SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2013; **43**:33–65
Published online 29 January 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2101

Enhancing OSGi with real-time Java support

P. Basanta-Val*,†, M. García-Valls and I. Estévez-Ayres

Departamento de Ingeniería Telemática Universidad Carlos III de Madrid 28911, Leganés, Spain

SUMMARY

OSGi was designed with embedded systems in mind, its current support is insufficient for coping with one main characteristic of many embedded systems: real-time performance. This article analyzes different key issues in providing OSGi with real-time Java performance covering motivational issues, and different integration ways and challenges stemming from the integration. It also contributes a general framework for introducing real-time performance in OSGi, which is called the real-time for OSGi framework. The framework uses real-time Java virtual machines and the real-time specification for Java. The adoption of this framework allows cyber-physical systems to experience real-time Java performance in their applications. The framework introduces several integration levels for OSGi and real-time specification for Java, and specific real-time OSGi services. An empirical implementation was carried out using standard software, which was extended with the new defined services. Copyright © 2012 John Wiley & Sons, Ltd.

Received 23 February 2011; Revised 10 November 2011; Accepted 22 November 2011

KEY WORDS: OSGi; real-time; RTSJ; Java; real-time Java

1. INTRODUCTION

One of the main initiatives that contribute to the adoption of component-based advantages in consumer electronics and in the home is the OSGi [1]. OSGi defines a reference framework on which application providers may deploy applications, called *bundles* in OSGi's terminology, which use other bundles and export their services to other bundles. OSGi is Java centric and inherits its pros and cons. On the one hand, it means that bundles may use a large number of APIs that help develop new services. On the other, the developer also has to deal with other Java issues related to poor efficiency and performance, garbage collection interference, and other sources of delay [2, 3].

One of the key requirements for the success of OSGi is standardization [1]. As long as the initiative maintains a coherent and updated description of the basic services included in a reference implementation and defines different profiles for specific domains, the interoperability among different applications and platform implementers will be preserved. This initiative should be dynamic and consider that new domains, not considered at its first design, may appear. One of these new environments is real-time systems. To date, OSGi does not target directly at having specific support for real-time performance in its core or as some restricted profile.

Real-time performance refers to applications where application correctness depends on the time at which a result is produced, in addition to the logical correctness of the result itself. In hard real-time applications a deadline is a typical real-time requirement. The smart home [4] offers us many examples of real-time applications such as TV and home surveillance systems. Many digital TVs [5, 6] use multimedia software stacks to reduce deployment costs and improve adaptability to new formats. On them, the primary goal is to deliver certain amount of frames per second (20 FPS)

*Correspondence to: P. Basanta-Val, Departamento de Ingeniería Telemática Universidad Carlos III de Madrid 28911, Spain.
†E-mail: pbasanta@it.uc3m.es

while keeping a low loss ratio. Other applications with real-time performance requirements [7, 8] have sensors spread throughout the home, notifying the home owner about intruders. In this case, the response may have tight constraints on reaction time to the intruder (to avoid certain attacks), which may involve actions such as: take a picture of the intruders, send it to the Internet and notify the owner about new intruders.

Traditionally, these applications have been deployed statically [9], each part of an application resides in a characterized node (in terms of other applications running on it) and applications may not appear and disappear over time. It is also typical that the feasibility analysis is carried out off-line making easier the real-time applications development. However, addressing a dynamic situation where the number of real-time applications running over time change is very important when applications do not have to be running forever or from the initialization of the system [10]. At that stage it is where OSGi is interesting for real-time Java programmers, as a system that may control life-cycle in applications [11]. The set of functionality it offers (i.e., start, stop, update, install and uninstall primitives) may provide developers with standard deployment and updating for real-time applications.

From the point of view of an OSGi developer, the use of techniques like those typically used in real-time is also beneficial for reducing the amount of resources required. If some kind of formalism is enforced (i.e., real-time scheduling techniques [12]) then several real-time applications (e.g., video decoder and surveillance modules) may cohabit with other general-purpose applications meeting globally their application constraints.

