiation.
- *expBool* is the boolean JSON expression to be evaluated.
- *seqRulesTrue* is the sequence of rules to evaluate if the boolean expression has returned *true*. It is a *NodeSeqRules* in its JSON representation.
- *seqRulesFalse* is the sequence of rules to evaluate if the boolean expression has returned *false*. It is a *NodeSeqRules* in its JSON representation. If there is nothing to interpret it corresponds to an empty JSON array.

Implementation: NodeIf.java

JSON format of a NodeExpBool

The JSON representation of a *NodeExpBool* is a JSON array where each entry is a JSON representation of *NodeSeqAndBool*. During the interpretation, the *NodeExpBool* launches the interpretation of each *NodeSeqAndBool*, waits for their results and performs a boolean OR on the results. This gives the final result of the node to be returned to its parent.

Implementation: NodeExpBool.java

JSON format of a NodeSeqAndBool

The JSON representation of a *NodeSeqAndBool* is a JSON array where each entry is a JSON representation *of NodeRelationBool*. During the interpretation, the *NodeSeqAndBool* launches the interpretation of each *NodeRelationBool*, waits for the results and performs a boolean AND on the results. Since a *NodeSeqAndBool* is always a child of a *NodeExpBool*, it ensures the priority of the AND over the OR.

Implementation: NodeSeqAndBool.java

JSON format of a NodeRelationBool

```
{
"type": 'NodeRelationBool',
"operator": 'operator',
"leftOperand": leftOperand,
"rightOperand": rightOperand
}
```

Where:

- *type* has always the value *'NodeRelationBool'*. Used for instantiation.
- *operator* is the operator to use to compare the two terms. The operators ==, *!*=, <= and >= are supported. The type of the two terms has to be identical otherwise the *NodeRelationBool* returns *false*.
- *leftOperand* and *rightOperand* can be either the state of a device or a term (described in the paragraph 5.2.9). In the case of a state, the JSON representation of the *NodeRelationBool* is described in the paragraph 5.2.10.

Implementation: NodeRelationBool.java

JSON format of a NodeTerm

```
{
"type": 'type',
"value": 'value'
}
```

Where:

- *type* can be *'string'*, *'number'* or *'boolean'*.
- *value* is the value of the term stored in a string.

JSON format of a Device state

JSON format of a *NodeRelationBool* in the case of a device state on the left operand to evaluate:

```
{
"type": 'NodeRelationBool',
"operator": 'operator',
"leftOperand": {
"targetType": 'device',
"targetId": 'id',
"methodName": 'method',
"args": [ args ]
},
"rightOperand": rightOperand
}
```

JSON format of a NodeWhen

```
{
"type": 'NodeWhen',
"events": seqEvents,
"seqRulesThen": seqRules
}
```

Where:

- *type* has always the value *'NodeWhen'*. Used for instantiation.
- *seqEvents* is a *NodeSeqEvent* stored in its JSON format.
- *seqRulesThen* is the sequence of rules to interpret when all the events have been caught. It is a *NodeSeqRules* in its JSON representation.

Implementation: NodeWhen.java

JSON format of a NodeSeqEvent

The JSON representation of a *NodeSeqEvent* is a JSON array where each entry is the JSON representation of a *NodeEvent*. During the interpretation, the *NodeSeqEvent* launches the interpretation of each *NodeEvent* and waits for the termination. When all *NodeEvent* have finished, the *NodeSeqEvent* fires its end event.

Implementation: NodeSeqEvent.java

JSON format of a NodeEvent

```
{
"type": 'NodeEvent',
"sourceType": 'sourceType',
"sourceId": 'id',
"eventName": 'eventName',
"eventValue": 'eventValue'
}
```

Where:

- *type* is always *'NodeEvent'*. Used for instantiation.
- *sourceType* can be *'device'* or *'program'*. Used for interpretation.
- *sourceId* is the device or program ID to listen to.
- *eventName* is the name of the event to listen to.
- *eventValue* is the value to wait for to validate the node. If the node is not interested in a particular value, *eventValue* is an empty string.

