AOLOA: A Composable Framework for Third-Party Applications for Smart Home Gateways

Eric Simon, Albert Royo Manjon, Sébastien Jean

AOLOA: a composable framework for third-party applications

for smart home gateways

Eric Simon, Albert Royo Manjón and Sébastien Jean
Univ. Grenoble Alpes, LCIS (CTSYS Research group),
F-26900 VALENCE, France {f_author, s_author}@lcis.grenoble-inp.fr

Abstract—In the last few years, the amount of smart devices in domestic environments has incredibly increased. Nowadays, a smart home is usually managed via a gateway offering value-added applications by connecting devices to the cloud. Every new device comes with its own features and protocols or cloud services. There is, consequently, a strong need for constantly modifying the gateway's behavior by deploying, removing or updating applications. However, there is no software architecture ensuring enough flexibility and trust to sustain this need. We consequently propose in this article a framework that allows to easily compose modular and context-aware software architectures intending to host third-party applications. This framework – called AOLOA (*Another OSGi-Like On Another*) – is based on OSGi and Java permissions. It ensures applications isolation, separates business-logic (higher level) and platform (lower level) layers and allows their trusted management.

Service-Oriented Computing, Ubiquitous Computing, Security, Software Isolation, OSGi, Java Security

I. INTRODUCTION

The next section introduces the context of our work and discusses its motivations.

A. Context

Our home is where most of our ubiquitous computing [1] experience takes place. The smart devices we bring (such as phones, watches or glasses) meet those already populating our home (like tablets, connected TVs or sensors) as well as the cloud, in order to offer us more and more connected services. Our home is smart enough to take decisions without us when we are out, but most often it lets us have remote control.

These services are usually orchestrated by a gateway, connected to the cloud and to our devices, hosting services that act as glue between them and where we can define context-aware behaviors.

Designing such gateways is not an issue, but performing this task becomes much more complicated when dynamism becomes a major concern. Existing software architectures already allow dynamically deploying or removing third-party applications (like Android, OpenHAB [2] or HomeOS [3]). These applications are generally made of software services, following the Service-Oriented Architecture (SOA) principles. In some cases, applications are provided from remote repositories in a "Foo-store" manner (like Android or iOS). However, these platforms are assumed to be used in a particular context (protocols, devices …) and consequently rely on a prebuilt and unmodifiable basis. Android, for example, considers only a predefined set of sensors/actuators and network interfaces. Taking into account a new sensor consequently implies updating the entire operating system.

As presented in Figure 1, home gateways traditionally distinguish several actors, with different roles:

- the *Gateway Owner* (GO) who is the resident and gateway's main end-user;
- the *Gateway Provider* (GP) who delivers the gateway to the GO;
- *Applications Providers* (AP) who design third-party applications and make them remotely available on applications stores (GPs can also be APs).

Services are often considered as building blocks of larger logical units called components. Components may provide several services and reciprocally require services provided by other components. Applications can then be considered as assemblies of components. Services running on gateways can be distinguished with regards to two major concerns. On one hand, technical services aim at providing hardware or communications abstractions (device drivers, network protocols …) as well as utility or administration features (logging, persistency, back-end …). On the other hand, business services embody the application's logic behavior.

B. Motivations and approach

Home gateways deserve better customizability. Our motivations are to enhance home gateways flexibility in order to make them as reusable as possible. In this article, we only focus on service-oriented gateways.

As previously said, services have different concerns. Technical services take place at a *platform* level, whose goal should be to customize the bare service runtime in order to take into account specific application domain and hardware. From a minimalistic point of view, GPs could deliver blank gateways, with a platform nevertheless tailored to their domain and hardware requirements. Business services, on the opposite, take place at an (upper) *application* level, and should allow customizing the platform for a specific use. This customization could ideally be done by the GO by deploying (and further updating/removing) third-party applications from remote stores. It is also conceivable that the platform level could also be partly customized by the GO (by GP delegation) or by an autonomic manager [4][5].

This customization makes more sense only if business and technical components' lifecycle can also be easily and dynamically managed. Nevertheless, because of the cost of such gateways as well as their energy consumption having to

remain low, the underlying software architecture must match classical embedded devices' footprints (typically, those of a Raspberry Pi or equivalent).

