Developing Dynamic Heterogeneous Environments in Smart Building
Using iPOPO

Shadi ABRAS 1,2, Thomas CALMANT 2,4, Stephane PLOIX 3, Didier DONZE 2, Frederic WURTZ 1, Olivier GATTAZ 4 and Benoit DELINCHANT 1
1 Univ. Grenoble Alpes, Laboratory G2Elab, Grenoble, France
2 Univ. Grenoble Alpes, Laboratory LIG, Grenoble, France
3 Univ. Grenoble Alpes, Laboratory G-SCOP, Grenoble, France
4 IsandlaTech, Grenoble, France

Keywords: Ambient intelligence, iPOPO, Energy, Smart-Home based architecture component.

Abstract: This paper contributes to the design of a smart monitoring system. The objective of this paper is to show that by using iPOPO framework, as an embedded application for sensors/actuators, it is possible to improve the configuration/reconfiguration of monitoring system. The proposed experimentation platform, called PREDIS/MHI, based on iPOPO framework, is characterised by its openness, its scalability and its capability to manage diversity. In this paper, we show how application based on iPOPO framework, well adapted to represent problems spatially distributed and opened, can dynamically be adapted to various contexts of environments.

1 INTRODUCTION

A monitoring system basically consists of sensors and actuators linked via a communication network allowing interactions for control purposes. Thanks to this network, a load control mechanism can be implemented: this is called distributed control. A monitoring system is made of sensors, which measure physical quantity, and actuators, which control some devices. However, faced to the increasing number of sensors and actuators in buildings, it is necessary to develop monitoring systems able to plan ahead changing configuration of buildings. The development of a dynamic environment composed of heterogeneous devices and technologies is very complex. Smart dynamic environments can be found in many domains such as: domestic, social, biology, medical, etc.

(7) proposed a GatorTech Smart House system. Sensors and actuators are fitted on a number of devices: mailbox, entrance door, floor etc which are connected to an operational platform designed to optimise the comfort of the inhabitant. It also uses a high precision ultrasonic tracking system to locate occupants and evaluate their mobility habits to better control the environment (7).

(7) proposed an adaptive control of home environment. It monitors the environment and observes the actions taken by the inhabitants (using lights, adjusting thermostat). This data is further used to infer patterns in the house, using reinforcement learning, a stochastic form of dynamic programming.

In the House of the future project (7) aimed to design strategies for more flexible environments that meets inhabitants physical and cognitive needs than current environments. The system consists of three components, set of state-change sensors used to collect data about the use of objects, a context aware experience sampling tool (ESM) used by the inhabitant to label his activities, and pattern recognition and classification algorithms for recognising activities. Nave Bayesian classifier is used to train the model and predict future activities of inhabitants

However, the previous works based their control system on a centralised approach; they does not opt for solutions fostering modularity.

Problem statement

Owing to the number and diversity of sensors/actuators in the home, leads us to opt for solutions fostering modularity. The need for structural self-adaptation, or more technically the need for plug-and-play appliance, leads to a Service-Oriented Architecture (SOA) platform. Indeed, a home moni-
toring system must be open-ended and extendible: it must be possible to add or remove appliances (or new types of sensors/actuators) at any time without calling into question the overall operation of the system.

This paper focuses on the development of smart domestic environments for connecting evolving sensors and actuators. The framework iPOPO has been used in order to develop a smart dynamic environment for PREDIS/MHI. It is an experimentation platform consisting of many kinds of sensors/actuators of different technologies such as X10, Oregon Scientific, Zigbee, HTTP GET and USB (Fig. 1).

Figure 1: Components of PREDIS platform

In what follows, in a first section, the paper describes briefly the PREDIS/MHI platform. In a second section, an iPOPO framework is presented. Section 3 describes the architecture of the iPOPO-based implementation of PREDIS platform. Finally, a conclusion of the contribution of the iPOPO framework based on SOA techniques to the monitor system is presented.

