A Dynamic and Service-Oriented Component Model for Python Long-Lived Applications

Thomas Calmant1,2 Thomas.Calmant@isandlatech.com
Joao Claudio Americo1 Joao.Americo@imag.fr
Olivier Gattaz2 Olivier.Gattaz@isandlatech.com

Didier Donsez1 Didier.Donsez@imag.fr
Kiev Gama3 Kiev.Gama@cesar.org.br

1Université de Grenoble INRIA/LIG SARDES Team Grenoble, France 2IsandlaTech Grenoble, France 3Recife Center for Advanced Studies and Systems (CESAR) Recife, Brazil

ABSTRACT
Dynamic runtime adaptations are a key feature for long-running applications. One of the most used languages for writing this kind of applications is Java, due to its reflection features, popularity and dynamism. However, as dynamic scripting languages (such as Python and Javascript) increase in popularity, it is desirable to be able to conceive long-running applications with them too. This paper introduces iPOPO and Pelix, a Python-based service-oriented component model and dynamic service platform respectively, which are inspired on two popular Java technologies for the development of long-running applications: the iPOJO component model and the OSGi Service Platform. To illustrate the approach, a usage of iPOPO and Pelix is presented on the context of mediation software.

Categories and Subject Descriptors
C.1.3 [Other Architecture Styles]: Adaptable Architectures

General Terms
Design

Keywords
Component model, SOA, Python, Dynamism

1. INTRODUCTION
Increasingly, software is required to accommodate new features or to correct behaviors after the design and deployment stages. Moreover, software needs to evolve at runtime with minimal interruptions and, when possible, never stop running. Different motivations push software design to allow such evolution at runtime. For example, production systems with critical availability requirements need to be updated with little perceived execution interruption by their users. In ubiquitous systems, appliances, devices and services can be added at anytime: software have to continuously adapt itself to take into account context changes without any human (i.e. the end-user) intervention. However, handling runtime software evolution is becoming a daunting task for software developers, architects and administrators. Numerous works propose comprehensive formalisms and frameworks to deal with runtime software evolution [6]. Component Based Software Engineering [7] is a good candidate since this approach improves software adaptability and flexibility. Nevertheless, few components models consider software evolution as a primordial concern. Examples of component models which were designed to deal with dynamic changes (software reconfiguration and update) are ServiceBinder [2] and iPOJO [3], both based on the OSGi platform [8], which is now the de facto standard for modularity in Java applications. At the same time, Python is a dynamic scripting programming language that is easy-to-learn and combines the advantages of object oriented programming, functional and procedural programming, strong typing and dynamic scripting. It also enables exploiting easily C and C++ libraries provided by the host operating systems. Several rudimentary component models were developed to structure huge Python-based software but none of them consider the requirement of the runtime software evolution. Python developers could benefit of the service-oriented approach for designing modular and long-lived applications with service-oriented component models (SOCM), as in the Java-based OSGi Framework. This paper presents a component model for Python inspired by main Java SOCMs. This new model named iPOPO (for injected Plain Old Python Object) preserves the practices of Python developers and enables the reuse of legacy Python modules and native libraries (i.e. without any preliminary refactoring). The component framework is executed by a container named Pelix (in honor to Apache Felix platform), an OSGi service platform for Python.

The remainder of the paper is organized as follows: Section 2 gives background and motivations, followed by the iPOPO component model description in Section 3. The implementation of the component container is explained in Section 4, limitations are discussed in section 5 and the validation is...
described in Section 6. Section 7 comments on related work followed by conclusions and perspectives in Section 8.

2. BACKGROUND & MOTIVATION

The Python programming language meets a successful and large adoption in the software industry. Popular applications (like Blender 3D, Dropbox), IT and web frameworks (e.g. Django and OpenERP) and large applications are continuously running in the clouds of IT major actors (such as Google). Python is an interpreted dynamic high-level language that was initially designed to be a bridge between C and shell programming [1], allowing access to system methods through the standard library, and to low level C libraries with a simple syntax, using the ctypes standard module. Although there are several component models for Python, such as the widely used Zope Component Architecture ¹, none of them is service-oriented, not allowing components to provide services that can be dynamically replaced by the service of other providers which implement the same specification.