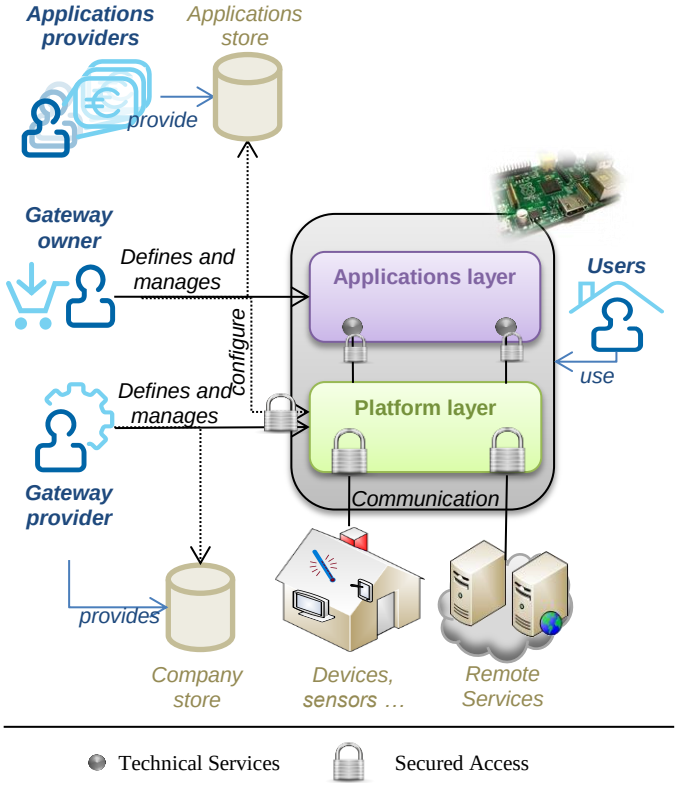

Figure 1. System actors

Finally, allowing third-party services to co-exist on top of a customizable basis, as well as allowing the end-user to deploy technical or business services, requires addressing some issues with regards to security and safety. As illustrated on Figure 1, our approach thus consists in clearly distinguishing the platform and applications layers, isolating services execution, and controling administration-related actions with roles and permissions. This approach is detailed in section III, after having presented some related work. Before concluding and drawing the lessons learned from our experiments, we introduce a reference implementation for the framework in section IV.

II. Related Work

Our work is an architectural solution that addresses various research domains, mainly isolation and security. Note that our framework – called AOLOA – aims at building a customized platform for a specific context. Consequently, some approaches cited in this related work section can be used in addition to our framework.

It is possible to classify "isolation" according to four layers that we further use to reference some relevant works.

- **Service isolation** (a usage scope of services), for example *OSGi service Hook* (cf. [6] pp.353-366) or *Region subsystem* (cf. [7] pp. 557-616)
- **Class isolation** (a use scope of classes) that can consist in import-export mechanism between class loaders (as in OSGi), in hiding packages for one or several groups (i.e. *Bundle Hook* (cf. [6] pp. 345-352), *Region subsystem* (cf. [7] pp. 557-616)), or in embedding systems in other ones (i.e.: *Region* or *V-OSGi* [8]).
- **Process isolation** (threads' execution separated in different system processes) requires in practice modifying the virtual machine in order to implement it. This is the case of I-JVM [9][10].
- **Virtual Machine isolation**, where applications and platform are hosted in different virtual machines is, here, a conceptual nonsense.

Our work can be seen as the continuation of [8]. In short, Virtual OSGi (V-OSGi) allows creating several application layers for a multiple-stakeholder context. In [11], authors blame that their work does not really address the problem of isolation for multi-stakeholder issues. This lack has motivated the I-JVM project (process isolation) [10]. Indeed, OSGi does not define any "process isolation" although it is required for multi-stakeholder. However, it is not an issue in our context where there is only one stakeholder, only one "owner", and some third-party applications which can be forced by security.

Concerning security issues, the framework proposed in this article is easily customizable, connected, and the dynamism and sharing mechanisms involve "some" complexity. In short, as discussed in [12], security for such a framework is essential, but also more complicated to implement. In this article, G. McGraw and G. Morrisett expose a methodology and classifications of attacks and solutions to establish the system security. Following [12] and [13], we have improved the security mechanisms of the AOLOA framework. Indeed, as explained above, we have similarities with the Android system: providing a platform, hosting third-party applications and provisioning through repositories/stores. Despite the fact that Android runs over a Unix system, it does not base this security on it, but on two main mechanisms: applications identification associated to a permissions list (IBAC) and (process) isolation maintained between applications and platform. All communications pass through the platform that checks through the ICC level. However, and also the platform dynamic management, in AOLOA – that are based on OSGi – applications can directly communicate with each other, but they must declare authorization to reach each other. This approach is also similartothe security-by-contract [14][15]. However, in the future, it will certainly be necessary to intercept communications to log and verify their content.

III. A Customizable Platform for Third-Party Applications

To sum up our motivations, the framework enhancing gateway customizability **must**:

- allow dynamically deploying and managing technical components on the platform layer;
- allow dynamically deploying and managing business components – which can be provided by third-party companies – on the applications layer from the platform;
- provide a mechanism to share targeted technical services with the business components of the applications layer.

The framework also has to take into account the following **constraints**:

- it **must** secure the execution environment from malicious actions coming from the applications layer;
- it **must** prevent and protect the platform from errors and faults originating from the applications layer;
- it **should** be resource-friendly.

This section briefly discusses the minimal targeted hardware before introducing our approach with regards to safety and security, as well as their impact on software architecture. Finally, it details the deployment unit's life-cycle.

