Copyright 2011 KSII

iPOJO-based Middleware Solutions for Self-Reconfiguration and Self-Optimization

Paolo Bellavista[1], Antonio Corradi[1], Damiano Fontana[2] and Stefano Monti[3]

[1] DEIS, University of Bologna, V.le Risorgimento, 2 40126 Bologna, Italy
[e-mail: paolo.bellavista, antonio.corradi @unibo.it]
[2] Dept. Engineering Science and Methods, Univ. Modena and Reggio Emilia,
Via Amendola, 2 42122 Reggio Emilia, Italy
[e-mail: damiano.fontana@unimore.it]
[3] EPOCA s.r.l, Via Parigi, 11 40121 Bologna, Italy
[e-mail: stefano.monti@epocaricerca.it]
*Corresponding author: Paolo Bellavista

Received May 17, 2011; accepted July 6, 2011;
published August 29, 2011

Abstract

In recent years, ubiquitous and pervasive scenarios have emerged as a complex ecosystem where differentiated software/hardware components interoperate wirelessly and seamlessly. The goal is to enable users to continuously access services and contents, and to always get the best out of their current environment and available resources. In such dynamic and flexible scenarios, the need emerges for flexible and general solutions for continuous runtime self-reconfiguration and self-optimization of ubiquitous support software systems. This paper proposes a fully reconfigurable middleware approach that aims at reconfiguring complex software systems made up of heterogeneous off-the-shelf components from both functional and non-functional perspectives. Our middleware can also extend already existing and non-reconfigurable middleware/applications in an easy and flexible way, with no need to re-design them. The proposed design principles have been practically applied to the implementation of a runtime self-reconfigurable middleware called Off-The-Shelf Ready To Go (OTS-RTG), implemented on top of iPOJO. The reported experimental results both exhibit a limited overhead and show the wide applicability of the proposed solution to many application scenarios, including complex, industrial, Enterprise Service Bus-based ones.

Keywords: Middleware, self-reconfiguration, iPOJO, autonomic computing, enterprise service bus (ESB)

A preliminary version of this paper appeared in ACM ICUIMC 2011, February 21-23, Seoul, Korea [20]. This version relevantly extends that work by showing how to adopt (and with which performance results) the proposed model and middleware solution into industry-level provisioning environments where services are orchestrated via Enterprise Service Bus frameworks.

DOI: 10.3837/tiis.2011.08.001

1. Introduction

Requirements of Ubiquitous Computing (UC) scenarios have relevantly grown, and applications more and more frequently have to adapt themselves to user needs and characteristics (e.g., currently used access terminal) and to environmental conditions (e.g., network connectivity type, status, user location) at runtime [1]. Middleware-based approaches are the key to simplify the design of applications in these scenarios and to keep the support infrastructure flexible and adaptable. A set of low-layer support facilities can provide the basis to build, add, aggregate, and coordinate novel and more complex application components on top of them, hence promoting modularity, reuse, and dynamic extension/tailoring. Nevertheless, the ever increasing heterogeneity and the dynamic nature of UC scenarios make runtime system reconfiguration/adaptation play a crucial role in providing users with applications tailored to specific preferences and environment conditions. Reconfiguration is becoming of paramount importance to redesign end-user functional application logic (e.g., service and content aggregation/provisioning), as well as to adapt and optimize internal non-functional building blocks of the middleware itself.

For instance, users increasingly exploit their mobile phones to access aggregated information, such as RSS feeds from their preferred social networks, mixed up with an up-to-date map of their online friends. Such a scenario intuitively stresses the application components in charge of aggregating RSS feeds and adapting/translating them to smartphone-enabled formats (e.g., dynamically resized to fit the terminal display). Under heavy load conditions, some adaptation components should be replaced by less accurate, but faster and more lightweight ones. In addition, application components may intensely use non-functional support features, such as, for instance, a persistence layer (backed by a relational database), an asynchronous messaging framework, or even an Enterprise Service Bus (ESB) infrastructure for the integration and orchestration of heterogeneous application components. These support modules, too, have a relevant impact on the overall UC scenario and may deeply influence the experienced Quality of Service (QoS) and responsiveness. This simple example clearly highlights that adapting and reconfiguring only some application-level components may become limiting in many practical industrial UC applications. In particular, we claim that real and wide-scale UC service provisioning scenarios call for a deep, coherent, and cohesive system reconfiguration strategy, which involves an integrated adaptation of both application-level components and non-functional support features.

In addition, UC scenarios are biased toward spontaneous interactions of end-user mobile devices, differentiated forms of wireless connectivity, and opportunistic sharing of infrastructure resources (wireless multimedia hubs, shared printers, ...). Those rich forms of interactions ultimately call for UC middleware that can arrange already existing off-the-shelf software/hardware components and can make them cooperate into a larger and value-added scenario. There is someway an analogy with the situation of a chef who wanders through a supermarket looking for the exact ingredients for her next big recipe. But what if an ingredient is not currently available? Or, even worse, what if the chef runs out of an ingredient just while cooking? Only a really creative and experienced chef could be able to pick up and concoct different ingredients to replace the lacking one, or to massage the original recipe to make the most out of the available ingredients. In UC scenarios, software and hardware "ingredients" (devices, network connections, physical resources, …) continuously become available and disappear, forcing a "chef" middleware to continuously adapt its "recipes" to the available resources, environmental conditions, and user needs.

System reconfiguration is a long debated research field and has been investigated for a broad range of dynamic systems, including UC-related ones: some relevant conceptual models and significant architecture solutions have been proposed in the last years with the primary goal of UC middleware reconfiguration. However, on the one hand, existing proposals are usually limited to application-level reconfiguration and ignore non-functional feature adaptation. On the other hand, reconfiguration solutions usually require infrastructure components (both application and non-functional features) to be explicitly conceived to integrate reconfiguration opportunities from the beginning. That basically hinders integration of existing and non-reconfigurable components, thus becoming a relevant limitation in open landscapes of heterogeneous services. We claim that a truly comprehensive UC middleware should consider adaptation/reconfiguration as one of its central design principles. Therefore, this paper presents a reconfiguration middleware that i) provides reconfiguration facilities for both functional and non-functional middleware/application components, and ii) allows to extend already existing components with adaptation/reconfiguration features. In addition, we claim that the main issue in realizing such a flexible and dynamic middleware relies in the need of keeping a consistent and up-to-date view of system inner state and behavior, so as to decide how to effectively react to dynamic changes in the execution environment.