Even if iPOPO is inspired by Java technologies, namely OSGi and iPOJO, the differences between Java and Python implied in having some different approaches. The most important change is the way to load classes, as there is a class loader object in Java, whereas a Python class is a special callable object. Also, the framework intent to be close to the OSGi and iPOJO, the differences between Java and Python are continuous running in the clouds of IT major actors (such as Google). Python is an interpreted dynamic high-level language that was initially designed to be a bridge between C and shell programming [1], allowing access to system methods through the standard library, and to low level C libraries with a simple syntax, using the ctypes standard module. Although there are several component models for Python, such as the widely used Zope Component Architecture ¹, none of them is service-oriented, not allowing components to provide services that can be dynamically replaced by the service of other providers which implement the same specification.

During the design of this new service-oriented component model for Python, some goals were established:

- Components must be sufficiently robust and dynamic to be used in mission critical applications developed in Python with C/C++ libraries;
- Components must be easy to deploy, using Python packaging alternatives: in a module, a single Python file; in a package, a tree of folders of modules; or in an egg file, a zip file containing packages, metadata files and extra resources;
- Developers do not have to handle all the components bindings and life cycle operations. This way, they can focus on the business code, reducing redundant code and potential sources of error;
- Components must be substitutable in order to enable their update or replacement by a brand-new or an alternative implementation. This can be achieved by automating the components bindings at runtime, according to a requirement description. Consequently, components only need to indicate which services specifications they depend on, allowing the framework to decide which services they will be bound to at runtime.
- Components must be able to interact with legacy modules, even closed source ones. In Python, a closed source module consists in a Python module in its platform-dependent byte code form, which cannot be easily reverse-engineered, but still is a standard module. Native modules, written in C or C++, should also be handled as closed source modules, converted to components as-

²PEP 8: http://www.python.org/dev/peps/pep-0008/

In iPOPO, components have services and requirements. Both entities refer to a specification, which will be used to verify which components can be linked together. Components may specify properties, which describe its configuration and capabilities. Services may also have properties, which are used by other components to filter among several providers. Services are kept in a centralized service registry, which is queried by the framework to fulfill the dependencies of service consumers. In fact, a service is an object instance registered in a centralized registry and combined with a list of implemented specifications and a set of properties. Each service has a constant unique ID, computed during its registration, and can have a rank property which can be modified at runtime by its registerer. Services are always sorted according first to their ranking and then to their ID, and they can be found by their specifications or using filters on their properties.

A component must have a unique name, constant during its life time, and a dictionary of properties, which may be configured at instantiation time. Concretely, a component is an instance of a factory class, coupled with an instance manager which will handle its life cycle and its dependencies. The factory contains all information needed to start and bind a service to its execution environment, and may also contain the default values of its components properties. After its instantiation, a component can be valid (i.e. all its required dependencies are bound) or invalid. The instance itself cannot modify its own state, thus all the life cycle process is handled by its manager. Upon the binding (or unbinding) of a dependency, the instance manager uses (or removes) it from the component and notifies the latter, in order to let it configure its newly coupled resource. In addition, the instance manager must propagate any property field change in the component and trigger the corresponding

Figure 1: iPOPO metamodel

In iPOPO, components have services and requirements. Both entities refer to a specification, which will be used to verify which components can be linked together. Components may specify properties, which describe its configuration and capabilities. Services may also have properties, which are used by other components to filter among several providers. Services are kept in a centralized service registry, which is queried by the framework to fulfill the dependencies of service consumers. In fact, a service is an object instance registered in a centralized registry and combined with a list of implemented specifications and a set of properties. Each service has a constant unique ID, computed during its registration, and can have a rank property which can be modified at runtime by its registerer. Services are always sorted according first to their ranking and then to their ID, and they can be found by their specifications or using filters on their properties.