A. *Targeted hardware*

Dynamism is a main concern in our approach, as deployment and management of both applications **and** platform could be done "on the fly". However, deployment and dynamic binding usually rely on either interpreted languages or virtual machines (HomeOS[3], OSGi[6], Kevoree[16]). Our approach thus considers using an embedded platform able at least of hosting a virtual machine. In our experimentations, we have been using a Raspberry Pi model B[1].

B. *Security and safety mechanisms*

Obviously, the *Gateway Provider* cannot trust third party *application providers*. An application can indeed be badly implemented or malicious. The framework must consequently have **safety** and **security** mechanisms: safety to prevent and protect the platform layer from faults originating from applications, security in order to block malicious actions. To ensure these properties, our framework relies on isolation and access control.

1) *Isolation*

As mentioned in the related work section, isolation mechanisms can be basically classified according to several layers: **object/class isolation** where it must not be possible respectively for an object/class to access to another object/class outside of its usage scope; **process isolation** where threads can be separated and isolated in different system processes; and finally **Virtual Machine isolation** where applications are executed in different virtual machines.

It is, here, a conceptual nonsense to isolate the platform layer and the applications layer in two **virtual machines** (not to mention the strong impact on performance). **Process isolation** offers a great isolation between layers running in the same VM. However, it supposes that the VM supports it and that is not the case for the classical Java or .Net VMs. Our technological choices – discussed in the section IV – imply using the Java VM. Consequently, implementing process isolation requires to modify the VM for each gateway. It is not consistent with our goal of alleviating the burden of the gateway provider when designing platforms.

We consequently focus on the classes and objects isolation. In opposition to [10][11], the gateway provider is, in our execution context, responsible for the platform layer while the gateway owner is responsible for the applications layer. As there is no multi-stakeholder, process isolation is consequently not mandatory. This isolation mechanism can be implemented by simulating several execution spaces where shared resources must seem stateless and private resources must be hidden (cf. [7] pp. 557-616). It can also be implemented by truly separating execution spaces. The latter approach simplifies security management and allows defining platform and application with few side effects. Although classes and objects are isolated between both layers, the framework provides a mechanism to export packages and services.

2) *Security*

We identified three angles of attack in the proposed framework (cf. locks of the Figure 1):

- Malicious access made from the web administration (apps and platform) interface;
- Malwares contained by corrupted components coming from unofficial (considered unsecured) repositories;
- Data collection and malicious behavior from an "official" signed component.

The subsection I.B has underlined three "roles" (apps user, gateway owner, gateway provider) and three "uses" (use apps, manage apps and manage platform). The table below sums up "uses" authorized for each of the roles. To prevent attacks through the web administration interface, **the framework should have a RBAC (*Role-Based Access Control*) [17] mechanism that applies on all administration operations**.

TABLE I. ROLE/USE MATRIX

		Authentication required	
	Locally use apps	**Manage Apps**	**Manage Platform**
User	Yes	No	
Owner	Yes		Limited by the provider
Provider	Limited		Yes

It is necessary to scan (cf. "Scanning for Known Malicious Code" section of [12]) potentially corrupted components before their installation. Generally, this scan is made when components are pushed to the official repository. However, if the gateway owner deploys from an unofficial repository, the framework integrity can be broken. Consequently, **the installation mechanism should scan components after their transfer and before their installation**.

Finally, the main security issue is to execute (without trust) third-party applications. Indeed, an application downloaded from an "official" repository does generally what it was intended to do. But sometimes, it has malicious behavior (in general, stealing users' information). Identifying malicious applications is more complicated because they do not necessarily have virus signature (detected by scan) and they do what they say they do. To detect them, it is necessary to have two mechanisms: resources access control and communications log.

The applications layer must perform access control on every resource in the execution environment. This access control is based on permissions declared by each business component (indeed, the gateway provider cannot define it a priori and exhaustively). Consequently, each business component must declare every access to framework resources: classes, business and technical services, files, system properties… It must also declare which of its resources could further be used by other components: business services,

[1] http://www.raspberrypi.org/

packages… Unspecified actions are automatically denied (file access, life-cycle change, socket access…). This list is provided to the deployer (GO or GP) that can dynamically grant or reject all or part of them. Thus, permission management of IBAC (*Identity-Based Access Control*) type is delegated to the deployer. Conversely, a business component can provide business services that want to restrict the use to a subset of business component always running on platform. For that, it should provide rules or signatures that the components must provide to use it: it is the purpose of security-by-contract [14][15].

Communication protocols (from/to outside) must be provided and controlled (logged) by the platform layer.

If these mechanisms are implemented, and thanks to the isolation between both layers, the platform layer can then be considered as a DMZ (*DeMilitarized Zone*). Consequently, installation and execution of technical components, usually managed by the gateway provider, no longer require any particular access control because they share the same level of trust. Then, the composition and management of the platform layer are simplified. However, as the platform layer can export some technical services and consequently share the associated classes, our framework must also isolate exported services, classes and classloaders. In our case, it is done through the use of proxies and temporary classloaders.