Implementation: NodeEvent.java

7.3.3. The SPOK Editor

In order to support a dynamically extensible grammar as well as to provide end-users with feedforward at the user interface of the editor, the grammar used by the editor is split into 2 parts: the root grammar and the device specific grammars.

The root grammar specifies the generic structures of an end-user program: loops, conditions, etc. It is described in: grammar.peg. The device specific grammars are separated from the root grammar to be able to dynamically build the final grammar to be compliant with what is currently installed and detected by the AppsGate server.

Each device type brings with it its own events, status and actions. These grammatical elements are injected into the root grammar when generating the parser and for compiling end-user programs. At the time of this writing, the device specific grammars are stored on the client side at the beginning of the file device.js under the global variable *deviceTypesGrammar*. At a later stage of the development process, these grammars should be stored on the AppsGate server and sent to the client on demand.

Parser generation

The goal of the parser is to compile end-user programs into the JSON format described in the previous section. As mentioned in Section 1, the parser relies on PEG.js. PEG.js takes a grammar as input expressed in the format described on PEG online documentation (Grammar format for PEG.js).

The final grammar is built in grammar.js according to the following steps:

- -For each detected device type, retrieve the events, state and actions that are stored in the JSON attributes *eventAnchor*, *statusAnchor* and *actionAnchor*.
- -Inject the final list of events, state and action into the root grammar.
- -Generate the parser thanks to PEG.js.

Then, the grammar is updated on receipt of the following events:
-Device renaming
-New device notification
-Remove device notification
-New program
-Remove program

User support and compilation

The end-user development environment is implemented as Module program.js. End-users enter syntactic units by pressing any button of a soft keyboard shown on the screen. The set of buttons and their label correspond to the terminals of the grammar. To achieve this, the end-user program under construction is analyzed by the parser after each button press. Two situations may occur:

- -**The program is not syntactically correct.** The parser throws an exception that specifies the set of the next possible correct terminals. The exception is caught by the editor and the set of buttons is updated according to the possibilities described in the exception.
- -**The program is syntactically correct.** The parser has succeeded to build the AST of the program. The AST is stored in a JSON format, ready to be sent to the AppsGate server. To be able to show the next possibilities in this case, an error is inserted at the end of the program and a new compilation is launched. Now, an exception is thrown with the possibilities that are expected next.

Figure 19. UML sequence diagram for the initialization of an AppsGate client.