From a technological perspective, OSGi offers a multiapplication framework for multiprogramming in which bundles represent applications that may be started and stopped at run-time and when the framework initializes. Unfortunately, OSGi is not particularly well-equipped for real-time because the platform itself and many of its bundles are Java programs and inherit Java's issues with real-time. These issues include: a fuzzy priority scheduling; the garbage collectors, which introduce high latency and overhead; and the byte-code execution model, which is slow when compared against C/C++ native applications. Some sources of unpredictability are more relevant than others because of the special architecture of OSGi. One of these mechanisms is the class loader of Java. In OSGi class loaders are essential to support multiple versions of a bundle to exist in a virtual machine. However, many times class loading is discarded from real-time phases (with remarkable exceptions like [13]) because of its higher unpredictability.

The provision of real-time performance in Java is being addressed by the real-time Java community [14–16]. This global effort contributes techniques used in real-time systems to Java's programming model and defines specific approaches to issues stemmed from the use of Java in real-time. To date, it has produced a specification, the real-time specification Java (RTSJ) [14, 15], which is implemented by several commercial and open-source initiatives [16–20]. In addition, it works on a distributed version of the specification: the distributed real-time specification for Java (DRTSJ) [21, 22]. RTSJ improves local predictability in a virtual machine while DRTSJ provides end-to-end real-time performance.

To date, the integration of real-time Java and OSGi is still an open issue that requires significant effort to become a real-time OSGi technology. Significant progress has been made in the definition of admission control techniques for OSGi ([10, 23–35]). However, these techniques are silent on OSGi's important issues like the definition of general real-time services, which architects a real-time framework for OSGi. This is specific contribution carried out in this article, that is, the definition of services for an RTSJ-based OSGi. The proposed architecture includes key entities and roles as new services in OSGi. It is also extensible and may admit several integration levels with different requirements.

Previous works (i.e., [10, 23, 24, 35]) may benefit from the work described in this article from different perspectives. The previous admission controllers may be integrated as different implementations of a scheduler service. Furthermore, the empirical evaluation offers insight on the performance a real-time OSGi enabled environment offered to the application. All together (the previous admission controller and the new services) may impulse the development of RT-OSGi. In addition, this article also identifies several aspects that are useful for other researchers to integrate RTSJ and OSGi.

DOI: 10.1002/spe

The remainder of this paper contributes a series of enhancements for the OSGi framework to offer real-time performance. Section 2 introduces OSGi, paying attention to the concept of bundles, services, and the functionality offered by OSGi to the application; they are required to introduce the architecture developed. Section 3 deals with the support offered by real-time Java to develop real-time applications and is focused on the benefits of having real-time Java in OSGi. Section 4 analyses different alternatives to integrate real-time Java with OSGi. The rest of the article is focused on defining and evaluating an integration framework between OSGi and RTSJ named real-time for OSGi (TROSGi). Section 5 presents its architecture while Section 6 defines an API for the architecture developed. Section 7 studies the empirical performance results obtained with TROSGi on a prototype, which runs on commercial-off-the-shelf software. Section 8 connects this work with other pieces of work from both real-time Java and OSGi communities. Finally, Section 9 concludes and relates our ongoing work.

2. THE OSGI FRAMEWORK

The Open Services Gateway initiative may be described as a technology that reduces the complexity of software development by means of component based software engineering mechanisms. The basic idea is that rather than providing new monolithic applications, the framework promotes that applications use standard modules, namely components, which may be reused over time in several applications.

Another type of techniques used in OSGi to reduce system development costs are the SOA tenets. In OSGi, services export their functionality that is available to compose applications dynamically.

2.1. Layering

The functionality offered by the OSGi framework is divided into the following layers (Figure 1):

- *Security layer*. This layer extends Java's security with a packaging format and runtime interaction models.
- *Module layer*. It defines a modularization model for Java with strict rules on how to hide and export packages.
- *Life-cycle layer.* Life-cycle management defines how bundles are started/stopped, installed, uninstalled and updated. Life cycle management requires the module layer to be operative, whereas the security layer is not strictly required.
- *Service layer*. The service layer provides a bundle model, which decouples bundle specification from bundle implementation.

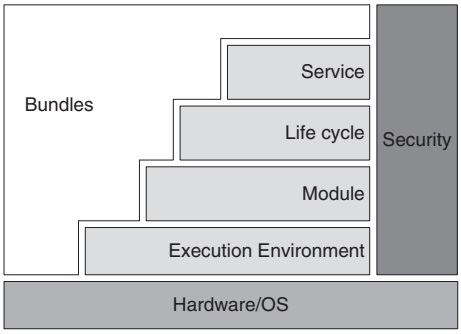

Figure 1. OSGi layering.