In particular, our approach envisions an off-the-shelf paradigm where a self-reconfigurable middleware can continuously rearrange already available (functional and non-functional) components into value-added and tailored aggregations. Our Off-The-Shelf Ready To Go (OTS-RTG) middleware adopts the novel iPOJO [2] implementation of the Service-oriented Component (SoC) paradigm [3][4]. Our proposal puts together the formalization of service contract definition, typical of service-oriented architectures, and the inherent component lifecycle, dependency, and composition management, which are traditional in component-oriented approaches. In our idea, middleware components declare service interfaces (contracts) to describe their behavior and state: those features drive how components cooperate with each other and how to influence component behavior (for instance by tuning operational parameters). Thus, for instance, when a platform component becomes either unsuitable or unavailable, our middleware may exploit component features either to look up for other components declaring the same interface and replace them, or to reconfigure the inner status and behavior of appropriate components. This approach pushes usual reconfiguration middleware approaches one step further and allows to reconfigure functional elements (e.g., components carrying out business logic) as well as non-functional components (e.g., elements offering horizontal support features). Moreover, the proposed middleware has been implemented, validated, and experimentally evaluated over different deployment scenarios of practical industrial interest, including ESB-based integration solutions. The paper finally reports lessons learned from practical deployment experiences and quantitative performance evaluations in these real testbeds.

2. Background and Related Work

This section describes related work about reconfigurable models/architectures and presents some implementation solutions that partially enable the dynamic reconfiguration of large-scale complex systems, with specific focus on UC scenarios.

2.1 Reconfiguration Model

Reconfigurable systems strive for the capability to dynamically and automatically change their structure in terms of components and aggregations (i.e., compositions) of cooperating entities

that provide functional and non-functional features. The main aim of reconfigurable systems is to self-tune and automatically self-adapt to ever changing conditions, e.g., to offer differentiated and proper QoS levels in any different situation. Hence, a reconfigurable system should analyze its execution context and adapt itself to specific runtime context conditions, by changing its structure and by reconfiguring its internal components.

In recent years, different architectural models have been proposed to address the reconfiguration issues of middleware for large-scale deployment environments. The Autonomic Computing (AC) initiative, initially proposed by IBM [5][6], has emerged as one of the key proposals. AC identifies some architecture design guidelines to realize systems that can react to context changes and reconfigure themselves almost as autonomously and automatically as the human nervous system can coordinate, regulate, and adjust all elements of the human body. AC architectures are typically inspired to closed-loop control theory, and are capable of reacting to events from their operating surrounding to introduce corrective adaptation actions. In the IBM proposal [7], the control loop got the name of MAPE-K (the loop consists of four main stages - Monitor, Analysis, Plan, and Execute): during the Monitor phase, sensors monitor different system parameters; the collected values are used in the Analysis phase to detect problems or any other specific system state; in the Plan phase, the autonomic system creates a complete executive plan for self-optimization via available effectors; the Executive plan is expected to be run at the execute stage and the effectors can modify system parameters at that time, e.g., by replacing a component without stopping service provisioning of the composition where it was included and by re-setting the properties of the other components.

2.2 Implementing Reconfiguration

By focusing on the implementation challenges of typical reconfigurable systems, two main directions of solution have emerged in the literature: reflective middleware approaches and SoC model ones. Reflective middleware solutions aim at providing the application layer with mechanisms and tools to introspect/modify middleware implementation details [8][9]. Reflective middleware achieves its goal by exposing and keeping a consistent middleware self-representation, in order to make the middleware internal state/behavior accessible and modifiable at runtime. Applications willing to modify middleware components/behavior can change the middleware self-representation via reflective APIs: changes are dynamically propagated to the current middleware implementation to reconfigure its state and behavior.

Reflection techniques are support features that come at a non-negligible cost in terms of computing resources. Any reflective component should maintain an internal meta-level description available to external controllers: keeping such meta-level description consistent with component behavior is complex and imposes some runtime overhead. For instance, [8][10] propose a reflection-based approach to AC middleware: although the reflective approach can be powerful in terms of monitoring and reconfiguration capabilities by allowing a fine-grained control over system components, reflective middleware usually suffers from limited performance and induces an increased management complexity. For these reasons, its adoption is limited, especially in industry-relevant wide-scale applications. In addition, the reflective approach usually forces to redesign middleware from scratch to add AC capabilities, because it deeply affects the middleware internal structure (e.g., meta-level description generation and runtime maintenance).

More recent proposals [11] overcome the design and implementation burden of reflective approaches via the SoC model [12][13]. In short, the main SoC model characteristics are:

- the SoC architecture provides specifications in terms of service syntax, behavior, and

dependencies on other services via service interface descriptions;
- components implement service interfaces and may leverage implementation-specific dependencies on other services;
- service-oriented interaction pattern is used to resolve runtime service dependencies;
- compositions are described in terms of service specifications and the service-oriented interaction pattern provides the basis for dynamic component replacement.

The SoC model promotes a different approach to dynamically change the structure and behavior of software systems. Primarily it relies on service registry and service tracking to enable dynamic service discovery and replacement, as detailed below.

The SoC model exploits service interfaces to expose the internal behavior and state of components, by relying on component-oriented lifecycle/dependency management features in order to monitor, plan, and enforce component reconfiguration. For instance, [14] proposes a SoC approach to create a middleware with AC facilities, by extending the iPOJO platform with self-management capabilities to support AC service provisioning. However, it only focuses on the support of application-level reconfiguration; no mechanisms are available to enable middleware self-reconfiguration, which is crucial in AC. [15] describes an AC framework built on top of the OSGi component model: its implementation forces to modify existing OSGi implementations and offers only a limited set of reconfiguration capabilities. [9] proposes an Adaptive Server Framework (ASF) to create adaptive applications for UC scenarios, by borrowing some solution guidelines from the AC research area. It focuses on introducing AC capabilities in the functional layer without modifying the application business logic. However, it cannot enforce any reconfiguration of the non-functional layer. In addition, it does not propose any layer for component dependency reconfiguration and does not adopt a flexible control loop, configurable with high-level rules. In short, to the best of our knowledge, there are no clear guidelines (and no concretely designed and implemented frameworks) to enable system self-reconfiguration, especially when self-adaptation of non-functional attributes is crucial.

3. Reconfiguration Requirements

To deal with the heterogeneity and dynamic nature of UC scenarios and to enable proper performance in differentiated execution contexts, UC middleware should be able to provide support facilities that adapt themselves to current context and offer dynamically differentiated QoS levels. In particular, our original middleware aims at tackling system reconfiguration in a twofold way. On the one hand, we propose a novel and fully reconfigurable AC-inspired middleware to serve as the basis for dynamically adaptable UC applications. On the other hand, the goal of our reconfiguration support is to apply to already existing and non-reconfigurable middleware/applications as well.

We claim that, in UC scenarios, system reconfiguration and QoS adaptation should involve reconfiguration as a key-feature of both functional and non-functional layers [16]. Therefore, we have designed our UC middleware according to a two-layer architecture: the application logic layer and the non-functional support one. The former includes the high-level features that help in modeling and managing business logic, independently from the specific application domain. The latter includes components to model non-functional features: for instance, a non-functional middleware component may provide coordination facilities for asynchronous message-driven components, whereas another one may offer persistence capabilities.

By going on with the cooking analogy, strictly following a good recipe and combining the right ingredients off the shelf may help you cooking a good meal. However, sometimes you need to "invent" due to dynamic "context elements": you may run out of a specific ingredient and need to replace it, or a guest may unexpectedly require you avoiding a given spice because of allergy. Replacing an ingredient may require adjusting doses and compositions of the other ones, so to re-balance the general taste of the dish. In other words, cooking requires adapting a good initial recipe to the "runtime environment", so as to make the best out of the current conditions, sometimes by replacing ingredients (application logic), sometimes rearranging how to cook them (non-functional logic).