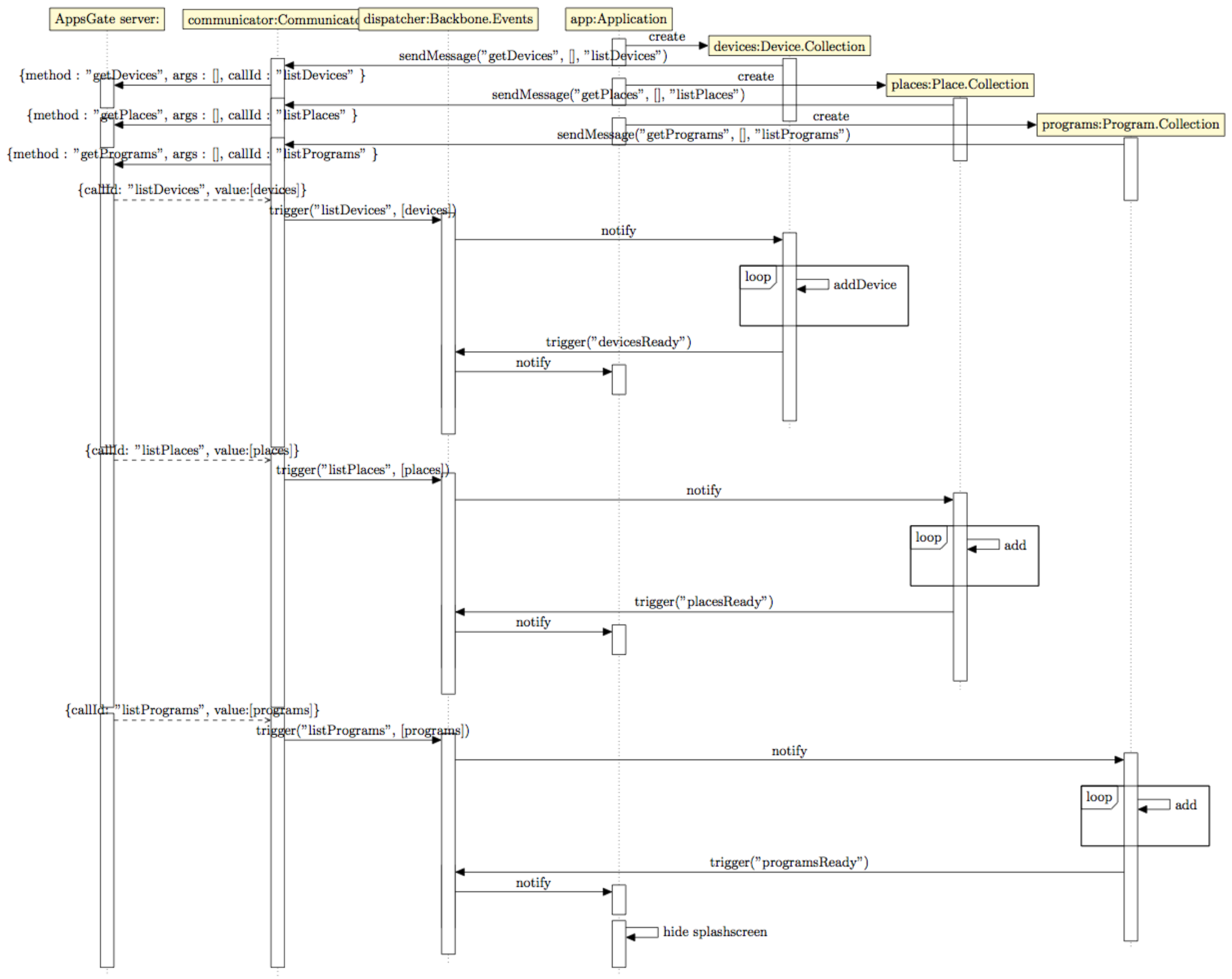

AppsGate server:
communicator:Communicator
dispatcher:Backbone.Events
app:Application
create
devices:Device.Collection
sendMessage("getDevices", [], "listDevices")
{method : "getDevices", args : [], callId : "listDevices" }
create
places:Place.Collection
sendMessage("getPlaces", [], "listPlaces")
{method : "getPlaces", args : [], callId : "listPlaces" }
create
programs:Program.Collection
sendMessage("getPrograms", [], "listPrograms")
{method : "getPrograms", args : [], callId : "listPrograms" }
{callId: "listDevices", value:[devices]}
trigger("listDevices", [devices])
notify
loop
addDevice
trigger("devicesReady")
notify
{callId: "listPlaces", value:[places]}
trigger("listPlaces", [places])
notify
loop
add
trigger("placesReady")
notify
{callId: "listPrograms", value:[programs]}
trigger("listPrograms", [programs])
notify
loop
add
trigger("programsReady")
notify
hide splashscreen

Figure 20. UML sequence diagram for a device-related notification.

(Here, new value for the name of a device).