2 PREDIS/MHI

PREDIS is a smart building experimentation, which aims to ensure better energy management and human comfort in buildings: the building’s maximal energy consumption did not exceed 50kWhPE (PrimaryEnergy)/year.m² (7).

It combines physical models and experimental measurements in order to have more adapted models for virtual simulation and optimal control (7).

This platform are developed around:

• Multi-sensor monitoring.
• User activities and their energy impact analysis.
• Multi-physical modelling, measurement handle and sensitivity analysing.

The first floor of this building is composed of two rooms: teaching room and researcher open space. The teaching room is used for training courses of Grenoble University and equipped with laptops, which are connected to the electrical grid and to photovoltaic generators. The researcher open space, where the control, supervision and measurement are installed, is daily occupied by about six researchers.

The first floor of PREDIS is equiped of many systems:

• Heating, Ventilation and Air Conditioning system.
• Lighting control.
• Experimental instrumentation: temperature sensors, airflow sensors, CO₂ sensors, energy meters, and switching devices.

In this platform, a system called HAL (Home Abstraction System) has been added as a general interface to the control systems and sensors/actuators. HAL integrates drivers of different communication protocols relating to physical devices. It shows data through a RESTful Web interface, accessible via a web browser or a client application. Its objective is to provide a uniform access to devices and information. The HAL system has been developed in Python because most of sensors drivers have been provided in Python. Figure 2 illustrates the architecture of components of the presented system (HAL).

Figure 2: Architecture of HAL system components

In figure 2, many types of sensors are used: temperatures, humidity, CO₂, airflows, electric energy, electric power, light, presence, door position. These sensors rely on many technologies: X10, Oregon Scientific, Zigbee, HTTP GET and USB. There
is a legacy energy management system: an Intouch SCADA system, which can be connected via the OPC protocol.

Once the HAL system has been deployed in the PREDIS system, all functions and facilities of this system can be used. However, if a driver linked to sensors/actuators fails, the entire integrated system does not well work: it leads to the failure part or the whole of the system. Indeed, when an additional sensor or actuator is put in operation, it call into question the overall operation of the system: restarting the system must be done in order to take into account the new appliance. If a driver linked to sensors/actuators is put out of operation, either accidentally or through administration action, the system and its peripheral components are severely affected.

In the HAL system, the life cycle of each sensor, actuator or driver is managed by the developer. For example, for each time, when a new sensor is added to the system, the developer has to do new configuration for the whole system, which takes a time for restarting the system. The HAL system depends on the life cycle of sensors and the configurations defined by the developer. The HAL system has to manage the access to the functionalities provided by each each of its elements. But, it has also to manage dynamism of the models representing the environment, which should be outside its scope. These two aspects of the system being particularly different, the implementation of HAL system has become complex, leading to a number of malfunctions.

Progress in the fields of Service-Oriented Architectures (SOA), Component-Based Software Engineering (CBSE) and Service-Oriented Component Models (SOCM), which are particularly adapted to distributed and open problems, is likely to lead to a "Home Monitoring System" made up of bundles providing the different sensors and actuators. Those bundles could, with embedded algorithms of intelligence, make joint decisions with respect to the operating requirements of the appliances to which they are connected.

A work close to this domain is that of Eclipse Smart Home project (http://trac.edgewall.org/wiki/TracDev/ComponentArchitecture), a framework for smart home and ambient assisted living which focuses on heterogeneous environments. Its objective is to provide a uniform access to devices and information and to facilitate different kinds of interactions with them. However, it is implemented in Java, upon the Equinox framework, i.e. an SOA framework implementing the OSGi specifications.

There are many frameworks for deploying and developing component-based python applications. Whilst some of them provide an interface for mature and well-established component models (such as CCM, EJB, Koala and Fractal), others are pure Python component models (e.g. Envisage\(^1\), Trac\(^2\), yapsy\(^3\), SprinklesPy\(^4\), etc). Most of them rely on a global registry, which acts as a broker, allowing components to register themselves and dynamically resolve their dependencies. Even so, among these component models, any of them take into account components’ dynamic availability: they are suited to build an application, but not to handle dynamic changes in the composition, neither to remove components.