A component must have a unique name, constant during its life time, and a dictionary of properties, which may be configured at instantiation time. Concretely, a component is an instance of a factory class, coupled with an instance manager which will handle its life cycle and its dependencies. The factory contains all information needed to start and bind a service to its execution environment, and may also contain the default values of its components properties. After its instantiation, a component can be valid (i.e. all its required dependencies are bound) or invalid. The instance itself cannot modify its own state, thus all the life cycle process is handled by its manager. Upon the binding (or unbinding) of a dependency, the instance manager uses (or removes) it from the component and notifies the latter, in order to let it configure its newly coupled resource. In addition, the instance manager must propagate any property field change in the component and trigger the corresponding

Figure 1: iPOPO metamodel
service change event in the framework. This will trigger a new binding loop to update all depending components state. The requirements definition is an element in the factory description. Flags indicate if a requirement is optional (that is, the component is considered valid without fulfilling this requirement) and if it corresponds to an aggregation of services (so that all matching services found in the register must be injected as a list). It may optionally contain a service filter too, which can be updated at instantiation time for a specific component. The substitutability of components is ensured by the service-oriented approach. In iPOPO, components are bound through services, which means that a component can be bound to any other component providing a service matching its requested specification and properties. While the service provider is available, the service consumer will be bound to it, but if it disappears, a new link to another provider can be established (if there are matching services in the register, otherwise the component will be invalidated). In addition, the instance manager listens to all incoming and outgoing services to update the state of its component. The model also allows component updates easily, by removing its previous version and starting a new one: while it is absent, its consumers may be bound to another implementation or be invalidated, causing the unavailability of the services provided by consumers, waiting for a new version to be instantiated.

3.2 Component Life Cycle
An iPOPO component can be in one of the following three states:

- **Instantiated**: The component object exists and is coupled to an instance manager. Its provided services are registered.
- **Validated**: All required dependencies of the component have been injected. Its provided services are registered and accessible to other components.
- **Killed**: The instance is being deleted and cannot be validated anymore. The provided services are unregistered and the component is removed from the registry.

A component can define four life cycle callback methods:

- **Bind**: called after the injection of a required service
- **Unbind**: called before the removal of a required service
- **Validate**: called after the injection of all non-optional required services and after the registration of the component services.
- **Invalidate**: called before the removal of a non-optional required service and before the the unregistration of a component service.

As it based on specifications and filters, a component can be bound to any service matching those requirements. That way, a specification implementation, provided by a component or a service, can be replaced by another one, causing an unbind and a bind event, respectively. If the specification was not optional, the component will also be invalidated then re-validated to ensure that its state is always semantically valid.

A common case is the interaction between three components. Two services with the same specification are provided by components A and B and a third component C requires a service with that specification. When A is instantiated, it registers its service in the framework, which triggers a registration event handled by the instance manager of C. The service is injected in the instance of the component C, which is then validated. When B is instantiated and registers its service, the instance manager of C is notified, but that service is not injected, because the requirement is already fulfilled. When A is killed, its service is unregistered and an event is triggered then handled by the instance manager of C. As A is an injected required service, C is first invalidated, then it asks the framework for a service with the same required specification. The service provided by B matches with the queried specification, thus it is injected in C, which is then validated again. The invalidation step is important to keep the coherence of the component C, avoiding calls to its service while one its dependencies is missing. Also, component A is invalidated after its service has been unregistered and after being unbound by all its consumers. That way, all dependent components are guaranteed to have access to a coherent service during the unbind operation.

4. IMPLEMENTATION

IPOPO was inspired by OSGi and iPOJO, which are respectively a service framework and a service-component model, both Java-based. In this section, the service platform for Python and the component model framework are described.

4.1 Pelix: A Python-based OSGi Platform

Pelix, named after Felix, an OSGi framework by Apache, implements the service handling mechanisms using an API similar to the OSGi specification. It provides a service registry, which can be browsed by specifications or by a filter on services properties. Pelix allows using listeners to register to service events, which are triggered when a service is registered, unregistered or when its properties changes. The main advantage of having such a service framework, separated from the component framework, is that a developer can register a manually handled service which will be accessible by the components.

4.1.1 Services

Services are objects that provide functionality, associated to properties stored in a service registry. Each service is associated to a unique ServiceRegistration and a unique ServiceReference instance. The registration object should be kept by the entity that handles the service life cycle, and may not be shared. It provides only way to modify properties of a service and to unregister it. Service consumers ask the framework for references matching a specification and/or a service filter, and then ask it again for retrieving the given service instance. This pattern allows the consumer to access the service information before using it.