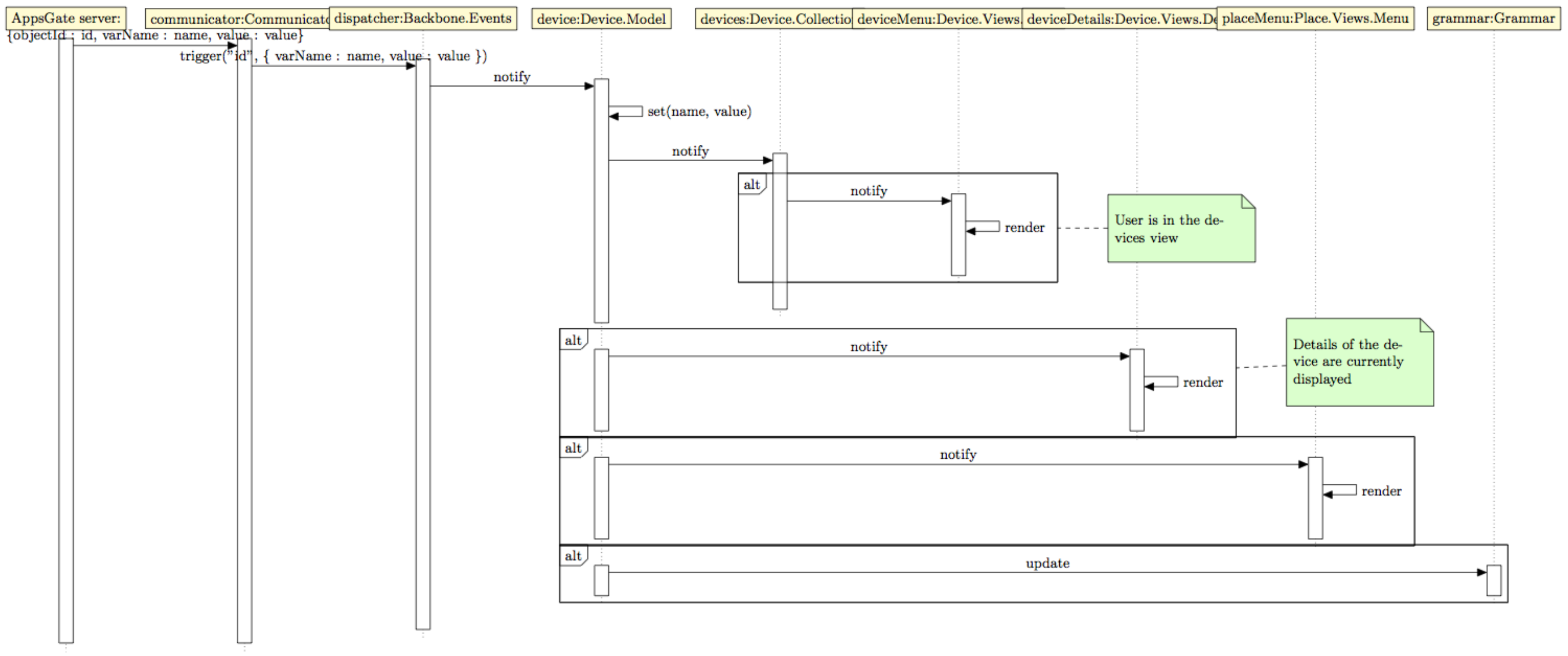

AppsGate server:
communicator:Communicato
dispatcher:Backbone.Events
device:Device.Model
devices:Device.Collectio
deviceMenu:Device.Views.
deviceDetails:Device.Views.De
placeMenu:Place.Views.Menu
grammar:Grammar
{objectId : id, varName : name, value : value}
trigger("id", { varName : name, value : value })
notify
set(name, value)
notify
alt
notify
render
User is in the devices view
alt
notify
render
Details of the device are currently displayed
alt
notify
render
alt
update

Figure 21. UML sequence diagram for the creation of a Place by the end-user.

