Flexible Service Composition Based on Bundle Communication in OSGi

Ramon Alcarria, Tomas Robles, Augusto Morales and Sergio Gonzalez-Miranda
Telematics Department, Technical University of Madrid
Madrid, 28040 - Spain
[e-mail: {ralcarria,trobles,amorales,miranda}@dit.upm.es]
*Corresponding author: Ramon Alcarria

Received September 1, 2011; revised January 16, 2012; accepted January 18, 2012; published January 31, 2012

Abstract

Service provision and consumption platforms are more and more used to enable communication between sensors, actuators and intelligent devices, since they provide mechanisms that make possible the combination of components to create composed services. However, these kinds of platforms have limitations to adapt themselves to new and unknown devices. In this work we analyze the challenge of component or bundle communication by using the Open Services Gateway Initiative (OSGi) technology and we propose three mechanisms with the aim of contributing to flexible component communication: Common Service, Specific Service and WSIF Web Service Invocation. We provide these solutions with some architectural models and validate them through different example services. Finally we compare them regarding performance, flexibility and application complexity.

Keywords: Ubiquitous computing, M2M, OSGi, self-adaptive systems

DOI :10.3837/tiis.2012.01.007

1. Introduction

Machine to machine communications have evolved in recent years to become an indispensable part of our lives and of our digital world. Recent related works combine characteristics of SOAs and Component-based Software Architectures (CBSE) to facilitate communication between sensors, actuators and other elements [1]. These studies contribute to the provision and consumption of composed services, which integrate device oriented information with valuable information for users [2][3].

In M2M environments, devices appear and disappear due to problems of range, battery or maintenance issues. Also, new devices are discovered and must be integrated into M2M solutions. This paper analyzes how OSGi technology can be used to enable flexible composition of services that try to access to unknown bundles. The integration of these services into M2M solutions allows M2M applications not only to access their functionality but also to obtain valuable information from them. In the next section we define our service composition model and the benefits of using the OSGi technology to implement it. Section 3 describes related work in Flexible Service Communication and Section 4 defines three mechanisms and reference architectures, which manage the problem of communication with unknown OSGi bundles. Section 5 shows our qualitative and performance analysis and Section 6 the conclusion of our work.

2. Service Composition Model

We define the logical structure of a service by different levels: service level, component level and implementation level. A composed service is a set of simple, interrelated elements, called atomic services. The process of the creation of new and complex composed services by service interconnection is called *Service Orchestration*. Atomic services are represented in the Abstract level as abstract services, which consist of a definition of service's functionality, and in the Implementation level as concrete services, which consist of a concrete implementation of the service logic behavior, provided by external elements, such as devices or web servers. In execution time, when a suitable functionality appears in the environment (e.g. new devices are found) the abstract service is matched with the concrete service that implements this functionality. This process is called *Service Resolution*.

To facilitate the understanding of these concepts we present the example of a service, called *Sport Tracker*, which aims to access the location information of a user and to represent it on a map along with information about his heartbeat. This service consists of three components which contain the definition of three services: A *Map* provider, a *Location* service and a *Pulse* service. These abstract services provide an interface that must be implemented so that the composed service can be executed. For example, the Location service needs to be resolved in a GPS device or a GPS capability offered by a mobile phone in order to get real information about the user's location. **Fig. 1** shows our service composition model, adapted to this simple example service.

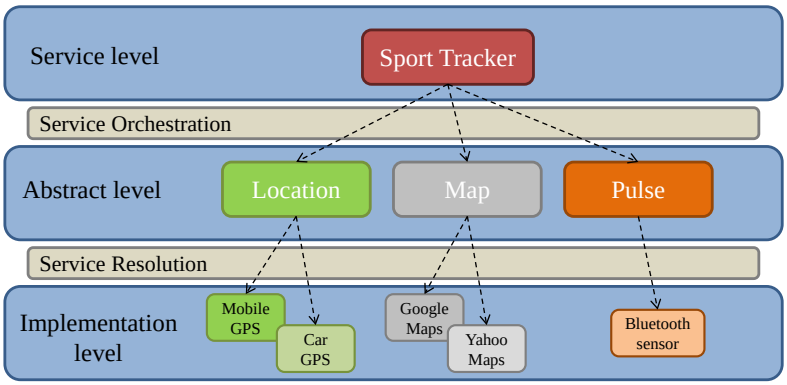

Fig. 1. Service Composition Model

2.1 Resolving Composition Challenges Using OSGi

The OSGi framework is a modular service platform for the Java programming language that implements a complete and dynamic component model. Applications or components (coming in the form of bundles for deployment) can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot. **Fig. 2** shows the communication between a consumer and a provider bundle. Each bundle has a manifest file (Manifest.MF), which contains instructions on how the OSGi platform manages the bundle. A service is a Java interface, which contains a number of methods, implemented by other classes contained in the bundle. Services are exported and imported through the OSGi context, which acts as a repository for publishing and searching services.