Another example is the Zope Component Architecture (ZCA) (?), a widely used component framework, which defines the concept of interfaces in Python. Components are bound according to their interfaces and to their names. The substitutability of components is based on the components name, which means that when a named component disappears, a new one with the same name and interface must be instantiated to restart the dependent components.

Those frameworks weren’t suited for the next step of PREDIS/MHI. This software requires a specification-based substitutability, and it must handle very dynamic environment. This is why the iPOPO(?) framework, an SOCM framework, was selected. This platform has been chosen for several reasons, including:

- The drivers of sensors were already developed and provided in Python.
- iPOPO is inspired on the OSGi specifications, ensuring a robust SOA API.
- It supports Remote Services natively, allowing to turn the application in a distributed architecture easily.
- iPOPO combines many advantages:
  - Simplicity: iPOPO provides a very simple development mode. It provides an excellent foundation for building dynamically extensible Python-based applications.
  - Power: Supporting the principles of OSGi services in Python, iPOPO provides many other features that aim to simplify developing sophisticated applications. For example, iPOPO supports configuring components, notifies of component events, provides a publish-subscribe service, etc. It also provides an embedded HTTP server, specified as an HTTP service,

\(^1\)https://svn.enthought.com/enthought/wiki/EnvisageThree/core.html
\(^2\)http://trac.edgewall.org/wiki/TracDev/ComponentArchitecture
\(^3\)http://yapsy.sourceforge.net
\(^4\)http://termie.pbworks.com/SprinklesPy
that will heavily simplify the implementation of PREDIS/MHI.

Performance: iPOPO is small and is designed to stay small. The core size of iPOPO is approximately 10k. It has been optimized to instantiate a large number of components in a short time.

In the next section, the principles of iPOPO framework are presented.

3 THE IPOPO FRAMEWORK

3.1 Main Concepts

iPOPO is a Service-Oriented Component Model (SOCM), i.e. components provide are bound by services. The framework is divided into two parts: the iPOPO Core, which handles the components, and a lower layer, Pelix, which handles the services and the bundles. Pelix aims to implement part of the API specified by the OSGi standard, whereas iPOPO aims to provide features similar to the ones of iPOJO, a Java SOCM based on OSGi.

3.1.1 Pelix

A bundle is a Python module, which does not execute code during its loading process. It can define a bundle activator that will be called during the bundle life cycle steps. Pelix provides methods to install, start, stop, update and uninstall a bundle at runtime, allowing a very dynamic approach in Python.

A service is an object instance registered in a centralized registry and combined with the list of the specifications it provides and a set of properties that describe its state and features. Each service has a constant integer identifier, unique inside a Pelix framework instance, and it can have a rank that gives it a priority level in the service registry. A consumer will request the framework the list of services that match a specification and a property filter. This list will be sorted according to the ranking and to the identifier of the matching services.

3.1.2 iPOPO

A component is an instance of a factory class, coupled with an instance manager which will control its life cycle, and a set of handlers which will inject its dependencies and register the services it provides. Those handlers are linked to the component and are managed by the instance manager. The factory is class manipulated by a set of decorators, which contains all the information needed to start a component instance and bind it to its execution environment. A component instance is associated to a unique name, constant during its life time. It can specify properties, which describe its state, its capabilities and its configuration, and which can be updated by the component itself. After its instantiation, a component can be either valid, i.e. all its required dependencies are bound, or invalid. The instance itself cannot modify its own state: its life cycle is controlled by its manager.

iPOPO is based on the concept of injection of dependency, i.e. the dependency handler injects directly into the component instance the services matching the requirements described in the component factory. A requirement is a set of properties describing the service to inject, indicating the specification of the injected service, a filter for its properties. It is also used to configure the behaviour of the dependency handler, i.e. if the service must be injected for the component to be valid, if the handler must inject a list of services instead of the first matching one, etc. The dependency handler is notified of service events by the service registry of the Pelix framework, and injects or removes services accordingly.