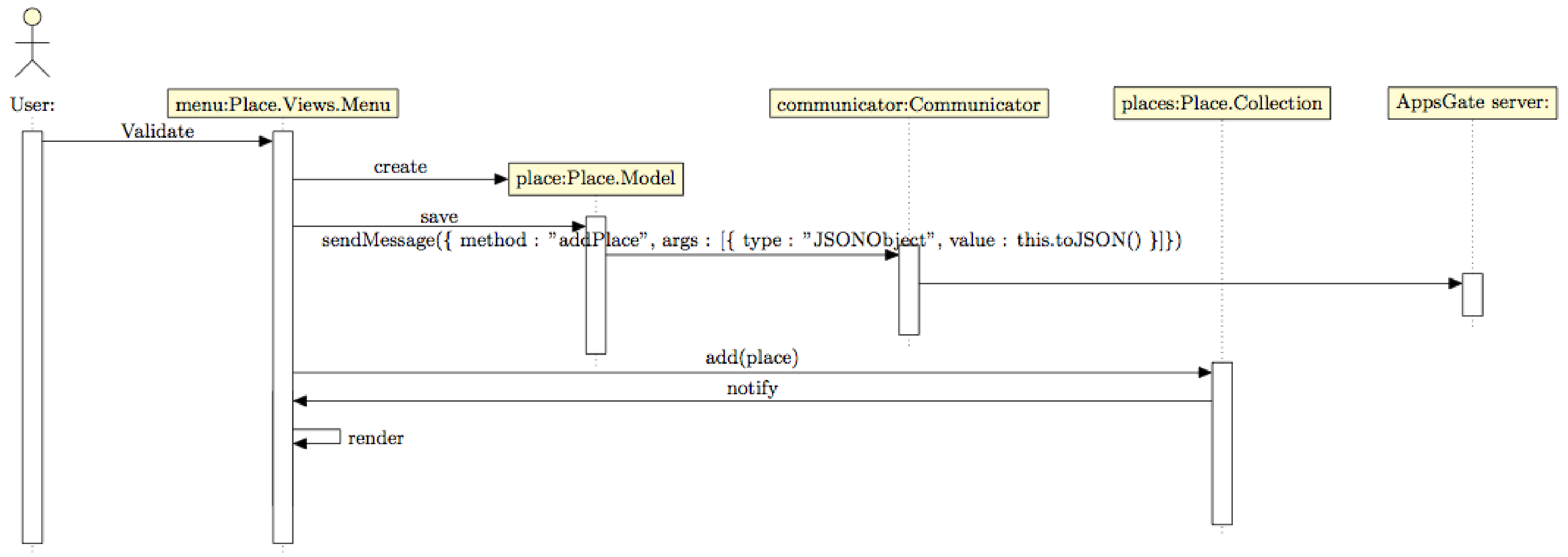


Figure 22. UML sequence diagram for the modification of a Place by the end-user

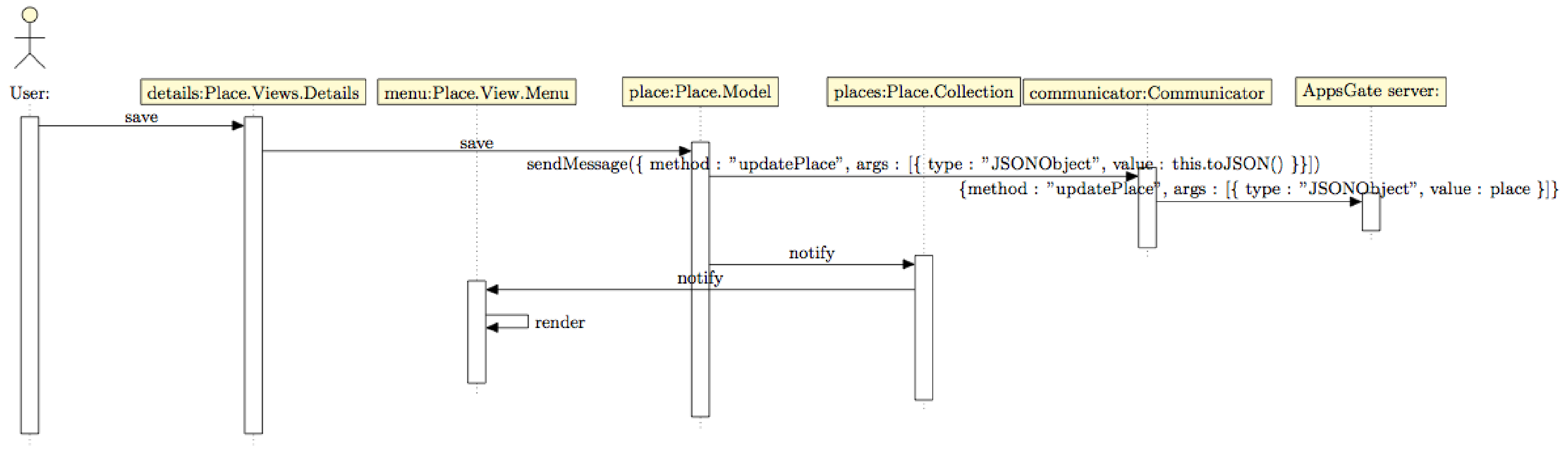

User:
details:Place.Views.Details
menu:Place.View.Menu
place:Place.Model
places:Place.Collection
communicator:Communicator
AppsGate server:
save
save
sendMessage({ method : "updatePlace", args : [{ type : "JSONObject", value : this.toJSON() }]})
{method : "updatePlace", args : [{ type : "JSONObject", value : place }]}
notify
notify
render