C. Deployment-unit concepts

In this article, we abusively use the word *component* to refer to the deployment unit (the atomic element to transfer). Indeed, a "true" component-model will further be defined and considered as an overlay of our framework because it is domain-specific. The only paradigm imposed by our framework is that platform and applications should be built on top of the SOA (Service-Oriented Architecture) of each layer. Thanks to SOA's loose coupling, the dynamic reconfiguration of layers is eased.

Life-cycle operations that apply to these components are currently the classic ones: transfer into the framework, loading, activation, deactivation, unloading, updating and removal. However, the isolation and security mechanisms impact the components life-cycle. Thus, a business component must declare the permissions to access resources.

The following example is further used to explain these mechanisms.

The purpose of this application example is to monitor ambient inside and outside temperatures and to notify an alert via emails when these temperatures exceed a predefined threshold. As shown in Figure 2, this application is composed of five deployment-units:

- *SMTP-Client Service* (MS) provides a service which allows receiving and sending emails using an IMAP mail server account.
- *Sensors Discovery Service* (DS) provides a service which allows registering a handler for each sensor type to receive their notifications. This service embeds a plug and play protocol such as UPnP or DPWS that interfaces a business-oriented overlay for these sensors.
- *HTTP Service* provides a service through which it is possible to register HTTP servlets to an embedded HTTP server.
- *Aggregator Service* (AS) aims at aggregating data coming from sensors and sending alert email when a predefined threshold is passed. For that, it provides a service which allows configuring the email account of the sender as well as that of the receiver. This service also allows configuring the inside and outside temperature threshold and getting the current temperature. Consequently, it uses MS and DS.
- *SenseUI Service* is the Web interface (Servlet) enabling AS access. So, it uses the AS service.

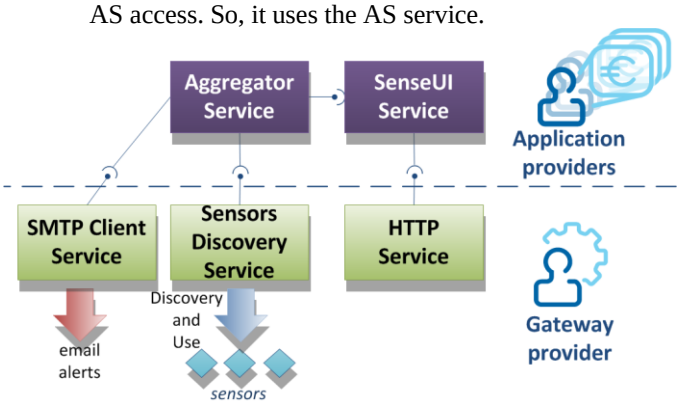

Figure 2. Application example using AOLOA

In this example the *Aggregator Service* (AS) requires the *Discovery Service* (DS) and the *SMTP Client Service* (MS), and provides a service used by the *SenseUI* graphical interface.

The AS must consequently declare three *requirements* to both get and register these services. It must also provide a *capability* that defines the permission required to use its service. Finally, the AS must declare, with the same logic, provided and required packages.

We have generalized the approach through the following model (cf. Figure 3):

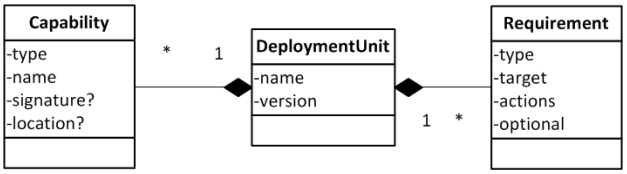

Figure 3. Deployment-Unit security model

- A **deployment-unit** provides capabilities (provided resources) and has requirements (required resources) to/from the execution environment.
- A **capability** is a resource provided by the deployment-unit. It is defined by its *type* (for example: service, event, data), the *name* of the targeted resource and the *location* or the *signature* of the deployment-unit authorized to use it. If neither the signature nor the location is specified, everyone is authorized to use it.
- A **requirement** is a required resource. It is defined by the *type* of the targeted resource, the *target* of the permission, the *actions* on the resource, and the fact that the requirement is *optional* or not (deployment-unit can be started without this resource).

These permissions are checked by the framework and prompted to the deployer (gateway owner), who might have to

validate them (or not). This step implies adding a specific state in the deployment-unit life-cycle.

D. Deployment Unit Life-cycle

The deployment-unit life-cycle that applies to the applications layer has the seven following states (cf. Figure 4):

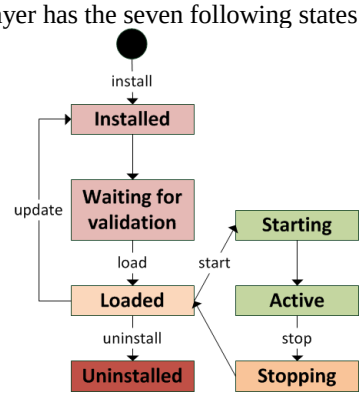

Figure 4. Business deployment-unit life-cycle

- **Installed**: The deployment unit is installed: the transfer has been done and it is ready to load.
- **Waiting for validation**: This state means that the deployment unit is frozen until its deployer accepts or rejects the set of required permissions.
- **Loaded:** This state means that the permissions have been checked and the classes and static resources (pictures…) have been loaded in the layer. At this state, the deployment-unit is ready to start.
- **Starting:** It is a transition state between the loaded and the active states, where the notifications are sent and the context of the deployment unit is initialized.
- **Active:** This state means that the deployment-unit, which was in the "starting" state, has been started and is now running.
- **Stopping:** During this state, notifications of the termination are sent. When the stop and underlying operations are processed, the deployment-unit returns to the loaded state.
- **Uninstalled:** The deployment-unit is uninstalled.