DOI: 10.1002/spe

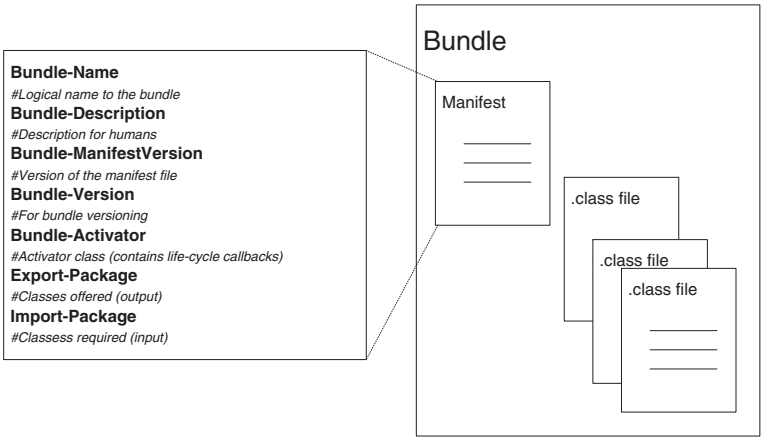

Figure 2. Bundle structure and the manifest file.

2.2. Bundles

A bundle represents a deployable unit. It does not define how or when it is installed, uninstalled, or updated. The underlying infrastructure decides when such actions are taken. Internally, the bundle is packed in a jar file that contains a manifest file and application-specific classes (Figure 2).

The descriptor, which may be created by an Integrated Development Environment (IDE), includes information about the name of the bundle, a human-readable description of its contents, the version of the bundle, and input and output dependencies. It also contains an *activator* entry that defines the class that contains the code executed when the bundle is started and stopped.

2.3. Service registration and standard services

The OSGi framework offers an API that allows a bundle to be accessible to other services. There are two primary mechanisms by which a bundle may use resources from another bundle. The first are the export and import sentences included in the descriptor. By using this mechanism, one bundle may export part of its API, which is accessible from another bundle. Another possibility is to use a registry, bundles may registry themselves, look up, and retrieve the list of available services by using the API. Dynamically, the bundle may use `registerService(class, name)` to export an interface from one bundle. This interface may be accessed from another bundle dynamically using a `getServiceReference("name")` statement.

The OSGi has characterized a number of services. They are typically grouped into system services (e.g., log service, and user admin service), protocol services (hypertext transfer protocol (HTTP) service, universal plug-and-play service, and Device Management Tree (DMT) admin service) and miscellaneous services (wire admin, XML parsers, and position specification). From the point of view of an implementation, not all these services have to be available in an OSGi environment. The specific contributions of this paper to the list of services are new services: real-time Java access, a real-time characterization service, a scheduler service, a recovery service, and a compositor service. They are explained in detail in Sections 5–7.

3. REAL-TIME JAVA ENHANCEMENTS OFFERED TO OSGI APPLICATIONS

This section reviews the abstractions defined in real-time Java, one-by-one, taking into consideration its impact on the OSGi programmer and its interest for OSGi applications. The analysis covers only centralized real-time Java (i.e., RTSJ) and set saside DRTSJ for future work.

3.1. Introducing centralized real-time support inside OSGi

The real-time specification for Java (RTSJ) was developed by the JSR-1 and JSR-282 (Java Specification Request) expert groups. JSR-1 produced the first specification, which is now under refinement

DOI: 10.1002/spe

Table I. Impact on OSGi from RTSJ techniques grouped by its different enhancement areas.

Enhancement area (RTSJ)	Contents	Impact on programmers	Interest for OSGi applications
Thread scheduling and dispatching	Scheduler	Medium	Medium
	Priority scheduler	Medium	Medium
	Real-time threads	Medium	High
	No-heap threads	High	Low
	Timers	Medium	Medium
	Time classes	Low	Medium
Memory management	Memory Area	High	Low
	Heap Memory	Low	Low
	Immortal Memory	High	Low
	Scoped Memory	High	Low
	Real-time garbage collection	Low	High
Synchronization and Resource management	Synchronization policies	Low	Low
	PIP	Low	Low
	PCE	Medium	Low
Asynchronous event handling	Event handler	Medium	Medium
	Async Events	Medium	Medium
	Async Event Handler	Medium	Medium
Transfer of control	Asynchronous transfer of control	Medium	High
Thread termination	Asynchronous thread termination	Medium	High
Physical memory access	Byte array access to raw memory	Medium	Medium

under the JSR-282's umbrella. The specification defines seven general guiding principles that define general goals for the specification. The first is that real-time should not be constrained to any particular environment (e.g., the use of J2ME, or some specific environment); in its definition, RTSJ should be generic enough to be deployed in multiple runtimes, including — of course — implementations for OSGi. It should be backward compatible with old nonreal-time Java applications too.