4.1.2 Deployment unit

In OSGi, the deployment unit is the bundle, a Java archive file (JAR) containing Java classes, resources and a description file, the manifest. In Pelix, a bundle is a Python module file, that is to say a source file (.py) or a compiled file (.pyc) containing classes definitions and possibly an activator that is deployed by the framework. As it is possible in Python to execute code while a module is loaded, any deployment time functional code should be avoided except the activator instantiation. An activator is an object instance with a
start and a stop method, allowing the bundle to register services and listeners, and to work with the framework. Each bundle is represented by a Bundle object, containing information of the services it has registered, and associated to a BundleContext object which allows the bundle code to work with the framework. A bundle is installed on request by the framework, and must be started manually. Its life cycle can be handled by any bundle in the framework, but when it is uninstalled, its remaining registered services are automatically unregistered. As it is a Python module, a bundle can be updated using the standard library which allows an inplace update of a module, re-interpreting it, and which gets back to the previous version in case of an import error. The current implementation does not handle dependencies at the bundle level. This means that there is currently no bundle wiring resolution, it must be handled by the developer. The bundle concept will be completed, as it is discussed further in the perspectives section. Nevertheless, the dependencies between bundles are way less important than in OSGi. As Python uses dynamic typing, there is no need to share an interface between a service and its consumers. The semantics used by the consumer will be validated when executed, not during a pre-validation time. That way, only constants definitions might be shared between bundles, which can be accessed using the bundle object retrieved from the framework by its symbolic name. The way modules are imported by the framework makes them unique per Python interpreter instance, not per framework. While a new loading way is not implemented, it is impossible to use more than one framework at a time per interpreter instance.

4.1.3 Bundle life cycle

A bundle life cycle depends on the life cycle of its host framework. The possible states of a bundle are:

- **INSTALLED**: the bundle has been installed and its module is accessible. The bundle does not provide any service. If the bundle was active, its activator is called and shall unregister its services.
- **ACTIVE**: the activator of the bundle has been called, its services may be registered.
- **UNINSTALLED**: the bundle is unloaded from the framework. If the bundle was active, it would be changed to the INSTALLED state in order to let the bundle unregister its services. The bundle cannot be re-activated after being uninstalled.

4.1.4 Events

Each state modification of a bundle or a service triggers an event, sent by the framework to registered listeners. While bundle listeners are called back for all event, service listeners can also add a filter to be notified only when the matching subset of the service registry has changed. Finally, the framework is able to call back registered listeners when it is preparing to stop, before stopping installed bundles.

All events are synchronous, which means only one event is triggered at a time. It is not guaranteed that listeners will be notified in the same order than events are triggered, but it will remain semantically correct according to the life cycle of the bundle or service raising the event.

4.2 The iPOPO Framework

The iPOPO component model framework relies on Pelix to handle the components provided and requested services. It must be installed in a Pelix framework, like any other bundle. This pattern simplifies the relations between iPOPO and Pelix, as the iPOPO bundle life cycle depends on the framework one, and because the iPOPO bundle’s activator can stop all components when it is stopped, cleaning up the state of the application. The activator is also used to register a bundle listener, allowing to register or remove bundles’ component factories dynamically.

An iPOPO service is registered in Pelix as soon as its bundle is started. Its service contains two methods: one allowing a consumer to instantiate a new component, indicating a factory name, an instance name and a set of properties; and another to kill a component using its name. In the current implementation, a component is an instance of a manipulated Python class, called a factory class. The instance manager is instantiated right after the component, before it is configured. This manager contains the component properties dictionary, and is called by the fields injected in the component during its class and its instance manipulation. It is also a service listener that injects references to required service instances directly in the component fields and class the registered component callbacks methods accordingly. The current implementation is not optimized: each time a service event is triggered, all components instance managers are notified and each one tests all of its associated component fields if they need to be updated. The manipulation consists in setting up and injecting in the class a FactoryContext object, which contains the description of the component type as defined in the previous section, by using standard Python mechanisms. Python decorators were developed in order to manipulate the class, which are called during the module import, and before the framework considers the bundle as installed. They are callable classes or methods, which means they can be both used in the module code, decorating classes like Java annotations, or programmatically, instantiating decorator classes and calling them recursively on an imported class. This way, a class from a close source or a legacy module can be manipulated and converted into a component. An instance manipulation is done right after the component instantiation to add the instance-level methods, like properties getters and setters depending on the instance manager instantiation. Listing 1 shows how to apply the iPOPO decorators on a component class.