Figure 23. UML sequence diagram for the deletion of a Place by the end-user

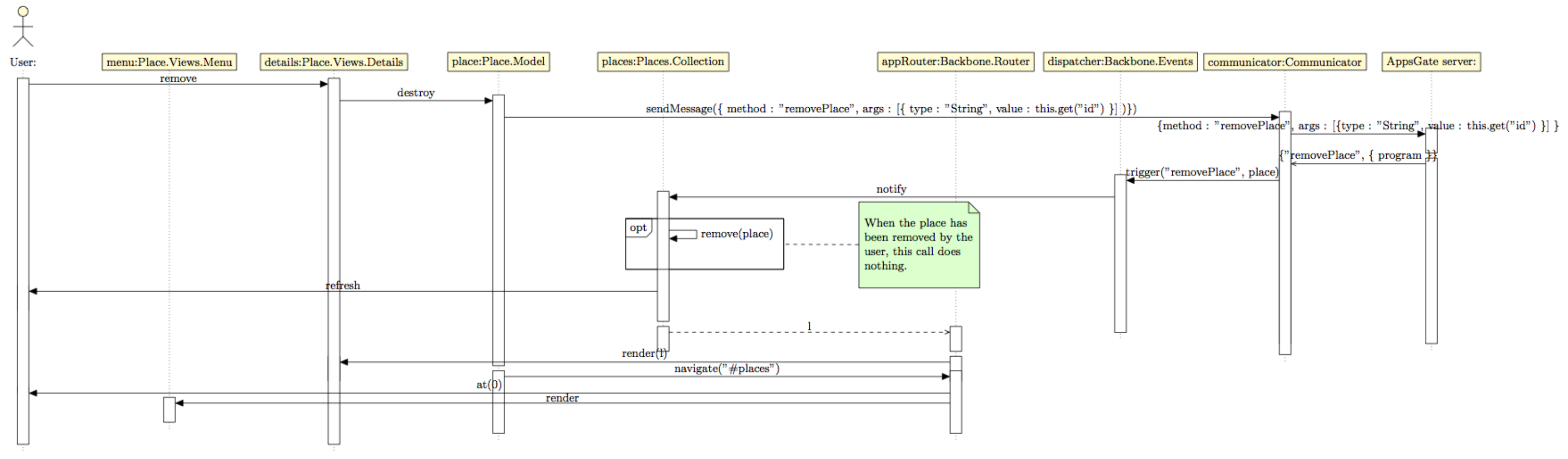

User:
menu:Place.Views.Menu
details:Place.Views.Details
place:Place.Model
places:Places.Collection
appRouter:Backbone.Router
dispatcher:Backbone.Events
communicator:Communicator
AppsGate server:
remove
destroy
sendMessage({ method : "removePlace", args : [{ type : "String", value : this.get("id") }] })})
{method : "removePlace", args : [{type : "String", value : this.get("id") }] }
{"removePlace", { program }}
trigger("removePlace", place)
notify
opt
remove(place)
When the place has been removed by the user, this call does nothing.
refresh
1
render(1)
navigate("#places")
at(0)
render