The deployment unit life-cycle in the platform layer has (cf. Figure 5) the **Installed, Loaded, Starting, Active, Stopping and Uninstalled** states previously defined for the applications layer.

However, it also has the **Exported** state. Indeed, once the deployment-unit is started and active, the framework checks whether resources must be exported to the applications layer. If the applications layer is started and if related exports have been made, the deployment unit then moves to the "exported" state.

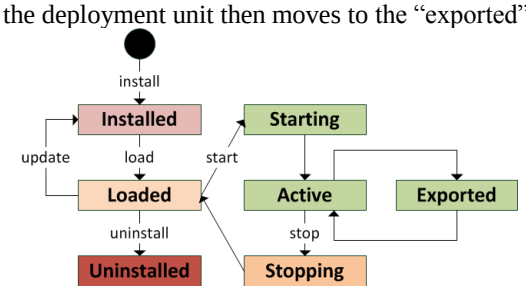

Figure 5. Technical deployment-unit life-cycle

If the applications layer is stopped, then the deployment unit comes back to the "active" state.

As suggested by the "exported" state, the applications layer itself has a lifecycle. Indeed, any modification in the platform layer may require the restarting of the applications layer. It is also possible to shut down the applications layer if there is no third-party application running.

Consequently, the applications layer's life-cycle (cf. Figure 6) has three states:

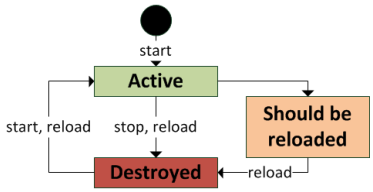

Figure 6. Applications layer's life-cycle

- **Active**: the applications layer is started and operational;
- **Destroyed**: the applications layer has been stopped, and all references to it have been released;
- **Should be reloaded**: a modification in the framework may require restarting the applications layer. It is up to the platform layer to decide when it is appropriate to operate this restart.

E. Synthesis

Our main objective is to give to the *gateway provider* the ability to design, deploy and dynamically manage its platform as a classic modular application-based service, where the life-cycle of applications is ruled by security mechanisms and the life-cycle of the platform itself.

There are three main concerns in the proposed framework:

- Security: inherent in the third-party applications hosting;
- Dynamic management: inherent in a constantly varying open environment;
- Resource-friendly: related to the economical aspect of technology adoption.

In this approach, **technical components** (cf. Figure 7) aim either at providing services and features for **business components** (like device communication protocols or remote services), or at refining and controlling both the platform's and applications' life-cycles.

For example, a **technical component** exposes a service to drive the house's HVAC, while a **business component** (cf. Figure 7) uses it to regulate inside temperature. Classes required to use services are shared with the applications layer, and service objects are exported/registered in the applications layer.

A modification (install, removal…) in the applications layer does not impact the platform layer. The reciprocity is false, the applications' life-cycle being linked to the platform's life-cycle. Consequently, a change occurring in the platform may have an impact on the application life-cycle. This is the case, for example, when a business component using a technical service or a technical API must be updated.

Both the security mechanisms (IBAC for the third-party applications, RBAC for the administration and "scan" before install) and the isolation mechanism finally allow considering

the platform layer as a DMZ. It consequently facilitates the composition and the management of the platform by the gateway provider.

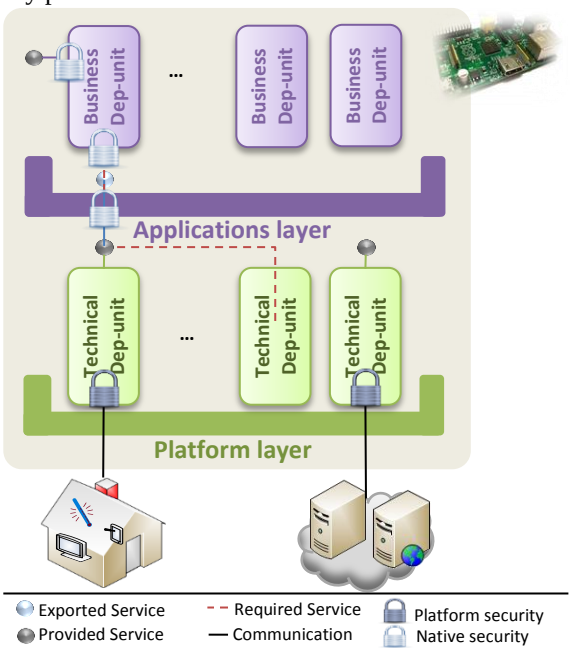

Figure 7. Framework architecture

IV. AOLOA Implementation and Experimentations

In this section, we firstly describe our technical solution. Then, we introduce some execution metrics gathered from experiments on a PC and a Raspberry Pi Model B.