The implementation of Pelix and iPOPO is fully compatible with Python versions from 2.7 to 3.2. The implementation has been tested with the following interpreters, on Linux and MacOS X:

- cPython 2.6, 2.7, 3.1, 3.2
- Pypy 1.8, supporting the Python 2.6 specifications

Using the backport of the importlib module, the code can also be executed by a Python 2.6 interpreter. Due to significant syntax modifications between the 2.5 and 2.6 versions, the code cannot be executed by older versions. IPOPO has also been successfully tested on an Android 4 phone, with the Python 2.6 interpreter available with the SL4A project (Scripting Layer for Android). Two types of test suites were executed: unit tests, which cover 85% of Pelix and iPOPO source code; and a validation test with 512 components wrapping C libraries. Since Jython still does not...
supports the 2.6 Python specifications, it cannot run this implementation.

5. DIFFERENCES AND LIMITATIONS

IPOPO has been inspired by the pair OSGi / iPOJO, both originating from the Java world. OSGi is a service framework, while iPOJO is an OSGi-based service-oriented component model.

5.1 Pelix versus OSGi

Pelix corresponds to a subset of the OSGi specification, close to OSGi Release 1, but without the configuration nor the security parts. Python standard library does not provide access policies, which are needed to implement the security aspects, but an implementation exists in the Zope project. There are many differences in the packaging handled by OSGi and by Pelix. In OSGi, bundles are JAR files, a ZIP file containing Java classes, a Manifest file and extra files like native libraries. An entry in the Manifest file, Bundle-NativeCode, allows to indicate which native file shall be used for the current operating system. Currently, Pelix only supports Python modules, containing classes and methods. Future versions will support Egg files, which are similar to Java JARs, that is to say a ZIP file containing Python packages and modules, a Metadata file and extra files. Also, most of the Python libraries are distributed with a specific egg file for each supported platform, if they contain native code.

5.2 IPOPO versus iPOJO

IPOPO has a flat model, i.e. it does not define any kind of components hierarchy, but this can be done using services properties and requirement filters. Moreover, components cannot be aggregated into a bigger structure (such as composites in iPOJO and SCA [5]). In addition, IPOPO does not allow the definition of binding policies, even though its current policy can be considered as iPOJO’s dynamic policy. IPOJO provides an Architecture Definition Language (ADL) that allows to describe a complete assembly of components and composites, sub-sets of components. IPOPO does not yet provide any kind of ADL, every instantiation must be done using a decorator or the IPOPO service.

While in iPOJO, extra functional properties are managed by handlers, themselves may be iPOJO components, while the current IPOPO model definition does not define any extensibility entry point for them. This might be added in a future version as discussed in the perspectives.

6. VALIDATION

The framework was validated in a project from a company called IsandlaTech\(^1\). IPOPO has been used in the software development kit and a runtime platform they have created, called PSEM2M, to produce long-lived software, allowing updates and reconfiguration with a minimum impact on the availability of the provided services. The first IPOPO projects developed by IsandlaTech suggest that using iPOPO implies a performance overhead during the initialization of the application, but not during its operation.

Long-Lived Experiment

Companies increasingly need to easily and quickly bind their ERP systems to third-party applications, mobile devices or industrial devices. When those bindings are remote services calls, they should rely on a fail-safe middleware to ensure seamless processing of all of the requests received. Otherwise, a part of the whole activity of the company may be interrupted. An example is an e-commerce website that must interact with the services of an ERP to retrieve information of items for sale (price, availability, and so forth) and record customer orders. If the update of a part of architecture induces to stop the service for a significant time, the company may lose sales. It is possible to resolve this issue by adding a component-based long-lived mediation software to the website, that can temporarily replace the actual services of the legacy ERP. The challenge is to produce a reliable software, without prohibitive cost, that can be reconfigured and updated without interrupting the services it provides.