Real-time specification for Java proposes seven enhancement areas for real-time Java (Table I). The goal in each enhancement is to improve the predictability of the virtual machine in a specific aspect. The seven areas refer to threads scheduling, memory management, new synchronization protocols, asynchronous events, transfer of control, thread termination, and physical memory access.

From the point-of-view of a programmer, not all these features are simple APIs. Some of them are really (e.g., the programming models for scoped-memory [25]) difficult for a programmer and others are really simple (e.g. the use of a real-time priority inheritance protocol in Java locks or a real-time garbage collector).

Among all of them, thread scheduling and dispatching is the main category, which includes mechanisms that allow a programmer to define periodic, sporadic, and aperiodic real-time activities. The scheduler class may carry out admission control (for instance, if the amount of CPU consumed is more than 100%, it does not allow a new application to execute). In essence, the feasibility algorithm is RTSJ implementation dependent and may do nothing more than always return true (feasible), provided there are no aperiodic threads in the feasibility set.

In RTSJ there are two main threads: real-time threads and nonheap threads; the first is simple and may be used as a substitute for the thread class. From the programmer's perspective, the only difference is the name of the class. However, nonheap threads are extremely complex from the point-of-view of applications because they have to be coded keeping the *assignment* and *single-parent* rules [25] in mind. The advantage provided by nonheap real-time threads (and its nonheap memory model) is that they allow programmers to avoid one of the main sources of unpredictability in Java: the garbage collector.

3.1.1. Impact on OSGi. From the point-of-view of other nonreal-time applications, for example, other OSGi bundles that are installed in the system, real-time threads and the use of timers and high-resolution classes are the most interesting features because they allow having certain control on applications.

DOI: 10.1002/spe

Memory management includes classes to enable the parameterization required for the use of real-time garbage collectors. Real-time garbage collection [26] has a low impact on the programmer (some garbage collectors may not need complex parameterization) and this type of mechanism may be interesting in other OSGi bundles that do not want to suffer the penalty of stop-the-world garbage collectors (because they may block applications for seconds). The second mechanism is to remove the interference from the garbage collector using regions [27]; regions follow a cactus automatic memory management discipline that enables predictable deallocation of objects. However, regions are complex from the point-of-view of the programmer; not all Java applications may be easily developed in this discipline which requires references among objects to keep a tree structure. Therefore, nonreal-time OSGi applications (bundles) should not have a special interest in this mechanism, which should not be used outside specific real-time profiles.

Real-time systems require the use of real-time synchronization protocols (e.g., priority inheritance protocol (PIP) and priority ceiling protocol) to avoid or bind priority inversion among applications. These mechanisms, which are essential in a real-time application, do not impact highly on general-purpose applications. Nevertheless, it may be of some interest on PIP, in some specific cases.

In plain Java, the programmer does not have control on which of the threads awaiting a lock is the one that enters it. Potentially, any thread awaiting the lock in a lock is the one that gets the lock. To have more control on this issue, Java offers a `notifyall` statement that potentially gives an opportunity to all threads blocked in the lock. This statement may be used to enforce an application-defined order to access the lock. Nevertheless, if the lock follows a PIP policy, then the thread that gets the lock is always the one with the highest priority. Therefore, in some specific applications where the thread to get access is the one with the highest priority, the protocol may offer a simpler support than having to activate all threads (by using `notifyall`) to explore which thread is the next to enter the resource.

Java has an event model for synchronous communications where the event thread calls the event handler of each thread. RTSJ has extended this model with asynchronous handlers that execute asynchronously. This allows complex event handlers to run without blocking the source of the event. The mechanism included in RTSJ is also interesting for general purpose OSGi programmers that may use it in event-driven applications that require asynchronous communication.

Another limitation of Java is that a thread cannot terminate another thread. The reason why that happens is that the second thread may hold resources that require certain management before being released. RTSJ addressed that issue with two mechanisms: asynchronous thread termination and asynchronous transfer-of-control. Both mechanisms allow execution control on one thread from another thread. This mechanism is general enough to be used by other general purpose OSGi bundles to destroy their threads in a safer manner.