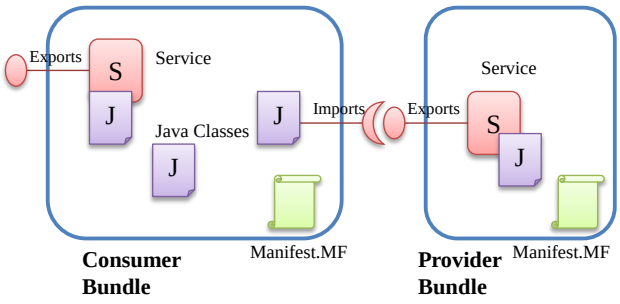

Fig. 2. Bundle Communication in OSGi

Fig. 3 shows how the OSGi model can be applied to access Location information, based on the *Sport Tracker* service. According to the figure, the *Sport Tracker* service is defined in an SDL (Service Description Language) document that is designed by a service developer in, for example, a WS-BPEL orchestration language, which specifies the OSGi services to be accessed, such as in the work of Diaz Redondo et al. [4]. A similar behavior exists between the abstract level of the service composition model and the concept of OSGi service, since OSGi services (which consist of interface definitions) can be used as basic functional units of a composite service. In addition, we also observe an equivalency between the implementation level of the composition model and the concept of OSGi service implementation, since there are two bundles that publish the same service (*Location Service*), but they implement it in different ways, one that accesses the user's mobile GPS and the other that connects to the car's

GPS device. The orchestration and resolution processes manage the interactions between the composite service and OSGi services and between OSGi services and the bundle that implement them respectively. These processes are modeled in OSGi through the reference architectures defined in Section 4.

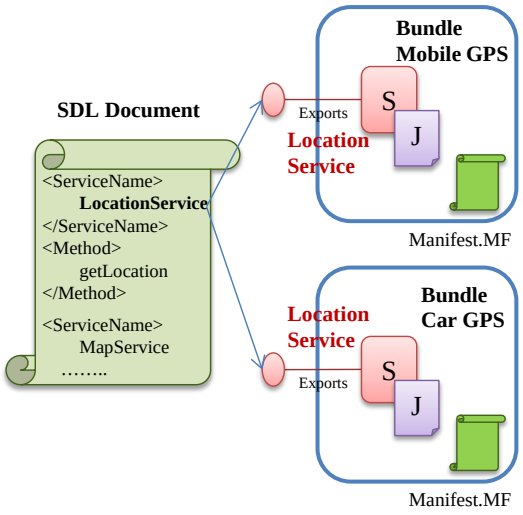

Fig. 3. Accessing location information using OSGi

2.2 Communication with Unknown Bundles

In OSGi, given the impossibility for the bundle manager to be created with knowledge of the provider bundles that are going to exist in the platform, it will be difficult for this module to know the name of the service to consume and the signature of the methods to execute until it has received the SDL document with the information regarding the services to invoke. Therefore, there is a situation where the communication environment will not know a priori the services to be accessed. We have called this problem *communication with unknown bundles* and its solution enables a *flexible service composition.*

3. Related Work in Flexible Service Composition

This section analyses how research projects and scientific contributions resolve the problem of communication with unknown services or components. Contributions to this problem can be classified into two groups: Those who opt for traditional communication systems via messaging (Message Oriented Middleware, MOM) and those who use the advantages of dynamic binding in Web service composition.

Within this first group is the project SOA4All, contributing to achieve a federation of ESBs for large SOA deployments [5]. Thus, PETALS ESB becomes a Distributed Service Bus, in which components, not known a priori, are interconnected by using the Java Message Service (JMS). S-Cube project addresses this problem, but within the field of self-adaptation of service-based systems. This adaptation aims to extend the architecture with new components and addresses the problem of interface adaptation [6], which often appears in environments where components to connect are not known a priori.

The PLASTIC project has developed a platform for rapid and easy development, deployment and execution of adaptable services over Beyond 3rd Generation (B3G) networks. In the frame of this project, a two-layer approach is proposed [2], in which the component layer manages the life cycle of the software components, whereas the service layer manages the description of services and their composition. Our defined model includes an additional level, the implementation level, dealing with real access to resources through the resolution process. Related to component communication, the PLASTIC project is aware that Web Services are not the only way to implement SOA. Message-Oriented Middleware systems, such as the IBM MQ series or CORBA are proposed as an alternative for component communication although they do not propose any particular definition language or model for message interchange. In other work, Autili et al. [3] propose a technique for Service Over The Air provision, for binding new components to the platform. This technique requires Inter Process Communications between midlets, such as file writing and reading, socket communication or database insert and retrieval.

The MOSAICO project [7] also supports an architecture for evolving component-based systems, in which new components can be plugged in at run-time. For doing that an extension layer is provided to make the binding with new run-time components through extension points (joint points), defined by an aspect oriented programming model.