The implementation is based on components chains, one for each ERP service. The reusability of services is important to reduce development costs, therefore all components have many configurable properties, to describe their per-instance behavior. The validation project is made of two parts, each one running in the same PSEM2M container: a Java OSGi-based part (Apache Felix), with iPOJO components; and a Pelix-based one, with iPOPO components. Each chain is executed in an individual process, called isolate, monitored by the framework which restarts it automatically on a failure scenario. Inside each isolate, a composition agent waits for orders from the framework monitor, to instantiate or kill components. All communications between isolates are done using JSON-RPC. The website sends its requests to the mediation software endpoint, a Java component, which will use the corresponding chain. Each chain first tries to propagate the request to the ERP, then uses its local business components on failure. Using iPOPO mechanisms, the Python components can be easily updated, with a really small time of unavailability, caused by the update itself and the binding re-computation. This can be masked by the preceding component in the chain, which can wait to be bound again to the updated component before calling it.

\(^1\)http://www.isandlatech.com
That way, the update corresponds to an increase of the processing latency, instead of a service failure. This validation project shows that the model is able to ease the development of business components, avoiding the whole life cycle and binding parts.

7. RELATED WORKS

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 propose an environment containing native Python component models (e.g. Envisage\(^5\), Trac\(^6\), yapsy\(^7\) and Sprinkle-sPy\(^8\)). Most of them contain a global registry, which acts as a broker, allowing components to register service definitions and dynamically resolve their dependencies. Even so, amongst these component models, any of them take into account components’ dynamic availability. Zope Component Architecture (ZCA), a widely used component framework, defines the concept of interface in Python and of components, bound by interfaces and by 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. As a comparison, iPOPO components are bound by specifications, which can be compared to the ZCA interfaces, and to properties, instead of a name, which ease substitutions as two different components, active at the same time, can have the same specifications and properties. The Colony framework\(^9\) is an OSGi-inspired framework which allows for its deployment units (called plugins) to be installed, started, updated, stopped and uninstalled without requiring a platform restart. As the OSGi framework, its implementation is heavily based on the inversion of control principle. It also has distribution and language interoperability features. However, Colony does not contain the dynamic service layer, which enables components to interact in a SOA fashion. The Picolo framework [4] is an extensible minimal component framework. Picolo component are assemblies of operations (provided methods), properties and ports (bindings) to an empty component shell. The components and their bindings can be described programmatically or using an XML description. The substitution of a component in a binding is done by replacing programmatically the output of a port by another component. Picolo components can only be selected by a name, not by a specifications and filter couple, which reduces the capabilities of automatic dependency injection.

8. CONCLUSION AND PERSPECTIVES

This paper introduces the first version of the iPOPO model, a service-oriented component model for Python. It also presents how it has been implemented, describing Pelix, an OSGi-like SOA platform, and the iPOPO framework, the component manager. IPOPO allows the development of loosely coupled applications and dynamic applications in Java. Moreover, since Pelix enables hot deployment of components on the platform, it is also possible to develop long-running modular applications. The whole framework is small, having less than two thousands lines of code. The component framework was validated in a demonstration project which mixes Java and Python components. Many things are planned to be added or improved in a near future. First, the bundle concept must be improved, in order to support more scalable deployment units, like Python egg files, and to support multiple versions of the same deployment unit in a framework instance. The model extensibility must be enhanced to handle non-functional properties and to provide different binding policies, to extend the dependency injection handling, etc. Finally, it may be interesting to let a component use flags to control its provided services states, in order to register or unregister them according to its internal state.

The iPOPO framework is being more and more integrated in the IsandlaTech PSEM2M platform, a commercial framework to deploy and execute components in individual processes. Many of the above perspectives will be implemented in this context. The implementation of iPOPO is available under GPLv3 license at http://ipopo.coderxpress.net/.

9. REFERENCES