The integration of off-the-shelf components into articulated and value-added "recipes" calls for coordination and support middleware able to continuously reconfigure compositions to runtime conditions. That typically involves:

- automatic composition - middleware should dynamically discover components that offer requested services and react to service disappearing by automatically repairing broken dependencies;
- lifecycle management - middleware should take care of enabling/disabling components. For instance, when some dependencies of a component could not be resolved, the component should be automatically deactivated;
- component reconfiguration - middleware should be able to tune components in terms of both their inner state and their dependencies;
- automatic synchronization during reconfiguration - the services offered by a component should not be accessed during its reconfiguration and, at the same time, no service requests should be lost;
- capacity of introspection - at any time, the middleware should be able to inspect the state of AC applications. In particular, it should dynamically determine the application structure/organization, e.g., which components are implementing it, which dependencies are broken, and which services a component offers and requires.

The solution proposed in this paper focuses on the need to introduce the AC vision without the need of necessarily redesigning or re-factoring the supporting middleware. This allows introducing new aspects in already existing functional and non-functional facilities. Our aim is to propose a general pattern to be used, if and when needed, to enhance any existing UC middleware, even when not designed from scratch for dealing with AC aspects. In addition, our pattern offers a general solution, which can also be applied to more traditional distributed component models.

4. OTS-RTG: Architecture and Integration Pattern

Our primary design guideline is to create a framework on top of iPOJO in order to flexibly implement all the features listed in Section 3. The goal is to realize the AC-inspired vision of transparent runtime reconfiguration of off-the-shelf software components, ideally with no additional burden on application developers. Our framework supports the lifecycle of the components by including a monitor layer, a control layer, and a reconfiguration layer, as depicted in **Fig. 1**.

The monitor layer implements monitoring facilities designed to be transparently attached to any system component without modifying its business logic. The monitor layer keeps track of different system/components aspects via sensor elements. Sensors are pluggable, can be added to the system without the need of restarting applications, and can be used by the control layer to measure a variety of system parameters. The control layer manages high-level control rules

to recognize states and, based on that, to trigger system reconfiguration. The control layer uses the reconfiguration layer to act on the AC system, by exploiting the capabilities of our SoC-based framework to reconfigure component dependencies/properties and to control component lifecycle. The reconfiguration layer offers high-level APIs, called effectors in AC systems, for acting on application reconfiguration, for instance to create/destroy component instances, to change high-level requirements on component dependencies, and to reconfigure the properties of a given component instance.

The implementation of our middleware components is based on iPOJO, thus facilitating the fact that the middleware components themselves are fully reconfigurable and replaceable without restarting the supported applications. For instance, that permits to replace one implementation of the active control layer with a new one with no need to suspend service provisioning.

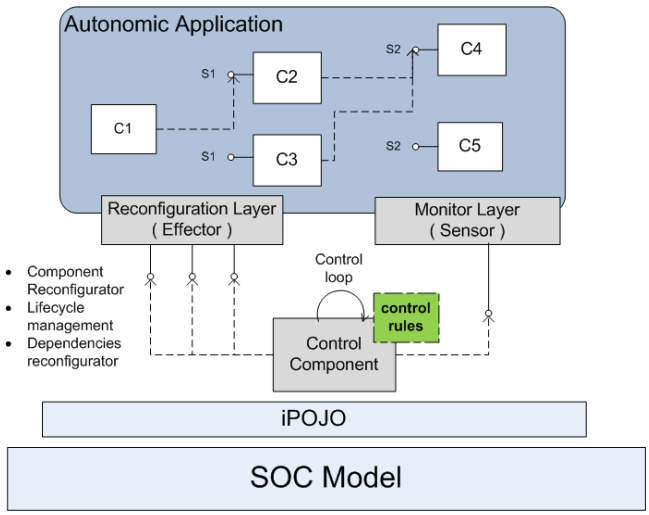

Fig. 1. The AC-inspired architecture of OTS-RTG.

OTS-RTG Integration Pattern

From the point of view of the integration pattern, the primary guideline of our original proposal is the creation of an AC integration framework for clients and service providers that do not support reconfiguration by themselves. The aim is to introduce the AC vision without redesigning existing middleware and/or application components. **Fig. 2** illustrates the proposed pattern: the integration framework acts as a bridge between clients and service providers. The façade provides the same service interface as the one expected by clients; its role is to completely hide the client from the interposition of our integration solution. In addition, it is up to the integration pattern implementation to automatically and transparently inject the façade into the client itself. The wrapper is a general-purpose component that can bind any service provider and acts as an invocation mediator for all the methods offered by the provider. Each client façade depends on one or more wrappers. Again, it is up to our integration pattern implementation to automatically create, make available, and inject wrappers for each service provider, transparently from the client/provider perspective.

Since façades and wrappers are fully-fledged OTS-RTG components, they can benefit from full reconfiguration facilities in their turn. For instance, dependency between a façade and a wrapper may be reconfigured at runtime, and façades/wrappers can be decorated with sensors to monitor their state. Or, just to mention another notable example, our OTS-RTG middleware

may be configured to include a simple load balancer that splits traffic among several instances of providers of the same service. The load balancer can in its turn be dynamically added and removed upon need, for instance when only one service provider is available at runtime due to the failure/unavailability of the others.

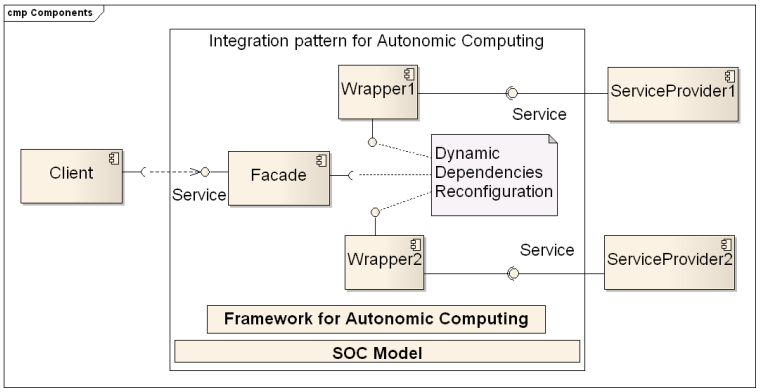

Fig. 2. Our AC integration pattern.

The integration pattern is inherently designed to flexibly provide reconfiguration/AC features to any kind of system. First experiences with OTS-RTG have considered component-oriented systems, in particular Enterprise Java Beans-based architectures. The results obtained clearly showed that our model can provide those architectures with the needed reconfiguration features in a seamless and effective way (the interested reader could refer to [17] for additional and more extensive details, including several quantitative experimental results about performance evaluation). More recent experiences with OTS-RTG have addressed even more challenging and articulated deployment scenarios, of relevant industrial interest, such as ESB architectures.