The problem of communication with unknown bundles, whose availability may also change dynamically, can be addresses from the perspective of Web service composition, which facilitates decoupling between orchestration logic and WS definition, using technologies such as WS-BPEL and WSDL. The ASTRO project [8] develops a platform that supports composition and execution of WS, invoked using ActiveBPEL, an execution engine which performs service invocations with SOAP over HTTP. The problem of these approaches is based on the premise that all services used in the compositions are WS, where services may be not related to the web world or are offered by devices which, due to their limited processing resources cannot implement SOAP/WSDL. However, 6LowPAN and the emerging Constrained Application Protocol (COAP) are considered to make real progress for providing the WS paradigm to limited devices. Access to non-web services may require installing additional software components that act as libraries or access drivers to devices. Existing solutions for invoking software components from a BPEL-style orchestration logic are:

- Directly embedding Java code in a BPEL process, using the Java BPEL exec extension (*bpelx:exec*). Used for very simple Java tasks and therefore not valid for our approach.
- Wrapping the code as a SOAP service (e.g. using the Java API for XML Web Services, JAX-WS). Although this solution enables code reusability, SOAP overhead produces very low performance.
- Using Web Service Invocation Framework (WSIF), which uses the native J2EE RMI protocol for communication with Enterprise Java Beans (EJBs). This mechanism provides acceptable performance values and will be considered for the analysis of Section 4 and the comparison with OSGi solutions of Section 5.

Most of the related work that contribute to flexible component communication try to solve the communication with unknown components through techniques based on messaging (JMS in SOA4All, MOM systems in PLASTIC or RMI in some WS-based projects). To solve the problem of communication between unknown bundles in M2M environments we propose two OSGi solutions based on direct bundle communication that offer significantly better performance than using messaging-based techniques.

4. Resolving Bundle Communication Issues

The problem related to communication between bundles is produced at runtime, when the platform, through a bundle manager, wants to access a provider bundle (may also be consumer and the problem would be equivalent) whose OSGi service (interface name and signature of the methods it contains) is described in an SDL document, like the one that defines how to access the location information in **Fig. 3**. Given the impossibility for the bundle manager to be created with knowledge of the provider bundles that are going to exist in the platform, two main problems appear:

- Problem #1: In the service provider's side, the provider bundle must inform the bundle manager of the specific service the provider deploys (method definitions and service names).
- Problem #2: In the service consumer's side, according to OSGi, the bundle manager must import the package containing the service's interface in its manifest file. (*importPackage: ProviderBundle1*). In addition, consumer classes must import the service interface reference and use it (*ProviderService ps = bundleContext.getService();*).

To overcome these problems we define three reference architectures which support two solutions that can be applied in the OSGi platform and another solution brought from the Web Service world: Specific Service, Common Service and WSIF Invocation. These solutions are based in other related studies. However, these studies do not take into account the problem of communication with unknown bundles neither different alternatives are analyzed nor compared in terms of performance.

4.1 Specific Service

This solution solves the problem of communication between the bundle manager and a specific unknown service. The bundle manager uses the Java reflection library to modify his own behavior at runtime. The reflection mechanism in OSGi is often used when it is needed to invoke services described in SDL documents, as in the work of Anke et al. [9], which exposes OSGi services as Web services. Reflection in OSGi is also used to make performance comparisons [10] and reconfigurable middleware [11] (e.g. through the iPOJO solution [12]). The architecture of the Specific Service (SS) solution is presented in **Fig. 4**.

The *Bundle Manager* performs the service orchestration process, as explained in the service composition model in **Fig. 1**. We assume that the bundle manager does not know a priori the orchestration logic, the services to run or the signature of the methods that compose them, but this information is available on the SDL document that is accessible by the *Service Reader*, so that this solves Problem #1. The Service Controller performs service resolution (i.e. the selection of the best bundle implementation of a given OSGi service) by invoking OSGi *BundleContext* method *getServiceReferences(String serviceName, String filter*), specifying a syntactic filter for bundle selection. The SS Architecture uses the reflection mechanism from the *java.lang.reflect* package for solving Problem #2, as there is no need to specify references to specific services in the consumer class, but dealing with a Java *Object* and invoking the method *methodName*, as follows:

```
Object providerService = context.getServiceReference("serviceName");
Class serviceClass = providerService.getClass();
Method serviceMethod = serviceClass.getMethod("methodName");
Object result = serviceMethod.invoke(providerService);
```