To reflect component changes, the service providing handler propagates every change of component property fields to the services provided by the component. Component properties are declared in the factory as a link between a field in the component instance and a property name. As the provided services are updated, the dependency handlers will be notified of those changes by the framework and will update the state of their associated component.

As component are bound by services, using the dependency handlers, they can be easily replaced by other components providing similar specifications and properties. When a service provider disappears, the dependency handler will try to replace it by a matching one and notify the component instance of this modification. If there is no matching service available, the handler changes to an invalid state, and the component will be invalidated by the instance manager.

A component factory, i.e. its implementation, can be updated at runtime by updating the bundle providing it. During this operation, the component instances are deleted and the consumers of their services runs a new binding loop. Those consumers will be either invalidated or bound to other implementations of the required specifications. If the bundle update succeeds, the new version of the component instances will be instantiated, and previously invalidated components will be bound to them; else, the previous version of
the bundle is kept and previous components are instantiated a new time.

3.2 Life Cycles

3.2.1 Bundle

The life cycle of a bundle depends on the one of its host framework. As described in Figure 3, the possible states of a bundle are:

- Installed: the Python module corresponding to the bundle has been found and parsed. If no error occurred, the bundle goes immediately in the next state.
- Resolved: the bundle cannot provide any service in this state, but others can access the members of its module. If the bundle was active, this states means it has been stopped: its activator is called and shall unregister its services.
- Active: the bundle has been started, i.e. the activator of the bundle has been called and may have registered services.
- Uninstalled: the bundle has been unloaded from the framework. A bundle is always stopped before being uninstalled, i.e. this state always follows the Resolved state. The bundle cannot be restarted after being uninstalled: it must be reinstalled.

![Figure 3: Pelix Bundle State Diagram](image)

When changing states from Resolved to Active (or from Active to Resolved), the bundle passes in a transition state: STARTING (or STOPPING). In this states, the bundle has the same rights than in the Active state, but the framework will be more permissive in case of error.

3.2.2 Component

In Figure 4, an iPOPO component can be in one of the following three states:

- Instantiated: the component object is coupled to an instance manager and all its handlers have been initialized. The component stays in this state while at least one of its handler is in an invalid state.
- Valid: all its handlers are in a valid state, e.g. all required dependencies have been injected.
- Killed: the component is being deleted and all its handlers are stopped.

![Figure 4: iPOPO Component State Diagram](image)

A component can be notified of events concerning its life-cycle, by defining callback methods:

- Validate: when the component comes into the Valid state.
- Invalidate: when it comes back to the Instantiated state. It is called only if the component was in the Valid state before.
- Bind: when the component has been bound to a service it depends on.
- Update: when a service bound to the component has updated its properties
- Unbind: when a service that was bound to the component is unregistered.

The life-cycle of a component is dependent of the one of the bundle that provides its factory, and it is handled by the instance manager associated to this component.

3.2.3 Events

The dynamism of the framework implies the triggering of many events.
The framework notifies bundle listeners each time a bundle is installed, removed or changes its state. For example, the iPOPO core service listens to those events in orders to load the factories provided by a bundle as soon as it is started, and to kill all of their component instances when it is stopped.

When registering for service events, listeners are notified when a service is registered, unregistered or if its properties have been updated. It is also possible to add a filter, in order to be notified of events only about services that match it. That is how the iPOPO handler managing the dependency injection can react upon the evolution of the other components. Services events also allows to implement the whiteboard pattern: instead of having consumers that will register to notifications from a provider, the latter can wait for the apparition of listeners according to their service. That way, the provider always an up-to-date list of listeners, possibly managed by iPOPO, while the consumers do not even have to know about the provider: they can just be notified of events by providing the right service.

It is also possible to listen to iPOPO events, i.e. when a factory has been registered or removed, or when the state of a component evolves.

Finally, framework listeners are notified right before the framework beings its stopping process. This is the situation of the iPOPO core service, which refuses to instantiate while the framework is shutting down.