ESBs are software architectures that serve as foundational layers for the integration and orchestration of complex ecosystems of heterogeneous components via an event-driven and standard-based messaging engine [18]. ESBs typically provide support for:

- service invocation, i.e., synchronous/asynchronous invocation of services and software components via heterogeneous protocols;
- message routing, i.e., addressing, static/deterministic routing, content/policy/rule-based routing of messages;
- messaging and mediation, i.e., message/protocol processing, adaptation, transformation, and enhancement;
- service orchestration, i.e., coordination of multiple service components/ implementations exposed as a single aggregate service;
- complex event processing, i.e., interpretation, correlation, and pattern matching;
- security and quality of service, i.e., support for different security mechanisms/solutions (e.g., message encryption and signing), reliable delivery, and transaction management;
- management, i.e., monitoring, auditing, logging, metering, and administration.

Notwithstanding recent standardization efforts, ESB platforms are extremely differentiated, with different implementations that tend to provide many proprietary heterogeneous features and service levels. In addition, AC reconfiguration is typically supported to a very little extent, if any. More recent implementations (e.g., Servicemix 4.X [19]) rely on a modular, OSGI-based architecture that allows to dynamically plug novel components into the target deployment environment, such as message transformers and protocol adapters. These plug-in

features, however, usually require explicit human intervention. Moreover, any middleware/application component (such as novel message routes) that plans to use plug-in features, again needs to be manually reshaped (at compile time) and redeployed. Hence, none of the available implementations offers the reconfiguration features that we consider crucial for AC scenarios, such as monitoring and autonomous reconfiguration of system components.

As a general consideration, our OTS-RTG integration pattern allows to seamlessly introduce AC self-reconfiguration and self-adaptation capabilities in core ESB functionalities, independently from the ESB implementation. In particular, in the following, we specifically chose to focus on routing features because they are at the heart of every ESB-based integration scenario. Our aim is to show how it is possible to dynamically change routing configurations depending on runtime context, without redesigning existing middleware/application components.

Fig. 3 shows how our integration pattern applies to a typical ESB architecture to provide dynamic AC reconfiguration of routes. The façade acts as a mediator between core ESB routing functionality and the supported components that implement the routing business logic; the latter are automatically wrapped for executing on our AC framework.

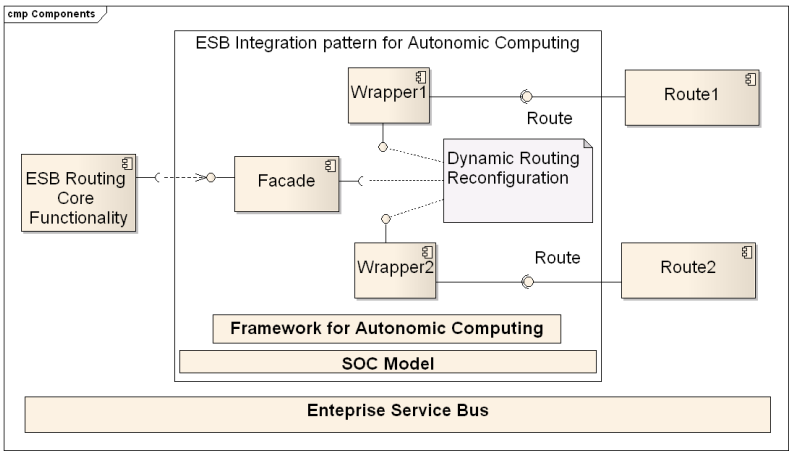

Fig. 3. Our AC integration pattern in the specific case of application to an industrial ESB-based deployment scenario.

5. Implementation Insights

The integration pattern described above has been implemented in our OTS-RTG prototype by exploiting standard Java technologies. In the following, we detail the main implementation aspects of the integration pattern, from its low level facilities, up to high-level components and APIs.

AC Support Framework

First, we have worked on the implementation of the OTS-RTG support framework, whose architecture is depicted in **Fig. 4**, on top of which our integration components execute. Our implementation is based on OSGi, a well-known and largely adopted Java framework that provides the typical features of a SoC component model. On top of OSGi, we have exploited the iPOJO framework for our implementation of the AC MAPE-K control loop.

iPOJO is a container-based framework that handles the lifecycle of simple Plain Old Java

Objects (POJOs) by supporting some management facilities like dynamic dependency handling, component reconfiguration, component factory, and introspection. Moreover, the iPOJO container is easily extensible and has the specific feature of allowing pluggable handlers, typically for the management of non-functional aspects. Even though the iPOJO container is a technically sound basis to start with and offers an interesting set of reconfiguration capabilities, we experienced that it is insufficient to support our envisioned AC reconfiguration scenarios and, for this reason, we decided to significantly extend and patch it, as detailed in the following.

Fig. 4 details our iPOJO components and extensions that realize the monitor, control, and reconfiguration layer. The core of our proposal is the implementation of a handler capable of taking care of all monitoring/control aspects; the handler should operate in synergy with a container that supports instance component execution. In our prototype, the handler is called MonitorHandler. By exploiting Java annotations, MonitorHandler can be attached to any iPOJO component, as shown in **Listing 1**, and can be configured by selecting the proper control rules, control frequency, and the list of POJO methods to be monitored in relation to latency time and requests per second.

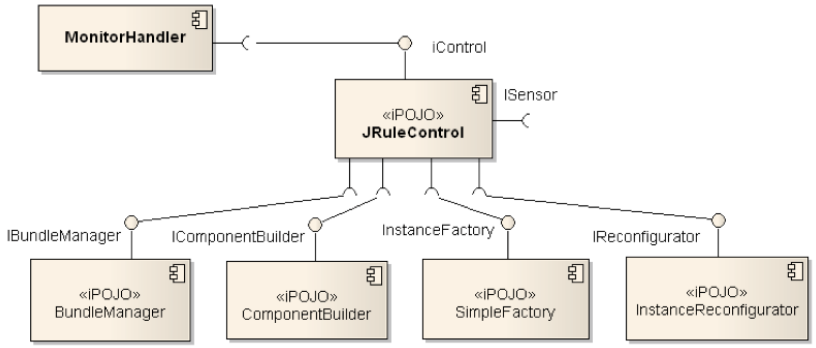

Fig. 4. The modular architecture of the implementation of our iPOJO-based AC framework.

By means of high-level control rules, developers can easily exploit the facilities of the reconfiguration layer to act on the autonomic system by reconfiguring it (in proper and given states). The interested reader could refer to [17] for more details on XML control rules.

```
@Component
@Monitor(ruleXMLFile="reconfigurator.xml", frequency = 10)
public class AutonomicObject {
        @MonitorMethod
      public void doSomething(String parameter) {     }          }
```

Listing 1. Annotations for MonitorHandler configuration.

Our MonitorHandler, as depicted in **Fig. 4**, has a dynamic dependency on the control layer for executing its XML control rules, i.e., a dependency on a support facility that implements the IControl interface. For instance, a control service could be a POJO managed by the iPOJO framework and able to execute XML control rules. The mechanism is highly general and could easily adapt to other binding needs in many relevant reconfiguration cases. In its turn, the control component has a pluggable dynamic dependency on components that offer the service described by the ISensor interface. The ISensor interface defines the general behavior of a component that can offer monitor capabilities to the AC system. In addition, the control component has dynamic dependency on the reconfiguration layer to execute the

reconfiguration action required. The reconfiguration layer consists of four main components that offer different reconfiguration capabilities, as depicted in **Fig. 4**.