Many real-time systems require low-level access to drivers mapped to memory. To improve this support from the virtual machine, the RTSJ defined special classes that map this memory to primitive data arrays. The mechanism is also interesting for low-level bundles that may use this mechanism to control low-level drivers into OSGi applications, directly from a Java application.

3.1.2. Impact on OSGi. The first enhancement area extends the distributed object model currently included in distributed real-time Java with facilities for end-to-end communications. The extra control offered to applications allows defining rules for connection management (e.g., when a new connection is open or closed and the number of threads awaiting incoming requests and events from the client). They also extend the model of the priority scheduler to offer end-to-end admission control, which guarantees certain maximum delay in communication. This mechanism is also interesting in general purpose OSGi applications that may to want to tune its performance and have certain control on how and when connections are established.

There are three main issues related to memory management in distributed real-time Java. The first is how to avoid the garbage collector in end-to-end communications. Applications may use a real-time garbage collector but they may also opt for the region model of RTSJ (including mechanisms like the nonheap remote object paradigm [30–32]). In addition, real-time Java's Remote Method Invocation brings in another problem: the use of distributed garbage collection (DGC) mechanisms to avoid the penalty of the garbage collector on end-to-end communications. DGC has an import

DOI: 10.1002/spe

impact on the programmer, which cannot use its traditional remote objects with a turned off garbage collector. If the collector is to be switched-off, these remote objects should include new methods to unreference remote objects when they are not in use. From the point-of-view of an OSGi application, the less interesting mechanism is the use of region-based memory models because they are complex. Real-time garbage collectors and distributed real-time garbage collectors have a lower impact on the applications and they may reduce the end-to-end response time remarkably.

Distributed real-time specification for Java, the main effort towards having a distributed real-time Java specification, is synchronous; that is, clients wait for a response from the server. Distributed events may enhance this support with an asynchronous event-driven communication similar to the model currently included in RTSJ. The model is also interesting for general purpose OSGi bundles that may communicate using a networked distributed event model in their general-purpose applications.

Lastly, distributed transfer-of-control and thread termination are two mechanisms that enable distributed thread termination and notification. From the point-of-view of the distributed model, its implementation is complex; it requires detection of failures and nodes that keep track of remote invocations that are running inside them. However, from the point-of-view of the programmer it is a powerful mechanism that enables distributed thread termination, which is also interesting in general-purpose OSGi applications.

4. KEY ISSUES AND APPROACHES IN REAL-TIME OSGI

Current real-time Java technology (i.e., the RTSJ and DRTSJ) may be used to improve predictability in OSGi. This section presents several alternatives to improve OSGi into four groups according to their natures.

4.1. Accessing real-time Java facilities

In its easiest way, OSGi may grant access to the RTSJ API. For instance, allowing a bundle to have access to `javax.realtime` and `javax.realtime.distributed` (i.e., the specifications currently available in real-time Java) enables programmers to design OSGi applications. The price paid for it is small; the underlying virtual machine should support the RTSJ and/or the DRTSJ. In this direct approach, the main advantage from the application perspective is to have a standardized control on its life-cycle (i.e., when it is started, stopped, installed, uninstalled, and updated). Currently, RTSJ and DRTSJ do not offer such functionality to the programmer, who has a starting point only.

4.2. Improving the predictability of the framework

A second family of improvements may extend the characterization of bundles to provide some type of real-time parameterization. Much information currently stored in an application (e.g., the priority of a thread and its cost) has to be stored in the source code, impacting on its portability negatively. Alternatively, the real-time information may be stored in the bundle descriptor in a file, which may be modified by design tools (e.g., to assign a priority according to a certain criteria) and accessed by the bundle (e.g., to get initial configuration parameters). Moreover, if the representation formats are chosen in the right way (i.e., using standards or de facto standards) then the descriptor file may be used in modeling tools.

Notice that the access to this file and some basic functionality may be added as a new service into the OSGi set of services. The definition of a service comes with advantages stemming from the OSGi framework such as the possibility of having different implementations for the same service. The main drawback is performance, which may be seriously reduced by the use of complex representation models.