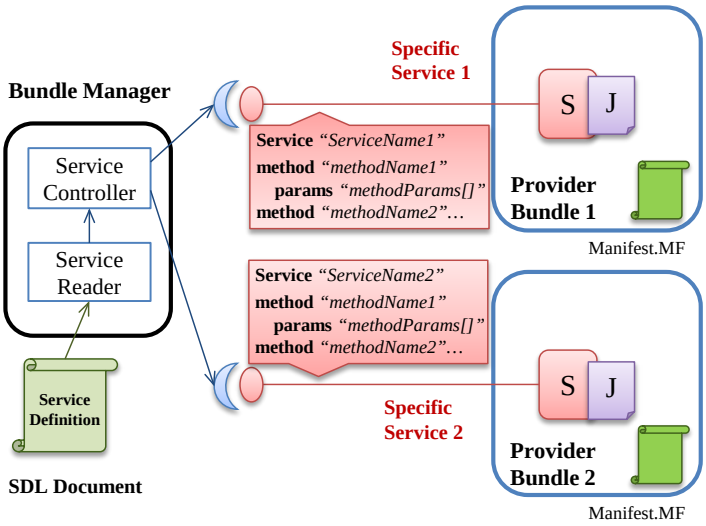

Fig. 4. Specific Service Architecture

4.2 Common Service

All the provider bundles publish the same OSGi service, i.e. the same interface. This standard solution has been utilized as a basic model in related work [4][13][14], where provider bundles are developed specifically for the platform. This solution is often combined with the inversion of control pattern, through which the *Bundle Manager* finds and starts the *Provider Bundles*, which consumes the common service. **Fig. 5** shows the application of the Common Service Solution to the defined SS Architecture.

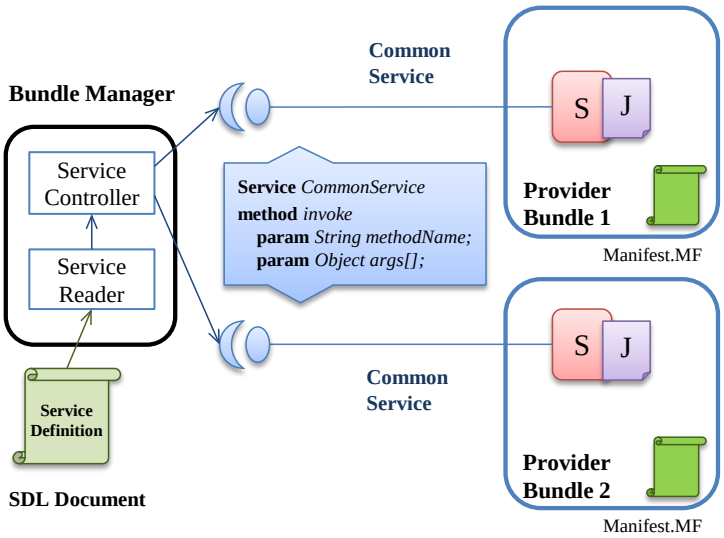

Fig. 5. Common Service Solution

The common interface contains the method *invoke (String methodName, Object args [])* that returns an *Object*, being *methodName* the method name to invoke and *args[]* an array of arguments that supports the method *methodName*. This solution eliminates Problems #1 and

#2 directly, since service definition is shared and the bundle manager has been designed for using the common service.

However, a theoretical analysis shows that the Common Service mechanism is not very flexible, since provider bundles have to publish a common interface, which imposes restrictions to the developers of these bundles. We can solve this problem by using dynamic proxy generation mechanism to translate invocations from the Common Service to the provider bundle's Specific Service. We define a Common Service with Proxy Generation (CSwPG) architecture (shown in **Fig. 6**), which supports the mechanism of dynamic proxy generation, used in R-OSGi [15] and iPOJO [12]. R-OSGi uses byte code manipulation through the ASM library to create a complete proxy implementation of a remote bundle at runtime, and packs it into a JAR file together with the specific service interface. iPOJO also uses bytecode manipulation for handling service dependencies. The CSwPG solution produces more overhead, due to dynamic code generation, as discussed in the Validation section.

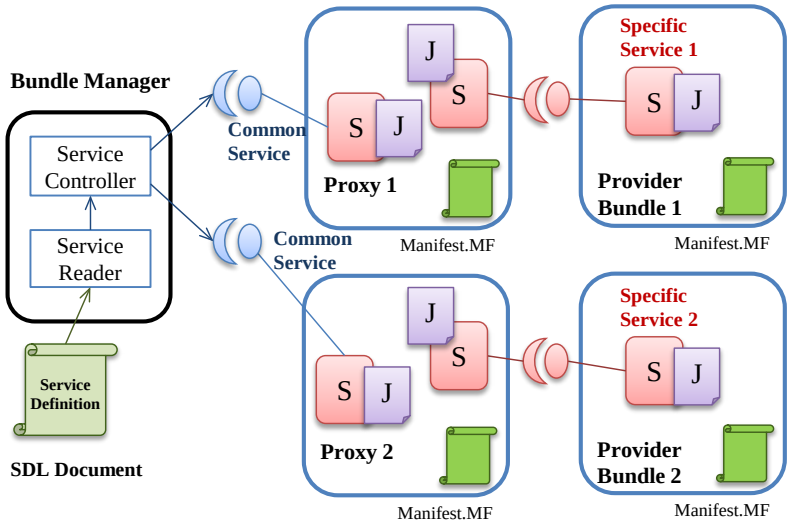

Fig. 6. Common Service with Proxy Generation Architecture

4.3 WSIF Invocation

This proposed solution uses Apache WSIF to execute software components in a flexible way without having to resort to a remote communication (i.e. through SOAP), in which the performance overhead is several orders of magnitude larger than invoking native Java classes or EJBs. As described in Section 3, the Apache WSIF API is the option that offers highest performance in binding between Java classes and WSDL interfaces. Related work shows this mechanism [16], which uses the Java RMI library, and other Web Service Invocation Frameworks, such as Apache Axis and CXF, which support SOAP and JAX-WS invocations respectively [17]. We include a solution based on the WSIF Framework because it offers a good performance to be compared with the previous OSGi solutions.