A. Technical solution

The AOLOA framework – for *Another OSGi-Like On Another* – is developed in Java-SE6 and uses the Felix OSGi implementation to develop both platform and applications layers. Security mechanisms are implemented by using the *Felix security bundle* (*PermissionAdmin*). AOLOA framework combines the OSGi and Java permissions and extends the OSGi bundle life-cycles for the both layers.

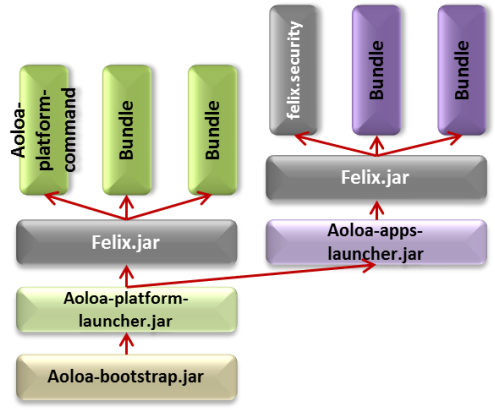

Figure 8. AOLOA framework launch hierarchy

AOLOA is currently (version 1.2.4) available[2], but some features are only partially implemented or unavailable. The current version has 8004 lines of code (obtained with sonar tool[3]) for 91 classes and 32 packages and dispatched in 6 modules (Classic Jar and Bundles). The unit test coverage of the core (excluding shell and GUI) is 46% (51,2% for the lines and 34,5% for the branches).

These 6 modules are:

- **aoloa-security-maven-plugin** is a maven plugin to generate security permissions used for the application layer. This plugin uses the maven-bundle-plugin properties and declared metadata (for example an access permission to a file) to generate the list required by the business component to be executed. The following snippet has been generated – with this plugin – in the *Aggregator Service* manifest from the maven-bundle-plugin information.

```
required-permissions:
[…].AdminPermission\$this\listener,metadata;
[…].ServicePermission\fr.lcis.ctsys.aoloa.services.discov
ery.demo.service.api.ServicesPublicationService\get;
[…].ServicePermission\fr.lcis.ctsys.aoloa.demo.alert.mail
.sender.api.AlertMailSenderService\get;
[…].ServicePermission\fr.lcis.ctsys.aoloa.demo.aggregator
.DataAggregatorService\register;
[…].PackagePermission\org.osgi.framework\import;
...
[…].PackagePermission\fr.lcis.ctsys.aoloa.demo.aggregator
\exportonly,import;
```

- **aoloa-bootstrap** initializes the boot classloader to share a static and unique API and libraries for the different framework layers (cf. Figure 9). This module is the environment launcher and allows starting and stopping the platform layer (aoloa-platform-launcher) (cf. Figure 8).

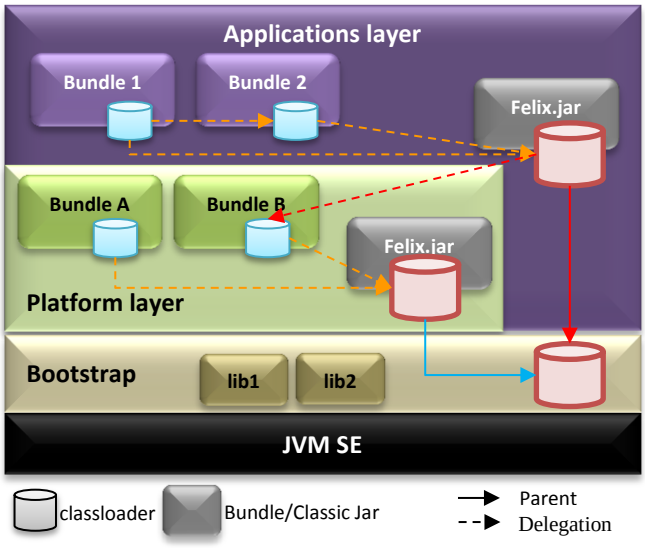

Figure 9. AOLOA Classloader architecture

- **aoloa-platform-launcher** initializes a classloader used to launch an OSGi implementation (platform layer) –

[2] https://sourcesup.renater.fr/projects/aoloa/
[3] http://www.sonarqube.org/

here org.apache.felix – and provides two services: one to manage (cf. Figure 6) the applications layer (aoloa-apps-launcher) and export packages and services, another to manage (cf. Figure 4) the bundles from the applications layer.

- **Aoloa-apps-launcher** initializes a classloader allowing delegating some packages to the bundle from the platform layer (cf. Figure 9). This classloader is used to launch another Felix for the applications layer where the felix.security component runs (cf. Figure 8).
- **aoloa-platform-command** and **aoloa-visu** are respectively a Java Swing GUI and a Web-GUI used to manage both the application and the platform layers.