Another important issue is the OSGi framework itself, which may be understood as a Java application that may benefit from having an underlying real-time virtual machine. OSGi offers certain services (e.g. security, module management, life-cycle management, and service discovery) that have to interact with other applications. Because these services are allocated with other real-time

DOI: 10.1002/spe

bundles, the framework requires a real-time characterization to be useful. One naïve approach is to force all threads that are running under the OSGi framework to have a priority lower than the priority of the real-time applications; another possibility is to use the framework only for scheduled updates performed in application maintenance. However, enhanced models may improve the basic model taking into consideration different aspects of their particular goals. For example:

- The *security layer* introduces on-line overhead related to the validation of its policies according to certain rules and constraints. An approach to alleviate that overhead is to move part of the validation process to design time and to choose on-line validation policies that do not take much time.
- The *module layer* is another source of indeterminism. It checks dependencies among bundles consuming resources (i.e., CPU and memory). Depending on the type of strategy used (e.g., if lazy activation is allowed or not) the performance varies remarkably. Another important challenge, which refers to the integration of this module, is to decide in which cases this validation may be part of a nonreal-time section of the application.
- The *life cycle layer* controls when a bundle is started and stopped. In these two operations the OSGi framework uses an internal thread to upcall the corresponding start and stop methods on the bundle activator class. These upcalls are carried out by threads allocated by the framework, which are presumably nonreal-time threads. To improve predictability in start and stop operations, the framework may change its implementation using real-time threads in the upcalls. In addition, the code required to start and stop bundles should be checked to establish worst-case scenarios in life-cycle operations.
- Lastly, the *service layer* allows publishing and recovering services from the platform by using user-defined names. In that case, the implementation should be rechecked to verify whether the implementation uses threads or not. It is very likely that no complex changes have to be included to produce a predicable service discovery.

  An additional enhancement is to add a real-time characterization (e.g. expressing the different cost of different service implementations) to the service definition. Nevertheless, to be compliant with the current OSGi, this type of improvement is better placed in a new real-time service.

4.3. Definition of new (real-time) services for OSGi

A third direction in which OSGi may be expanded is the definition of new real-time services. The combination among OSGi and a RTSJ-DRTSJ tandem may result in a new set of services for real-time applications. These services may offer powerful abstractions highly based on the features available in OSGi. The main difference between these services and the mechanism included in RTSJ and DRTSJ is its focal point. RTSJ and DRTSJ are more focused on threads, while a hypothetical OSGi technology would be focused on bundles (its operational unit).

- *An admission control service*. The current OSGi life-cycle model does not check the feasibility (e.g., if an application meets the deadline or not) dynamically. This is one of the directions to extend the current set of services described in OSGi. Taking bundle descriptors extended with a real-time characterization as an input, the new service could carry out this analysis using different existing scheduling techniques (including on-line and off-line) with different service implementations for each development. As an additional support, the admission controller may use underlying services from the framework to control (start/stop) bundles if it is necessary.
- *A fault detection and recovery service*. Many real-time systems have to assume faulty environments, in which typically errors appear over time because of misbehaviors in their modules. In that direction a new service in charge of detecting typical system failures is useful as a complement for another functionality that recovers (e.g., using a backup copy or just restarting a bundle) the system from failures. As in the previous case, the support offered by the current framework may help update different bundles.
- *A compositor service*. An advantage stemmed from SOA is that implementations are hidden and many different implementations, with different QoS models, may satisfy the same service.

DOI: 10.1002/spe

Therefore, having a piece of code in charge of selecting a running configuration in complex scenarios becomes useful as a service. This service may be used in the basic framework functionality to subscribe and unsubscribe different bundles to the system.

- *An adaptation service*. Lastly, the three previous services may be part of a high level service in charge of adapting the system performance to the availability of new bundles (which include services), failures in the system, different bundle implementations that may appear over time, and underlying resources. Furthermore, the real-time adaptation service itself may be a real-time application with deadlines.

4.4. Specific service optimizations

Lastly, the specification may opt for introducing real-time performance in any of the different services defined in the framework. Potentially, any service included in OSGi may be improved by having a real-time Java virtual machine support it. For instance, the HTTP service may update its implementation using real-time threads instead of its plain threads, thus increasing the predictability of the HTTP communication. It may also use budgets (called processing group parameters in RTSJ) to bind the interference introduced by the application. Lastly, part of these parameters may be available to the rest of the system changing its current public service definition into a new real-time HTTP service.