InstanceReconfigurator

InstanceReconfigurator is the infrastructure component able to reconfigure any service running in the AC environment, offered either by iPOJO components or by other components that have registered their services through the OSGi APIs. It supports the reconfiguration of both components (e.g., modification of property values of a component field) and their dependencies. To achieve this goal, it acts as a mediator between already available iPOJO/OSGi reconfiguration features and our newly created reconfiguration-oriented components, as described in the following.

InstanceFactory

In order to add reconfiguration facilities to any object instance, the iPOJO framework wraps them with a "container", namely a ComponentInstance iPOJO, which enriches the original object with reconfiguration capabilities. The act of wrapping an object instance into a ComponentInstance container is performed by Factory objects. Therefore, factories play a central role in adding reconfiguration features to any POJO because they basically create the connection between a simple POJO and its reconfiguration features.

However, in the current iPOJO implementation, once the connection gets created, factories are no longer able to retrieve a ComponentInstance from the original object instance. This ends up in an extremely limiting issue since our AC framework, to realize the highly flexible and dynamic reconfiguration goals needed in UC scenarios, has the crucial necessity of explicitly acting on ComponentInstances at runtime. This is necessary to accomplish the required reconfiguration tasks (e.g., to reconfigure dependencies) at provisioning time in a very general way. This is one of the main reasons why we have decided to extend the current iPOJO implementation and to give the possibility to our Factory objects to retrieve ComponentInstances from the original object instance names. Now InstanceFactory is exploited by the InstanceReconfigurator every time an object reconfiguration container (ComponentInstance) needs to be modified or changed.

BundleManager

Bundles usually represent software aggregates that encapsulate code from one or more components. iPOJO exploits the underlying OSGi bundle management features to load/unload bundles (hence, adding/deleting available component code). OSGi offers articulated and complex APIs to deal with bundles. Since our scenario requires more basic and high-level features (primarily, load/unload a bundle), we decided to "wrap" those functionalities in a series of coarse-grained features, to ease software development and maintainability. In particular, our BundleManager offers simplified features to load, unload, start, and stop bundles on top of OSGi bundle-related features.

ComponentBuilder

ComponentBuilder coordinates with InstanceFactory to create run-time iPOJO components from simple Java classes, giving the information required by the framework to configure and manage non-functional aspects through the associated XML file, by skipping the bytecode manipulation phase required at compile time.

In addition, in order to manage reconfigurable component creation, iPOJO basically offers

two distinct ways. On the one hand, the more traditional way exploits compile-time bytecode instrumentation. A developer is expected to annotate her components with suitable iPOJO annotations (e.g., to declare reconfigurable dependencies) and, at compile-time, the iPOJO framework translates those annotations into concrete reconfiguration code. This declarative approach based on annotations provides developers with the finest level of control on the specification of reconfigurable components. However, its compile-time approach makes it intrinsically limited in very dynamic scenarios. On the other hand, one can exploit a set of runtime APIs to dynamically "decorate" existing components. Even if more suitable for dynamic scenarios, this approach suffers from limited expressiveness. Runtime APIs only enable the specification and usage of a subset of the reconfiguration features available via annotations. Moreover, and most relevant, those APIs are intrinsically hard to use for application developers: being extremely low-level, they force developers to undergo a series of repetitive and error-prone tasks.

In our opinion, our ComponentBuilder should take the best of both approaches (at the same time dynamic, highly flexible, and easily usable in a declarative way). Therefore, we designed and implemented the ComponentBuilder with a dynamic runtime feature to enable the creation of reconfigurable components: in OTS-RTG, developers are allowed to invoke the ComponentBuilder at runtime and to pass a suitable declarative description (e.g., an XML file) of components, of their dependencies, and of their reconfiguration characteristics. Then, our ComponentBuilder transparently exploits low-level iPOJO APIs to dynamically create the components and to instrument their code by enriching it with reconfiguration features.

Integration Pattern

On top of this AC kernel layer, we have designed and implemented our façade and wrapper, as components of our OTS-RTG integration system.

Façade Implementation

As already stated, the façade must implement the same component interface requested by associated service clients. Our OTS-RTG framework lets developers determine such implementation either at development time, by manually implementing the target interface, or dynamically, by automatically generating façade code at runtime via bytecode creation/manipulation facilities. On the one hand, the first approach allows the developer to keep a detailed and fine-grained level of control on the façade implementation (for instance, by introducing personalized load balancing strategies). On the other hand, the second approach permits to automatically integrate any component with very limited need of intervention by developers. An example of a development-time-defined façade is shown in the **Listing 2** below:

```
@Component
@Provides
@Monitor(ruleXMLFile="/Users/UserName/Desktop/reconfigurator.xml")

public class PrinterFacade implements IPrinter {
    @Requires(filter="(wrapper.type=IPrinter)",policy="dynamic-priority", nullable=false)
    private IEJBWrapper printer;
        @MonitorMethod
        public void print(String s) {
                try { printer.invoke("print", new Object[]{s});
            } catch(Exception e) {...}      }
```

Listing 2. Example of façade definition at development time.

The façade for a printer service must implement the Printer interface, hence concretely implementing its methods (in our case, the print method). Invocation of the method on the actual component (i.e., the real EJB component) occurs via a generic EJB wrapper (EJBWrapper); EJBWrapper takes care of concretely looking up the printer service EJB and of invoking the method. The @Monitor and @MonitorMethod annotations declare how to monitor such an object activity to determine when it needs reconfiguration. In addition, we have also implemented a load-balancing façade that can split the invocation load among different implementations (e.g., different EJBs implementing the same printing service).

Moreover, we have decided to enhance the ASM framework [20] to support the dynamic generation of the façade code. ASM offers APIs for inspecting class bytecode through the visitor pattern and for creating new visitors able to manipulate or generate new code starting from the visited class or interface. In particular, we have implemented a visitor that inspects the interface definition and automatically crafts the façade code: it creates the stub methods, which exploit the dynamic dependency of the façade on the wrapper in order to invoke the methods of server components.

Wrapper Implementation

Finally, we have implemented a wrapper that offers a dynamic invocation interface and, thus, a very general way of invoking methods for all components. **Listing 3** shows an example of possible implementation of the generic invocation method for an EJB wrapper.