Figure 24. UML sequence diagram for the creation of a Program by the end-user

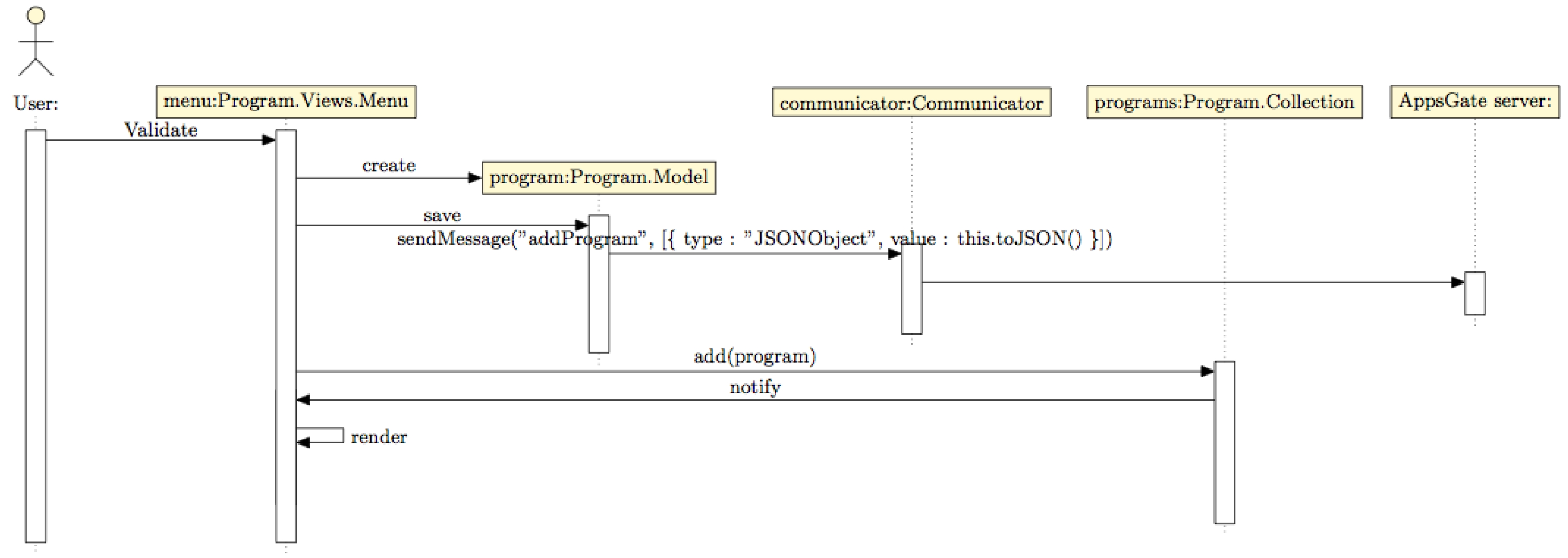

User:
menu:Program.Views.Menu
communicator:Communicator
programs:Program.Collection
AppsGate server:
Validate
create
program:Program.Model
save
sendMessage("addProgram", [{ type : "JSONObject", value : this.toJSON() }])
add(program)
notify
render

Figure 25. UML sequence diagram for the modification of a Program by the end-user