4 IPOPO-BASED INTERFACE FOR PREDIS/MHI

4.1 iPOPO-based Implementation

The objective of this work is to design a Home Monitoring System on a dynamic, modular and configurable basis. Figure 5 shows the new design of HAL system, based on the notion of services. The drivers already developed in Python, in the centralised version, are separated and encapsulated into bundles and transformed into components. This operation is done in two steps: (a) moving all the module initialisation code into the bundle activator and (b) decorate the classes to let iPOPO consider them as components, providing a service according to their features. The sensors and actuators are also represented in components providing probing or control services and consuming driver services.

The new version of HAL system based on iPOPO framework is installed in the PREDIS platform.

Figure 5: HAL system service oriented architecture

In this new implementation of the HAL system, most of the raised problems in the previous version have been resolved: This version charges only access to the functionality of its elements environment, and it is based on iPOPO to manage its dynamic model. This model greatly simplifies the code base, by limiting the functionality of the system, but also by basing on a management tool specialized for a large number of components. In addition, the dynamic capabilities of iPOPO used to manage environmental changes and configurations defined by the developer without restarting the entire system.

As mentioned, PREDIS/MHI platform is an experimental platform. Students could be testing the sensors: remove, add or change the configuration of system. With the centralized approach, whenever there is a change in the system configuration, the system must be stopped to do the new configuration change and start it again. This causes a hole in a 5-minutes data received from sensors. This data are generally used in many projects: average structure temperature estimator, occupation estimator, usage and system diagnostic analysis, etc. This data hole has a direct influence on the results of these projects; previously, an estimator of data was used, but it does not always work. Actually, as the new version of HAL is installed, the system configuration is almost instantaneous: there is no more data hole and the results are more interesting.

Recently, an EnOcean driver has been installed in the PREDIS/MHI platform.

The integration of this driver in the HAL system was easily done and has no effects on the overall system running. On the other hand, the new version allows the debugging of codes in real time: some modification can be done and uploaded to the HAL system without penalising the overall system running.

It is important to note that while the new version of
the HAL system is more modular and has must more features, including dynamic platform updates, it kept the same performances than the old one, in terms of response time as well as memory consumption.

4.2 Discussion

It is difficult to adapt the centralised approach to real home automation contexts as it is not suitable for frequent and variant configurations / reconfigurations. This means that it is not possible to have an open-ended system unlike the service-oriented system where appliances can be added or removed without having to reconfigure the system and without penalising the overall system running that has to be potentially capable of taking on board all kinds of requirements. The service-oriented system based on iPOPO/OSGi techniques allows for extendibility: new types of appliances can be added to the system without it being necessary to entirely reconfigure it. With the centralised system, the platform algorithms must be changed to take into account new appliances, whereas the service-oriented implementation adapts automatically to the type of home automation system, i.e. the physical distribution of different types of sensors/actuators.

5 CONCLUSION AND PERSPECTIVES

Service-Oriented Component Models are an excellent way to ensure that developed software comes with built-in best practices such as reusability, high cohesion, loose coupling, high flexibility and dynamism. The proposed system based on iPOPO does offer advantages over the centralised approach: its openness, its scalability and its capability to manage diversity.

As the new version of PREDIS/MHI global interface was developed both by PREDIS and iPOPO developers, the contributions were not for one side only. This project implied the evolution iPOPO, by bringing new use cases in terms of architecture as well as of metrics. For example, iPOPO had to deal with tens of thousands of components and services during the tests of PREDIS. This implied to rewrite parts of the service registry and of the dependency handler to drastically increase their performances in terms of execution time.

The next step in the development of PREDIS will be to use the remote services provided by Pelix/iPOPO. They will allow to dispatch parts of PREDIS/MHI near the sensors they must monitor, e.g. one in each room, by providing an easy way for them to communicate with the HAL central server. The decoupling implied by remote services will also allow to use sensors develop in different languages. A Java version of Pelix Remote Services is under development, in order to let Pelix/iPOPO components interact with OSGi/iPOJO ones.