Table II summarizes the main advantages stemming from the adoption of the previous techniques. For each group of approaches, the table summarizes their advantages, the effort required to have the technique included in an OSGi implementation, and the impact of the expected impact of the technique on the current OSGi APIs. Among them, the first technique is a good candidate to achieve a low-cost integration between real-time Java interfaces and OSGi. On the other side of the balance, the definition of new real-time services is the approach that may gain more direct visibility on the APIs, with new services in the list of OSGi services.

The rest of the article (Sections 5–7) reports on the integration framework, called TROSGi, which is based on the elements identified in previous sections. The current implementation of TROSGi integrates RTSJ within OSGi defining new services for OSGi. Section 5 describes the architecture of TROSGi, Section 6 the new API defined for TROSGi, and Section 7 evaluates the performance of the temporal overhead introduced by their services.

5. THE TROSGI ARCHITECTURE FOR OSGI

This section defines an integration framework between OSGi and real-time Java, namely TROSGi. The architecture is described in a modular approach with different integration levels. Each integration level provides the OSGi developer with extra facilities and requires different implementation efforts. The integration levels shown in this section together with the API described in the next section may contribute to the adoption of real-time Java services in OSGi.

Table II. Approaches for the integration Of RTSJ with OSGi analyzed (summary).

Approach	Advantage	Effort req.	Impact on the OSGi API
Access to the RTSJ and DRTSJ API	Full API access Real-time performance	Low	Low
Predictability in the framework	Predictable life cycle Service management Characterization service	Medium /high	Low /medium
New (real-time) services	Scheduler service Recovery service Compositor service	Medium	High
Reimplementation of standard services	Real-time performance in current implementations API extensions in services	High	Medium /high

DOI: 10.1002/spe

Table III. Integration levels in real-time OSGi with real-time Java in TROSGI.

Integration level	Contents	Elements included	Impact on OSGi
RT-OSGi L0	Real-time Java available for OSGi bundles	Real-time Java virtual machine `javax.realtime` API	Real-time as an API
RT-OSGi L1	Real-time bundle description	Standard real-time bundle parameters	No changes in the core of OSGi
RT-OSGi L2	Enhanced OSGi for real-time performance	Admission control Fault tolerance Multiconstrained and adaptive admission control	Changes required in the implementation of OSGi

As shown in Section 4 there are many directions in which OSGi may be integrated with real-time Java. On the one hand, one may define a minimalistic approach that consists of a plain OSGi implementation running on a real-time virtual machine; its implementation is almost direct and it does not require special changes in OSGi. On the other hand, one may think of a real-time OSGi infrastructure as an ecosystem with a bundle descriptor that allows specific actions such as carrying out admission control, fault detection and recovery, and multiconstrained admission control.

In the terminology used in this section, the first approach refers to the Level 0 integration, and the second to the Level 1 and Level 2 integrations (see Table III). Level 1 contains a real-time characterization and Level 2 adds specific services for OSGi.

5.1. Level 0 integration

Level 0 refers to the minimal integration level for real-time Java and OSGi. In this level, the application sees real-time Java as another API available for applications (see Figure 3). There are no changes in OSGi, which is used to deploy applications. The advantage provided is that the application developer may access the real-time Java API, which may be invoked from other bundles.

Under these assumptions not many changes are required for having real-time Java performance on applications:

- The underlying virtual machine should support the `javax.realtime` API.
- The underlying operating system (if any) should offer basic real-time performance to the real-time Java virtual machine.
- The OSGi engine should export `javax.realtime` to the bundles. By default, the OSGi environment hides all `javax.*` classes to the upper layers. To access to the API, it requires that the API is exported in the framework.

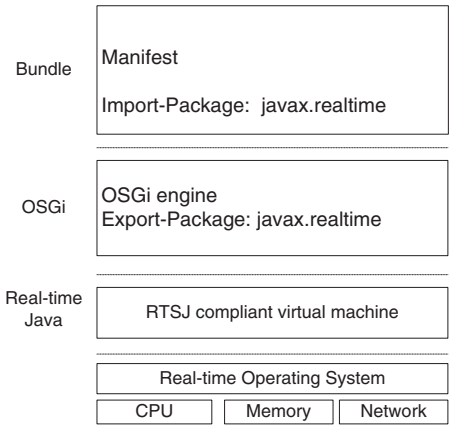

Figure 3. Software stack for the Level 0 integration in TROSGI.