The proposed architecture in **Fig. 7** supports the process by which a Java class or an EJB can be used as a WS. First of all we must define the signature for each operation and the corresponding input and output messages (1). Then we must define WSIF bindings (2): binding type (specifying the target of the invocation, a Java class, EJB, JCA, etc), type mapping (for all complex types, simple types such as '*int*' are mapped automatically) and operation mapping (where we must specify for each WSDL operation the corresponding

operation in the target resource). The <*partnerLinkType*> declaration in the Service Definition (3) will redirect the service invocation (4) through the WSIF Framework and then through the Provider Component (5).

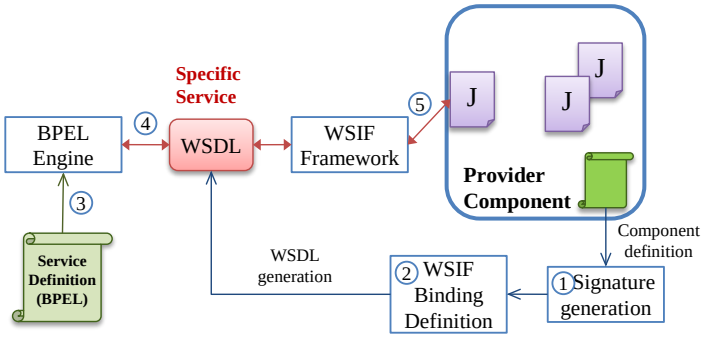

Fig. 7. WSIF Architecture

This solution is not affected by Problems #1 and #2, as the invocation is decoupled from service definition through the WSDL document. WS-BPEL or any other language that supports WSDL orchestration could be used to define the composite service. However, parameter passing is not direct but by Java RMI, which affects performance.

5. Evaluation

We have implemented the previous three solutions with the provided architecture and we have developed two services for these solutions. We analyze the architectures, show some performance results for both services and, finally, we compare the results with expected values.

5.1 Prototype Applications

We introduce two composed services, which follow the service composition model defined in Section 2. These services offer a complete and final functionality, as they are orchestrated by OSGi services in the Abstract level, which are implemented by bundles located in a repository (Implementation level).

Service # 1: *Sport Tracker*. This service, described in Section 2, consists of 4 OSGi services, published by 4 provider bundles; two of them publish a PulseService service, another one a LocationService and the last one a MapService. In the Implementation level, to obtain the location information, the Location Service is resolved to a provider bundle which manages the internal GPS of a Nokia N97, and the PulseService can be resolved to either a B600 FRWD heart rate monitor (FRWD) or a BT microX Medical RGB (RGB) pulse oximeter, depending on information for the resolution process stored in the SDL document of the Sport Tracker service, which we don't take into consideration for benchmark analysis. Only for carrying out comparative tests between solutions we have simulated the behavior of these devices, in order to exclude Bluetooth connection errors and data access delays from the benchmark analysis.

Service # 2: *Fire Presence*. This more complex service consists of 3 OSGi services (PresenceService, Camera and MMSService) and 12 provider bundles, which control 10 presence sensors, a presence camera and a MMS (Multimedia Messaging Service) server. The presence sensors are responsible for launching an event every time they detect any kind of

presence or movement in the environment and the camera is responsible for, once an event has been launched by any presence detector, taking a picture to see the physical object that triggered the sensor. The MMS bundle has the functionality to transmit the photo by the camera using MMS. Unlike the previous case, access to device functionality has been simulated to carry out all tests.

5.2 Qualitative Comparison

We perform a qualitative analysis of selected solutions taking into consideration the solution implemented and the architecture characteristics. A summary of the evaluation comparison can be seen in **Table 1**. SS means Specific Service, CSwPG means Common Service with Proxy Generation and WI means WSIF Invocation.

Table 1. Qualitative comparison summary

Attributes	Solutions		
	SS	CSwPG	WI
Architecture complexity (# of bundles)	One bundle for each provider bundle	Two bundles for each provider bundle	One bundle for each WS and bundles for WSDL generation
Difficulty of architecture development	Simple programming structures and reflection	Dynamic code generation	Knowledge of Apache Framework is required
Flexibility (adaptation to unknown bundles)	Immediate bundle integration	Immediate bundle integration	Immediate bundle integration and WSDL specification

In terms of architecture complexity, the SS architecture requires fewer bundles to work properly, because it does not need a proxy for each provider bundle, required by the CSwPG architecture. The CSwPG architecture is the most complex one, because it requires an extra bundle for each provider bundle. This architecture obtains the highest value for the SRP (Service Realization Pattern) metric in [18], which measures the number of bundles that exposes services by indirect exposure (i.e. requires additional bundles or proxies to mediate between a service and an IT system). The WI solution also requires extra bundles to generate the WSDL interface.