Our wrapper can be reconfigured at runtime to bind to another service provider and, thanks to its dynamic invocation interface, can be used to invoke its methods. The rebinding phase requires performing component lookup. For EJB components, this usually means querying the Java Naming and Directory Interface (JNDI) and loading the corresponding component class definition. This part may generate limited but non-negligible implementation issues to application developers, since the lookup happens on the OSGi classloader, which unfortunately has no visibility of classes outside its bundle (e.g., EJB interfaces and JNDI lookup helper classes). For this reason, in the current OTS-RTG implementation, we have decided to embed JNDI lookup helper classes into the wrapper bundle and to temporarily replace the OSGi classloader with the standard JVM one (during the lookup phase).

```
@Component(propagation=true, immediate=true)
@Provides
public class IPojoEJBWrapper implements IEJBWrapper{
public Object invoke(String method, Object[] args) throws Exception {
        try{ Class parameterTypes [] = new Class [args.length];
                for (int i = 0; i < args.length; i++) {
parameterTypes[i] = args[i].getClass(); }
                Method m = ejb.getClass().getMethod(method, parameterTypes);
                return m.invoke(ejb, args);      }
        catch(InvocationTargetException e) {...}
        }  }
```

Listing 3. Example of EJB wrapper.

ESB Integration Pattern

The viability of the proposed integration pattern to a wide variety of deployment environments is clearly demonstrated by our experience with OTS-RTG for ESB-based service provisioning scenarios. On top of our AC layer, we have designed and implemented the components of our OTS-RTG for AC reconfiguration features in ESB scenarios. Our façade interacts with the core ESB routing functionality. In our practical deployment experience (described in the

following) we chose the widely diffused Apache Camel [21] integration framework for routing capabilities; Apache Camel is embedded in most industry-scale ESBs nowadays and has proven to be an effective solution to typical routing issues. In particular, we realized a façade that interacts with Apache Camel to support runtime deployment and reconfiguration of integration routes. Integration routes are pieces of business logic that enable the integration of heterogeneous systems, e.g., a Web Services system and a messaging one, for creating new business flows. Apache Camel supports the creation of such integration routes by minimizing the effort required to application developers.

Façade Implementation

Our façade acts as a container of an instance of Apache Camel and is designed as a fully-fledged iPOJO component. As shown in **Listing 4**, the Camel instance gets created when the *start()* method is invoked, just after the instantiation of our façade component. In addition, the façade has a dynamic dependency, managed according to a plug-in strategy, towards the components that wrap a Camel integration route. It can dynamically add routes to the integration framework when they become available, as well as to dispose routes when they are no more available. **Listing 4** provides an excerpt of some crucial parts of our façade implementation, by highlighting the logic that gets invoked when dynamically adding or deleting integration routes to the ESB.

```
@Component
public class RouteManager{
        @Requires(policy="dynamic-priority", nullable=true, id="wrapper", optional=true)
        private List<RoutesBuilder> camel_route = new ArrayList<RoutesBuilder>();
        private DefaultCamelContext camelContext;
        @Requires(from="it.epocaricerca.servicemixipojo.routemanager.RouteManager")
        private Factory factory;
                private int num_routes = 0;
        private HashMap<RoutesBuilder, String[]> route_cache = new HashMap<RoutesBuilder, String[]>();
        @Validate
        public void start() throws Exception {
                CamelContextFactory ccfb = new CamelContextFactory();
                …
                this.camelContext =  ccfb.createContext();
                this.camelContext.start();      }
@Invalidate
        public void stop() throws Exception { this.camelContext.stop(); }
        @Bind(id="wrapper")
        private synchronized void bind(RoutesBuilder route) throws Exception {
                this.camelContext.addRoutes(route);
                int actual_size = this.camelContext.getRoutes().size();
                …          }
        @Unbind(id="wrapper")
        private synchronized void unbind(RoutesBuilder route)  throws Exception {
                String routes_id[] =  this.route_cache.remove(route);
                …
                this.num_routes -= routes_id.length;     }          }
```

Listing 4. A code excerpt of the implementation of our ESB routing façade.

Wrapper Implementation

As already shown, our wrappers can be easily implemented as bean components decorated with iPOJO annotations (see **Listing 5** below). Dynamic route implementation in OTS-RTG poses very limited (almost negligible) burden on middleware/application developers: the route

is a traditional Apache Camel one, with no further requirements, except for the declaration of a few iPOJO-specific annotations. The façade declares a dependency towards components that implement the *RoutesBuilder* interface, i.e., the interface that classes have to implement in order to be capable to configure Apache Camel routes. As a consequence, we simply require decorating the target class with iPOJO annotations for managing the wrapper via our AC framework. **Listing 5** reports a code excerpt that shows how easy it is to implement a generic wrapper that configures an Apache Camel route.

```
@Component(immediate=true)
@Provides(specifications=RoutesBuilder.class)
public class FetchQueue extends RouteBuilder {
        @Override
        public void configure() throws Exception {
                if(getContext().getComponent("activemq") == null)
                getContext().addComponent("activemq", activeMQComponent("vm://localhost")); }        }
```

Listing 5. Code excerpt from our implementation of the Apache Camel route wrapper.

The wrapper above creates a Camel route that fetches messages from a queue, passes them to a POJO, and sends them to an in-memory queue for further computation. The iPOJO component described above extends the *RouteBuilder* class and offers the *RoutesBuilder* interface as its exposed service. The components that offer the *RoutesBuilder* service are plugged into the façade and deployed in the considered Camel instance as integration routes. Being fully-fledged wrappers, routes can be dynamically created and disposed, thanks to this easy integration with our AC framework. This allows for extremely powerful, yet easy to implement, reconfigurable scenarios. For instance, it is particularly easy to design and implement a reconfigurable integration flow, i.e., a more complex integration solution resulting from the composition of different routes. The flow can also be dynamically reconfigured by replacing a route; this is obtained simply by deleting a wrapper instance and by creating another instance of it with different business logic.

6. Experimental Results

To quantitatively assess and validate the performance of our highly flexible OTS-RTG, we have deployed and tested it on servers with Core Duo 2CPUs operating at 2.2GHz and 2GB of RAM. In the tests we used MacOSX v10.6 with Java Virtual Machine v1.5; the extended application server is JonAS v5.1.1, running on Apache Felix v2.0.3 OSGi implementation; the extended ESB architecture is Apache Servicemix v4.2.

OTS-RTG Prototype: Performance of AC Support

As a first experimental evaluation, we have estimated the overhead of a reconfiguration action in a simple application scenario, where the provided service comes from the composition of two components (a client and a server) that are supported by our OTS-RTG middleware. Our performance results (extensively reported in [17]) show that, when reconfiguration takes place, invocation latency is comparable to usual component instantiation and invocation

Secondly, we have evaluated the overhead given by the presence of our middleware if compared with pure OSGi. To this purpose we have deployed a complex application with a variable number of components that require services offered by other components and, of course, with component reconfigurations during test execution. OTS-RTG scales well while growing the number of reconfigured components, as shown by the extensive experimental

results reported in [17].

OTS-RTG Prototype in ESB scenario: Performance of Integration Pattern

In order to assess and validate the implementation of our integration pattern, we have chosen to test it in a real usage scenario of practical industrial interest. In particular, we have deployed our prototyped solution in a case study where the ESB middleware, enhanced with the AC capabilities deriving from our integration pattern, can dynamically reconfigure an integration route. The goal is to adapt to dynamically changing traffic load levels, by trying to maintain the same QoS level, independently of traffic. Our integration business logic fetches messages from a source queue and stores them in two different destination queues, after having performed some management operations on the handled messages, as depicted in **Fig. 5**.