The AOLOA framework combines a classic classloader hierarchy with the classloader delegation network from OSGi (cf. Figure 9). It allows launching two distinct OSGi frameworks in a same classloader and shares packages and services from the platform layer to the applications layer.

B. Metrics

We have extracted some metrics from the current implementation of AOLOA. These metrics are:

- the starting time (ST) for the boot, the platform layer and the applications layer;
- the framework memory footprint (MF) (used memory is raised after a garbage collector execution at the starting and when the initialization is done).

The procedure to gather these metrics is classic: the framework is launched 100 times, the 10 min and 10 max values are removed, and the result is the average value.

This procedure has been performed both on a laptop (Dell Latitude E6420, Windows 7, Oracle JDK1.6.0_35) and on a Raspberry Pi (Model B, Raspbian, Open JDK 1.6.0_27).

TABLE II. EXECUTION METRICS

	Dell Latitude E6420	Raspberry Pi Model B
Boot ST	36ms	581ms
Platform ST	543ms	7524ms
Applications ST	321ms	9883ms
Framework MF	2,78Mo	NC

The framework memory footprint is not communicated because it seems wrong (>0,4Mo). We currently cannot explain why the platform layer starts faster than the applications layer on the laptop and inversely for the Raspberry Pi (CPU instruction? JDK?...); this issue will be analyzed further.

C. Experimentation

To experiment the AOLOA features, we have developed the "ambient temperatures monitoring" application described previously (cf. subsection III.C). In the initial case, the *HTTP service* and the *AOLOA Visualization service* (web administration interface) were pre-installed on the platform-layer. By the way of the *Visualization*, the *Services Discovery Service* (based on an ad'hoc discovery protocols) and the *SMTP mail service* were installed and started into the platform layer. Next, the *Aggregator Service* and the *SenseUI service* were installed. However, and in opposition to the technical services, we were prompted during their installation to validate the resources' access permissions. The gateway administration service has alerted about the fact that the AS requires to bind to both the DS and MS services. Once the permissions have been granted, the AS service has been started properly. The same permissions granting step has occurred while installing the *SenseUI service* and before it turns active. Once the application has been activated and operational, the previous services have been stopped and uninstalled.

This scenario has been considered as an *acceptance testing*. These experiments allowed us to underline a set of theoretical and technical problems, most of which are discussed in the following section.

V. LESSON LEARNED AND ROADMAP

Since AOLOA is still under development, some features are not yet totally implemented. This is particularly the case for the capabilities, where only requirements are yet developed. However, through the different experimentations, we have learned the following lessons.

A. Theoretical issues

The motivation of security in a dynamic and open context is easy to understand. In opposition, its effective implementation is extremely complicated. In a first step, the "attack endpoints" of the system must be identified and typed. Then, for each of them, well-known mechanisms can be applied. A formal verification could be performed to check the closure of the system. However, the platform running on AOLOA is not predefined and the deployer must validate (or not) the security permission applying to the third-party applications. A major problem occurs with the last point: the end user is prompted. Indeed, he does not necessarily have the security skills and so he can compromise the integrity of the security system (e.g.: in authorizing writing to file system or Java reflection). One of the questions we have been asking ourselves for some time: should we protect the user from himself? And in such case, how?

B. Technical issues

AOLOA was designed and developed from the OSGi specifications in hoping to be able to substitute the OSGi framework implementations (e.g.: Felix, Equinox or mBS of PROSYST). Although OSGi standardizes a way to launch the OSGi framework allowing embedding it in another framework; this does not mean that the OSGi implementations and their "official" bundles take it into account. We have identified technical problems while using the Felix OSGi implementation; especially the two mentioned below.

AOLOA uses the framework bundle context from the applications layer to delegate on the other bundle from the platform layer, in substituting its classloader parent with ours. Of course, we check that the desired class belongs to an exported package, and that the application has the required permissions. However – for some reason we have not yet identified – the context used for permissions is not the application's, and therefore it does not have permission (in the experiments).

The second problem is related to the persistence of states when the platform restarts: some components can be restarted before the end of the AOLOA security mechanism initialization and consequently have an unsafe behavior.

In these two cases, we have found a "neat" solution, that requires changing the source code of the OSGi implementation (here Felix) and a "dirty" one (costly in terms of performance and bad in terms of software engineering in comparison with the "neat" one), but that does not require changing the source code.

C. Roadmap

In parallel with the previous technical and theoretical problem resolution, we will begin the following roadmaps: one that could be named "engineering" (that we must deal with in the case of industrial partnerships) and the other one being more "research-related".

In the "engineering" roadmap, we must develop the Web administration interface with a Role-Based Access Control, and a repository with the associated client pro-active in the malicious code detection.

In the case of the "research" roadmap, we should study the "real" component models like SCA [18], Blueprint [19] or iPOJO [20] and the associated security.