Regarding the level of difficulty for architecture developers, we analyze the code complexity of the solutions. The code of the SS solution must support reflection and WSIF architecture requires knowledge of the Apache framework. The dynamic generation of code that supports CSwPG is, in our opinion, the most complex mechanism.

With respect to the flexibility, the integration of new bundles is immediate with the three solutions if we properly specify the interfaces of provider bundles. Service selection in the three cases is dynamic. Therefore, the solutions are very flexible and agile according to the DSSS (Dynamic vs. Static Service Selection) metric [18], which evaluates the number of services that are selected dynamically over the total number of services that are selected dynamically or statically. We believe the WI solution is the most flexible option, as it converts service specification to a standard format (WSDL).

5.3 Performance Evaluation

We carried out a performance evaluation, based on the two services above. The tests have been

performed by codifying the following criteria in the SDL document that feeds the proposed architectures:

1. Execution time of CSwPG, SS and WI mechanisms for the *Sport Tracker* service, without taking into account the proposed architecture, service validation, error and inconsistency handling and logging features.
2. Measurement of the execution time for 1, 10, 100 and 1000 concurrent *Sport Tracker* (service # 1) and *Fire Presence* (service # 2) services. The execution time is the time it takes to run a service iteration after registering the required bundles for the service (in both OSGi and WSIF cases).

The result values offered by these web services have been simulated, as in the OSGi solutions, to avoid wrong measurements due to transmission delays or information processing errors. The invocation process is split into two subprocesses: service instantiation and data interchange.

In our tests we have observed a fairly linear behavior of the solutions, so that we only show, in **Table 2**, the performance results of the instantiation of one *Sport Tracker* service and the interchange of 1000 data.

Table 2. Performance results for SS, CSwPG and WI mechanisms in ms

	Instantiation	Interchange	Total
SS	92	51	143
CSwPG	950	78	1028
WI	390	211	601

As **Table 2** shows, service instantiation for SS is much lower than CSwPG's, since the latter has to dynamically generate proxies and register them in the OSGi platform, which take about 300 ms for each proxy. In the data interchange process the performance is similar, with the particularity that in the CSwPG solution, dealing with a proxy requires two service invocations for each invocation in SS, so the average time per invocation is about 40 ms, lower than in the SS case because CSwPG does not use reflection. Regarding WI solution, the instantiation time is also high (having to interpret the WSDL service interface) and the interchange time is also very high (using object serialization in RMI rather than reflective or direct methods). Other reviewed performance analysis [10] confirms these values. The obtained results by performing the second test in these solutions are shown in **Table 3**.

Table 3. Execution time for SS, CSwPG and WI architectures

# of concurrent services	Execution time (ms)					
	Service #1			Service #2		
	SS	CSwPG	WI	SS	CSwPG	WI
1 service	8	13	69	38	63	297
10 services	71	109	613	322	511	2559
100 services	641	965	5596	2540	3908	22870
1000 services	5024	7532	51432	17381	27380	206646

These measurements are made on a Dell M1530 with Core 2 Duo CPU T8100 @2.10 GHz and 3.50 GB of RAM. We have used Eclipse Helios Release 1 environment with Sun

JavaSE-1.6 and OSGi Equinox 3.6.1. We have selected the average value of three measurements for each case. The memory consumption in all cases has not exceeded 150MB (≈ 4%).

To interpret the results, **Fig. 8** and **Fig. 9** show the invocation time per service iteration for the Sport Tracker and Fire Presence services. WI solution has a very slow execution time compared to the other two solutions, among which SS is the fastest. Due to JMV compilation time optimization mechanisms (e.g. method inlining), the average invocation time is decreasing in all architectures and more noticeable in Service # 2, which is the most complex (more interactions between bundles), which reaches 50% in 1000 services.

After evaluating the results of both tests we consider that the wrapping mechanism of Web services as software components is not worth for situations of intensive use of local components or services. Also, the CSwPG solution has poor performance for situations with not too many invocations, due to the instantiation time, whereas the invocation time is ≈ 35% slower. We conclude that this solution is valid for flexible local communication, but more appropriated for distributed communication, for which dynamic proxy generation was designed.