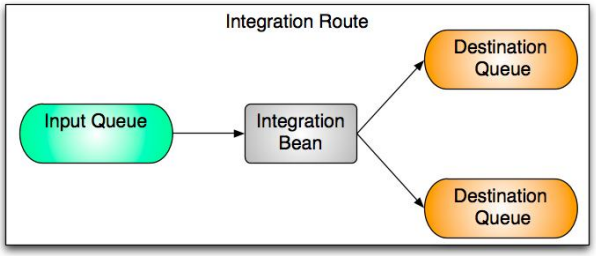

Fig. 5. Message flow used during the evaluation of the performance of our OTS-RTG integration pattern when applied to an ESB-based architecture.

The initial integration route has a non-functional requirement that forces to process messages transactionally. When the system is overloaded, however, the time that a message spends in the input queue grows, with negative effects on the perceived quality level. Therefore, in this simple example, the primary idea is to reconfigure the integration route, in response to dynamic load growth, to avoid the overhead due to the transactional requirement. Thus, the removal of the non-functional requirement (transactionality) should reduce the average enqueuing time. Of course, this is only a possible example, used for the sake of simplicity and briefness, which makes sense in real scenarios only if the targeted application can tradeoff its transactional requirements for better response times without infringing application-specific consistency requirements.

To validate our middleware implementation, we have stressed it by generating a high traffic load of 1000 messages per second (with small payload) and by analyzing the average enqueuing time for every 1s interval. The rationale behind this performance experiment is to determine whether, on an average execution case, route reconfiguration imposes a relevant overhead for the AC middleware support. First, we collected some initial results to assess the basic route performance in static cases (i.e., with no reconfiguration). The corresponding experimental results are reported in **Fig. 6**. In particular, the figure compares the average message enqueuing time in a static configuration, when using either a transactional route or a non-transactional one. As expected, transactional queuing generates a higher overhead on the middleware; messages remain queued for longer time intervals, even though the difference between transactional and non-transactional behavior remains relatively small (a couple of ms in the worst case). The initial overhead associated with the transactional route (see the spike at about 2s in **Fig. 6**) depends on the initial setup of resources, needed by the transaction manager to deal with atomic operations. On the contrary, the initial setup overhead associated with the non-transactional route is negligible.

As a second evaluation step, we introduced our middleware reconfiguration features and

forced the system to self-adapt by passing from a transactional queue to a non-transactional one when the average enqueuing time goes over a given threshold. Specifically, in our tests we configured our middleware with one of two possible reconfiguration options. In the first one, when enqueuing time exceeds the threshold, the middleware disposes the transactional route and creates a non-transactional one. In the second one, the self-reconfiguration consists in keeping the transactional route working and pairing it with the newly created non-transactional one; the rationale is to have two routes working concurrently in order to reduce the overall average enqueuing time.

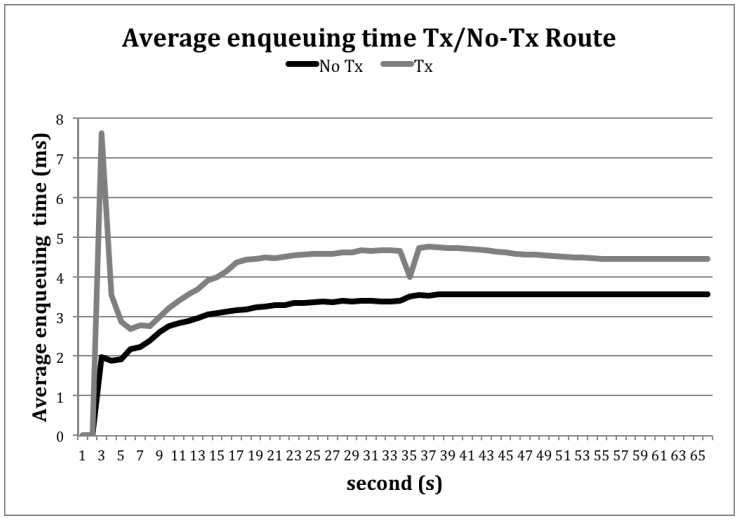

Fig. 6. Comparison of average enqueuing times for the transactional and non-transactional cases, under high traffic load (1000 messages per second).

The collected experimental results are reported in **Fig. 7**. The figure compares the average enqueuing time of the messages with the two types of reconfiguration (reconfiguration triggered when average enqueuing time goes over the threshold of 3 ms). As expected, the first case, which has to perform the management operations of deleting an existing route and creating a new one, is characterized by an initial limited reconfiguration overhead. In fact, for a short time interval, the ESB switches off the first queue and then switches on the new one. The grey line shows how the average enqueuing time increases during these reconfiguration operations (the spike at about 4s): this is mainly due to the fact that, during reconfiguration, no messages are fetched from the input queue. Anyway, the message enqueuing time remains extremely well controlled (around 50ms) and is very confined in time. In fact, after reconfiguration, the system has exhibited to fetch messages faster: the reconfigured system no longer needs to deal with transactional behavior, with the performance advantages shown by the reported exponential decrease of the average enqueuing time.

The second reconfiguration type, described by the black line, shows a more efficient reconfiguration strategy, where the transactional route works together with the non-transactional one. Our middleware creates a non-transactional route that helps the overloaded one by concurrently fetching messages from the input queue. That has demonstrated to reduce the average enqueuing time (around 1ms) if compared with the static configuration, by imposing very limited management overhead on the overall system.

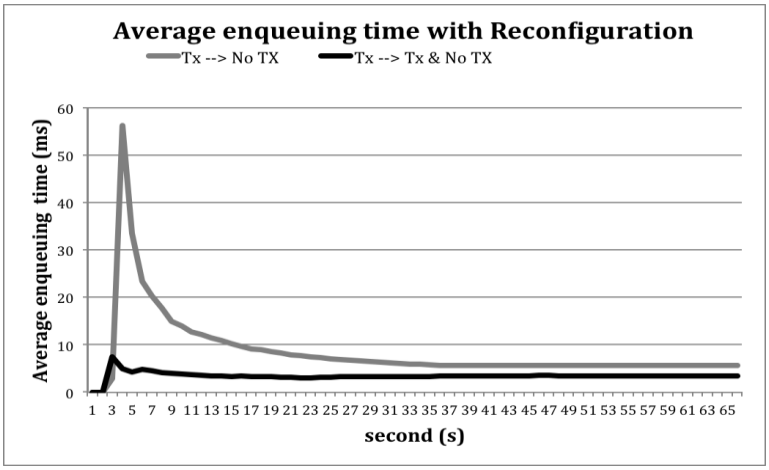

Fig. 7. Automatic reconfiguration of employed route(s) depending on runtime traffic and experienced QoS: case 1 with reconfiguration from transactional to non-transactional queue; case 2 with reconfiguration from one transactional queue to one transactional plus one non-transactional queue.

Let us finally notice that, as a second step of quantitative validation, we have tested our integration pattern in a real J2EE usage scenario, where an application needs to reconfigure itself to offer the suitable QoS levels in all executing conditions. The related results are reported in the experimental result section of [17].