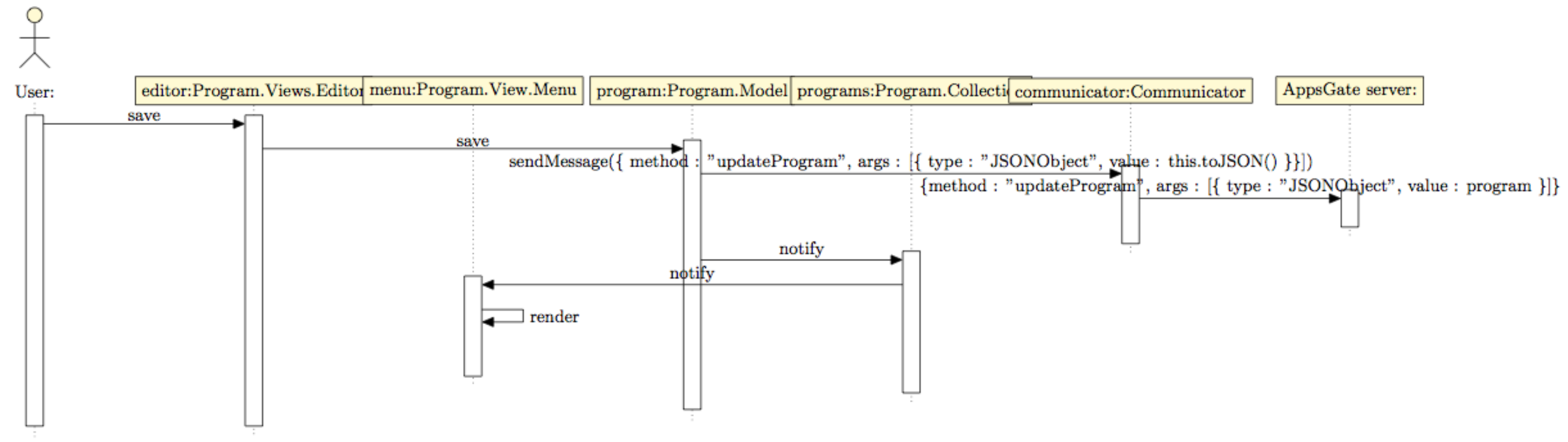
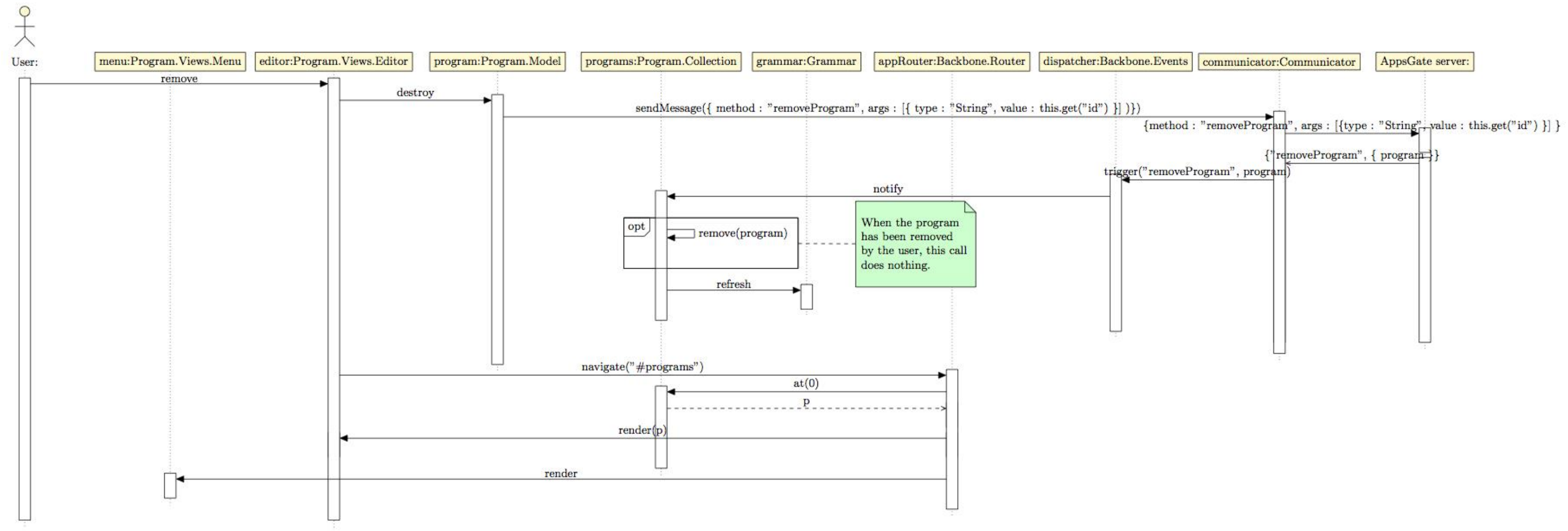

User:
editor:Program.Views.Editor
menu:Program.View.Menu
program:Program.Model
programs:Program.Collection
communicator:Communicator
AppsGate server:
save
save
sendMessage({ method : "updateProgram", args : [{ type : "JSONObject", value : this.toJSON() }]})
{method : "updateProgram", args : [{ type : "JSONObject", value : program }]}
notify
notify
render

Figure 26. UML sequence diagram for the deletion of a Program by the end-user.


User:
menu:Program.Views.Menu
editor:Program.Views.Editor
program:Program.Model
programs:Program.Collection
grammar:Grammar
appRouter:Backbone.Router
dispatcher:Backbone.Events
communicator:Communicator
AppsGate server:
remove
destroy
sendMessage({ method : "removeProgram", args : [{ type : "String", value : this.get("id") }] })
{method : "removeProgram", args : [{type : "String", value : this.get("id") }] }
{"removeProgram", { program }}
trigger("removeProgram", program)
notify
opt
remove(program)
When the program has been removed by the user, this call does nothing.
refresh
navigate("#programs")
at(0)
p
render(p)
render