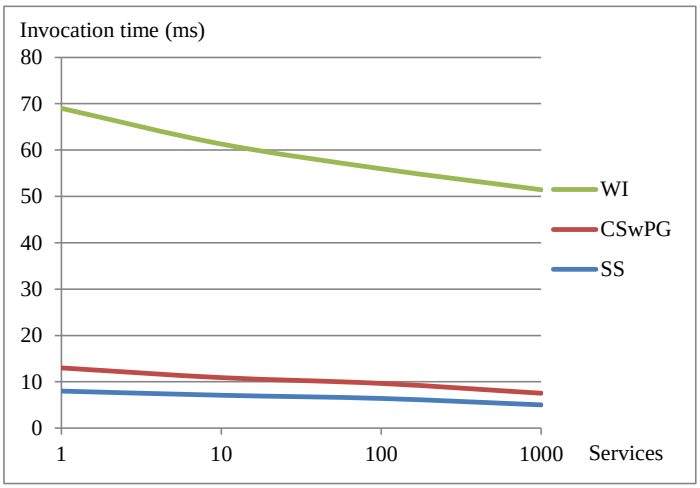

Fig. 8. Performance results for Service # 1

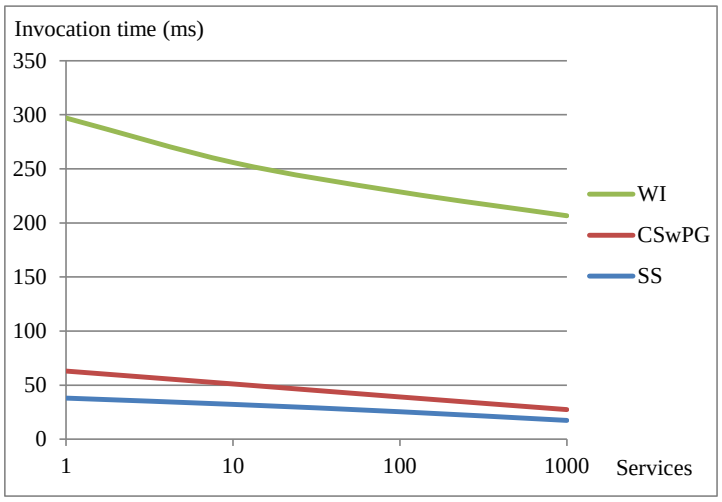

Fig. 9. Performance results for Service # 2

6. Conclusion

It is envisioned that some systems will shift from current machine-to-human communications to the machine-to-machine paradigm with the rapid penetration of embedded devices [19]. This work has presented the OSGi technology as a candidate to influence future M2M solutions by enabling communication with bundles that did not previously exist in the platform. The use of OSGi is not restricted only to the integration of home appliances but has the potential to enable communication with any device to be networked and controlled by other devices. The OSGi technology is suitable for our composition model, based on composed services, defined in SDL documents, abstract services, which consist of a definition of service's functionality and concrete services, which represent the concrete implementation of the service logic behavior. After reviewing related work in flexible service composition we present the SS and CSwPG OSGi solutions and a solution from the world of Web Services (*WSIF invocation*) to deal with unknown components. The proposed architectures have been developed and evaluated theoretically and by a performance analysis.

We conclude that, although the solution to convert bundles into WS (by generating their WSDL document) and orchestrate them with BPEL-like languages provides more flexibility, the performance of this solution is far below compared to reflective mechanisms or direct access to local components. Although the performance of the CSwPG solution is close to the SS's (around 35% worse), the instantiation time and complexity of the CSwPG architecture makes us consider using it only in environments where it is needed (remote invocation and distributed bundles), which leaves the SS solution as the best default option and specifically for systems with limited resources or where performance becomes a key factor.

References

[1] Francisco Curbera, Rania Khalaz, Mirmal Mukhi, Stefan Tai and Sanjiva Weerawarana, "The next step in web services," *Communications of the ACM*, vol. 46, no. 10, pp. 29-34, Oct. 2003. Article (CrossRef Link)