7. Conclusions

This paper has presented a novel iPOJO-based middleware for self-reconfiguration/ optimization of both functional and non-functional features. The proposed approach is oriented towards off-the-shelf component reconfiguration, with the primary goal of easily enabling the monitoring of runtime conditions and consequent adaptation operations based on dynamic dependency management. We also proposed an original integration pattern to extend traditional non-reconfigurable systems in an easy, flexible, and scalable way: our implemented integration pattern can supply AC features also in middleware solutions that were not designed to cope with self-reconfiguration/auto-optimization. We extensively benchmarked our platform and collected several experimental results to verify the practical feasibility of the proposal: in all the targeted deployment scenarios we experienced good scalability and very limited latency, thus quantitatively confirming the suitability of the proposed approach.

The promising results already achieved are encouraging further research activities. In particular, we are currently working on a smarter control component to include in our middleware: that control component will be able not only to trigger reconfigurations based on fixed rules, but also to infer proper behavior from previous session history, e.g., previous component executions, environmental variations, and reconfiguration operations occurred in the past.

References

[1] M. Weiser, “Hot Topics: Ubiquitous Computing,” *IEEE Computer*, vol. 16, no. 10, Oct. 1993. Article (CrossRef Link)

[2] C. Escoffier, R.S. Hall, P. Lalanda, “iPOJO: an Extensible Service-Oriented Component

Framework," in *Proc. of Int. Conf. Services Computing* (*SCC*), 2007. Article (CrossRef Link)
[3] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, T. Sivaharan, "A Generic Component Model for Building Systems Software," *ACM Transactions on Computer Systems*, vol. 26, no. 1, Jan. 2008. Article (CrossRef Link)
[4] N. Huhns, M.P. Singh, "Service-Oriented Computing: Key Concepts and Principles," *IEEE Internet Computing*, vol. 9, no. 1, Jan. 2005. Article (CrossRef Link)
[5] J.O. Kephart, D.M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 26, no. 1, Jan. 2003. Article (CrossRef Link)
[6] IBM, "Autonomic Computing: IBM's Perspective on the State of Information Technology," http://www-1.ibm.com/industries/government/doc/content/resource/thought/278606109.html
[7] M.C. Huebscher, J.A. McCann, "A Survey of Autonomic Computing," *ACM Computing Surveys*, vol. 40, no. 2, Aug. 2008. Article (CrossRef Link)
[8] G. Huang, T. Liu, H. Mei, Z. Zheng, Z. Liu, G. Fn, "Towards Autonomic Computing Middleware via Reflection," *Technical Report*, University of Beijing, China. Article (CrossRef Link)
[9] F. Kon, F. Costa, G. Blair, R.H. Campbell, "The Case for Reflective Middleware," *Communications of the ACM*, vol. 45, no. 6, June 2002. Article (CrossRef Link)
[10] W. Ding, G. Marchionini, "A Study on Video Browsing Strategies," *Technical Report*, Univ. Maryland at College Park, USA, 1997.
[11] S. Zachariadis, C. Mascolo, "The SATIN Component System-A Metamodel for Engineering Adaptable Mobile Systems," *IEEE T. Software Engineering*, vol. 32, no. 11, Nov. 2006. Article (CrossRef Link)
[12] Y. Liu, I. Gorton, "Implementing Adaptive Performance Management in Server Applications," in *Proc. of Int. Workshop Software Engineering for Adaptive and Self-Managing Systems* (*SEAMS*), 2007. Article (CrossRef Link)
[13] H. Cervantes, R.S. Hall, "Autonomous Adaptation to Dynamic Availability through a Service-Oriented Component Model," in *Proc. of Int. Conf. Software Engineering*, May 2004. Article (CrossRef Link)
[14] A. Diaconescu, J. Bourcier, C. Escoffier, "Autonomic iPOJO: Towards Self-Managing Middleware for Ubiquitous Systems," in *Proc. of IEEE Int. Conf. on Wireless Mobile Computing, Networking & Communication*, 2008. Article (CrossRef Link)
[15] J. Ferreira , L.Rodrigues "A-OSGi: a Framework to Support the Construction of Autonomic OSGi-based Applications," *Technical Report RT/33/2009 INESC*, May 2009.
[16] A. Corradi, E. Lodolo, S. Monti, "Dynamic Reconfiguration of Middleware for Ubiquitous Computing," in *Proc. of 3rd Works. Adaptive DependAble Mobile Ubiquitous Systems* (*ADAMUS*), 2009. Article (CrossRef Link)
[17] P. Bellavista, A.Corradi, D. Fontana, S. Monti, "Off-the-shelf Ready To Go Middleware for Self-reconfiguring and Self-optimizing Ubiquitous Computing Applications," in *Proc. of 5th Int. Conf. on Ubiquitous Information Management and Communication* (*ICUIMC*), Korea, Feb. 2011. Article (CrossRef Link)
[18] J. Yin, H. Chen, S. Deng, Z. Wu, C. Pu, "A Dependable ESB Framework for Service Integration," *IEEE Internet Computing*, vol. 13, no. 2, Mar. 2009. Article (CrossRef Link)
[19] The Apache Software Foundation, "Apache Servicemix 4," http://servicemix.apache.org/smx4/
[20] E. Kuleshov, "Using ASM Framework to Implement Common Bytecode Transformation Patterns," in *Proc. of Int. Conf. on Aspect-Oriented Software Development* (*AOSD*), Mar. 2007.
[21] The Apache Software Foundation, "Apache Camel," http://camel.apache.org/

Paolo Bellavista graduated from the University of Bologna, Italy, where he received a Ph.D. in computer science engineering in 2001. He is now an associate professor of computer engineering at the same university. His research activities span from middleware for mobile computing in general to location/context-aware services, from adaptive multimedia and vehicular sensor networks to mobile agent technologies. He serves on the Editorial Boards of IEEE Communications Magazine, IEEE Transactions on Services Computing, Elsevier Pervasive and Mobile Computing Journal, and Springer Journal of Network and Systems Management. He is a senior member of IEEE and ACM.

Antonio Corradi graduated from the University of Bologna and from Cornell University in electrical engineering. He is a full professor of computer engineering at the University of Bologna. His research interests include distributed and parallel systems/solutions, middleware for pervasive and heterogeneous computing, infrastructure support for context-aware multimodal services, network management, and mobile agent platforms. He is a member of IEEE, ACM, and AICA.

Damiano Fontana received his degree in Computer Science Engineering in 2010, from the University of Bologna, Italy. He is currently working as a research fellow at the University on Modena and Reggio Emilia. His areas of interest are mobile and pervasive applications and computing and enterprise, service oriented integration systems.

Stefano Monti received his degree with honors in Computer Science Engineering in 2004, from the University of Bologna, Italy where he also received a PhD with a thesis on Middleware Principles and Design for the integration of Ubiquitous Mobile services. He is currently working as a software architect at Epoca s.r.l. (http://www.epocaricerca.it). His areas of interest are mobile and pervasive applications and computing and enterprise, service oriented integration systems.