Finally, once we have all these bricks, we will finally address the autonomic management for the platform and the applications, and the active analysis of the security logs.

VI. Conclusion and perspective

The approach proposed in this article aims at providing a composable framework dedicated to host third-party context-aware applications on gateways. Existing software architectures already allow dynamically deploying or removing third-party applications. However, they generally assume being used in a particular domain and so rely on a prebuilt basis. The number of possible domains is wide. Consequently, our goal is to provide a generic basis that allows to quickly and easily compose a domain-specific platform.

The proposed framework is composed of two layers: one to define and manage the domain-specific platform and the second to deploy and run third-party applications. The platform layer allows managing the applications layer and sharing resources (classes, services…) with it. Consequently, Security and isolation are two major concerns.

Users do not trust third-party applications that are dynamically installed. When a deployment unit is to be installed, the framework has to check the declared security permissions and delegate the acceptance or the rejection of all or part of these permissions to the deployer.

On the other side, the framework ensures isolation, forbidding access to resources that are not to be shared between both layers and to limiting the side effects. Layers are isolated thanks to a combination of a classical Java classloader hierarchy and OSGi classloader delegations networks.

This approach has led to the development of the AOLOA framework (*An OSGi-Like On Another*) based on Java6 and OSGi (Felix implementation). To validate it, an "ambient temperatures monitoring" example has been developed and rolled out over a PC and a Raspberry Pi.

Although our approach is still incomplete, initial experiments are quite encouraging, pushing us to continue established roadmaps.

References

[1] T. Kindberg & a. Fox, "System software for ubiquitous computing", Pervasive Computing, IEEE, 2002, 1, 70-81

[2] OpenHAB. Open HAB site. http://www.openhab.org/index.html

[3] C. Dixon, R. Mahajan, S. Agarwal, A.J. Brush, B. Lee, S. Saroiu, & V. Bahl, "The Home Needs an Operating System (and an App Store)", in HotNets IX, ACM, 20 October 2010

[4] P. Horn, "Autonomic computing: IBM's Perspective on the State of Information Technology", IBM, IBM, 2001

[5] J.O. Kephart & D.M. Chess, "The vision of autonomic computing", Computer , vol.36, no.1, pp.41,50, Jan 2003

[6] OSGi Alliance. OSGi Core Release 5 specification. http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf

[7] OSGi Alliance. OSGi enterprise Release 5 specification. http://www.osgi.org/download/r5/osgi.enterprise-5.0.0.pdf

[8] Y. Royon, S. Frénot & F. Le Mouël, "Virtualization of service gateways in multi-provider environments", Proceedings of the 9th international conference on Component-Based Software Engineering (CBSE'06), Springer-Verlag, 2006, 385-392

[9] N. Geoffray, G. Thomas, B. Folliot & C. Clément, "Towards a new isolation abstraction for OSGi", Proceedings of the 1st workshop on Isolation and integration in embedded systems, ACM, 2008, 41-45

[10] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot & B. Folliot, "I-JVM: a Java Virtual Machine for Component Isolation in OSGi", Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN 2009), IEEE Computer Society, 2009, 544-553

[11] S. Frénot, F. Le Mouël, J. Ponge & G. Salagnac, "Various Extensions for the Ambient OSGi Framework", International Journal Adapt. Resilient Auton. Syst., IGI Global, 2011, 2, 1-12

[12] G. McGraw & G. Morrisett, "Attacking Malicious Code: A Report to the Infosec Research Council", *Software, IEEE* , vol.17, no.5, pp.33,41, Sept.-Oct. 2000

[13] W. Enck, M. Ongtang, & P. McDaniel, "Understanding Android Security", Security & Privacy, IEEE , vol.7, no.1, pp.50,57, 2009

[14] A. Philippov, O. Gadyatskaya & F. Massacci, "Security of the OSGi Platform", Proceedings of the Doctoral Symposium of the International Symposium on Engineering Secure Software and Systems (ESSoS 2012), 2012, 11-16

[15] N. Dragoni, F. Massacci, C. Schaefer, T. Walter & E. Vetillard, "A Security-by-Contract Architecture for Pervasive Services", Proceedings in Security, Privacy and Trust in Pervasive and Ubiquitous Computing, (SECPerU 2007). 2007, 49-54

[16] Kevoree project. http://kevoree.org/

[17] R S. Sandhu, E.J. Coyne, H.L. Feinstein & C.E. Youman, "Role-based access control models", Computer , vol.29, no.2, 1996

[18] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni & J.-B Stefani, "A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures", In Software: Practice and Experience, Wiley, 2012, 42, 559-583

[19] IBM. OSGi Blueprint Container Specification http://pic.dhe.ibm.com/infocenter/radhelp/v8/index.jsp?topic=%2Fcom.ibm.osgi.common.doc%2Ftopics%2Fcosgiblueprint.html

[20] C. Escoffier, R.S. Hall & P. Lalanda, "iPOJO: an Extensible Service-Oriented Component Framework", Proceedings in International Conference on Services Computing SCC 2007, 2007, 474-481