[2] Marco Autili, Vittorio Cortellessa, Antinisca Di Marco and Paola Inverardi, "A conceptual model for adaptable context-aware services," in *Proc. of Web Services Modeling and Testing*, pp. 15-33, Jun. 2006.
[3] Marco Autili, Paolo Benedetto and Paola Inverardi, "Context-aware adaptive services: the PLASTIC approach," in *Proc. of 12th Int. Conf. on Fundamental Approaches to Software Engineering*, pp. 124-139, Mar. 2009. Article (CrossRef Link)
[4] Rebeca P. Diaz Redondo, Ana Fernandez Vilas, Manuel Ramon Cabrer, Jose Juan Pazos Arias and Marta Rey Lopez, "Enhancing residential gateways: OSGi service composition," *IEEE Transactions on Consumer Electronics*, vol. 53, no. 1, pp. 87-95, Feb. 2007. Article (CrossRef Link)
[5] Françoise Baude, Imen Filali, Fabrice Huet, Virginie Legrand, Elto Mathias, Philipe Merle, Cristian Ruz, Reto Krummenacher, Elena Simperl, Christophe Hammerling, and Jean-Pierre Lorre, "ESB federation for large-scale SOA," *ACM Symposium on Applied Computing*, pp. 2459-2466, Mar. 2010. Article (CrossRef Link)
[6] Marlon Dumas, Murray Spork and Kenneth Wang, "Adapt or perish: Algebra and visual notation for service interface adaptation," in *Proc. of the Int. Conf. on Business Process Management*, pp. 65-80, Sep. 2006. Article (CrossRef Link)
[7] Henry Muccini, Andrea Polini, Fabiano Ricci and Antonia Bertolino, "Monitoring architectural properties in dynamic component-based systems," in *Proc. of the 10th Int. Symposium on Component-based Software Engineering*, pp. 124-139, Jul. 2007. Article (CrossRef Link)
[8] Michele Trainotti, Marco Pistore, Gaetano Calabrese, Gabriele Zacco, Gigi Lucchese, Fabio Barbon, Piergiorgio Bertoli and Paolo Traverso, "ASTRO: Supporting composition and Execution of web services," in *Proc. of the Int. Conf. on Automated and Planning Scheduling*, pp. 495-501, Jun. 2005. Article (CrossRef Link)
[9] Juergen Anke and Christian Sell, "Seamless integration of distributed OSGi bundles into enterprise processes using BPEL," in *Proc. of the Conference on Communication in Distributed Systems*, pp. 1-6, Mar. 2007.
[10] Kiev Gama, Walter Rudametkin and Didier Donsez, "Using fail-stop proxies for enhancing services isolation in the OSGi service platform," in *Proc. of the 3rd Workshop on Middleware for Service Oriented Computing*, pp. 7-12, Dec. 2008. Article (CrossRef Link)
[11] Paolo Bellavista, Antonio Corrado, Damiano Fontana and Stefano Monti, "iPOJO-based middleware solutions for self-reconfiguration and self-optimization," *KSII Transactions on Internet and Information Systems*, vol. 5, no. 8, pp. 1368-1387, Aug. 2011. Article (CrossRef Link)
[12] Clement Escoffier, Richard S. Hall and Philippe Lalanda, "iPOJO: an extensible service-oriented component framework," in *Proc. of the IEEE Int. Conf. on Services Computing*, pp. 474-481, Jul. 2007. Article (CrossRef Link)
[13] Pang-Chieh Wang, Cheng-Liang Lin and Ting-Wei Hou, "A service-layer diagnostic approach for the OSGi framework," *IEEE Transactions on Consumer Electronics*, vol. 55, no. 4, pp. 1973-1981, Nov. 2009. Article (CrossRef Link)
[14] Wei-Ting Cho, Chin-Feng Lai, Yueh-Min Huang, Wei-Tsong Lee and Sing-Wei Huang, "Home energy management system for interconnecting and sensing of electric appliances," *KSII Transactions on Internet and Information Systems*, vol. 5, no. 7, pp. 1274-1292, Jul. 2011. Article (CrossRef Link)
[15] Jiankun Wu, Linpeng Huang, Dejun Wang and Fei Shen, "R-OSGi-based architecture of distributed smart home system," *IEEE Transactions on Consumer Electronics*, vol. 54, no. 3, pp. 1166-1172, Aug. 2008. Article (CrossRef Link)
[16] Matjaz Juric et al., "BPEL Cookbook: Best Practices for SOA-Based Integration and Composite Applications Development," *Packt Publishing*, Jul. 2006.
[17] Philipp Leitner, Florian Rosenberg and Schahram Dustdar, "Daios: Efficient dynamic web service invocation," *IEEE Internet Computing*, vol. 13, no. 3, pp. 72-80, May-June 2009. Article (CrossRef Link)
[18] Mamoun Hirzalla, Jane Cleland-Huang and Ali Arsanjani, "A metrics suite for evaluating flexibility and complexity in service oriented architectures," *Lecture Notes in Computer Science*,

vol. 5472, pp. 41-52, Dec. 2008. Article (CrossRef Link)
[19] Yan Zhang, Rong Yu, Shengli Xie, Wenqing Yao, Yang Xiao and Guizani, M., "Home M2M networks: Architectures, standards, and QoS improvement," *IEEE Communications Magazine*, vol. 49, no. 4, pp. 44-52, Apr. 2011. Article (CrossRef Link)

Ramon Alcarria received his Master degree in Telecommunication Engineering from the Technical University of Madrid in 2008. Currently, he continues his studies as a PhD student and participates in several national and international research projects. His research interests are Service Architectures, Sensor Networks, Service Composition and Prosumer Environments. He is a member of IEEE, IEEE Communication Society and ACM.

Tomas Robles received a M.S and Ph.D. degrees in Telecommunication Engineering from Technical University of Madrid in 1987 and 1991, respectively. Since 1991 he is associate professor on Telematics Engineering at the E.T.S.I. Telecommunication of the Technical University of Madrid. His research interest is focused on Advanced Applications and services for Broadband networks, both wired and wireless networks.

Augusto Morales received his Bachelor's degree in 2007 from the University of Panama, and his Maser's degree from the Technical University of Madrid in 2010. Since 2008 he has been working in several areas related Service Architectures, NGN and Network Security while he pursues his PhD. He holds several IT Certifications such as CEH, Security+, Linux+ and CCSE.

Sergio Gonzalez-Miranda received his Bachelor's degree in Informatics from Aguascalientes Institute of Technology (Mexico) and he is a PhD student for the Technical University of Madrid. His main research interests include: Telco and Web 2.0 mashups, Fixed-Mobile convergence, NGN services and security.