DOI: 10.1002/spe

- Bundles should import `javax.realtime` in their manifest file. To allow an application to use a javax API, the bundle has to import it in the bundle. Without such import statement, the bundle cannot access to the real-time Java API (even, if it is exported from the framework engine).

5.1.1. Programming perspective. By using the infrastructure described, the OSGi programmer may develop bundles that use real-time support. It only has to hook its code to the life-cycle methods of the bundle.

Applications may use `start()` and `stop()` callback methods included in bundle activators to launch real-time applications. From the point-of-view of the real-time programmer, there is not a great difference between using a `static main` method and the `start`. Both are invoked from the underlying engine to start an application. In addition, the `stop` method may be seen as a deallocation mechanism useful to remove resources allocated at runtime. The model is closer to other Java technologies like *applets*, which have two callback methods, one for initialization and another for finalization.

Listing 1 shows the example of a bundle activator, which creates and launches a real-time task. In each loop, the task prints out `'RT hello'` (line 09) and sleeps 10 ms until the bundle stops.

The `start` method is invoked from the OSGi engine when the bundle is started. The basic real-time thread (lines 06-14) is similar to the nonreal-time thread, whereas advanced implementations carry out admission control on the local scheduler and may define CPU parameters. After launching the thread (line 15) the `start` method ends. The `rt` thread continues its printout until the OSGi engine decides to stop the bundle.

The bundle is stopped via a console manager or from other bundles that use the management API. For this example, there is an external entity that invokes the stop method. In this method the application should remove any resource allocated previously; in this particular example, it stops the thread previously allocated. This is accomplished (line 18) using the interrupt method defined by the RTSJ. This method raises an exception on the rt thread. This exception is handled at the thread (line 13), which exits its run method.

```
00: import javax.realtime.*;
01: import org.osgi.framework.*;
02: public class ExampleRT implements BundleActivator{
03: BundleContext ctx;
04: RealtimeThread rt;

05: public void start(BundleContext context) throws Exception {
06:       rt=new RealtimeThread(){
07:        public void run(){
08:         try{
09:            do{ System.out.println("RT hello");
10:           RealtimeThread.sleep(new AbsoluteTime(10,0)); //mit=10 ms
11:          }while (true);
12:         }catch(Exception e){
13:            System.out.println("STOP called: bye, bye");}
14:       };
15:       rt.start(); //Launch the real-time thread
16:  }

17: public void stop(BundleContext context) throws Exception {
18:       rt.interrupt(); //Launch an exception
19:  }
20:.}
```

Listing 1. Example of a bundle activator that starts/stops a real-time thread

DOI: 10.1002/spe

To work, the bundle still needs some information regarding its name, symbolic name, description, an activator class, and the import package. The most relevant parameters in this example are two. The first is the activator package (`ExampleRT`), which should refer to the class that activates and deactivates the bundle. The second is the set of packages imported, which should include `javax.realtime`.

```
00: Bundle-Name: Hello RT Threads
01: Bundle-SymbolicName:
es.uc3m.it.drequiem.trosgi.RTthreads
02: Bundle-Description: A Hello World bundle for RTSJ
03: Bundle-ManifestVersion: 2
04: Bundle-Version: 1.0.0
05: Bundle-Activator: ExampleRT
06: Import-Package: org.osgi.framework, javax.realtime
```

Listing 2. Bundle descriptor for Listing 1

5.2. Level 1 integration

Typically, real-time applications require a real-time parameterization to work. This includes parameters used by underlying real-time systems such as periods, deadlines, costs, and priorities. The virtual machine uses them to guarantee some type of global performance. Although these parameters may be included in the manifest without modifications, there are many reasons to add them in a standardized format. Some of them include the use of external tools that verify that the performance required from the system is the right one. The standardization process of this information is what defines the Level 1 integration boundaries.

The previous software stack requires changes to accommodate the new requirements. Figure 4 shows a new architecture for the new level. This application requires a new package (`es.uc3m.it.trosgi`) that includes a new bundle that allows access to the resource information model. The approach followed is backward compatible; plain bundles may access to a characterization file.

The Level 1 integration includes the definition of a real-time characterization for bundles and the API required accessing these resources. Continuing with the previous example, the bundle should contain now a reference to a resource model descriptor (e.g., an `xml` file like in Listing 4).

The file contains a resource model that may be used from the bundle and from external tools. It describes the content of the bundle as a set of tasks which consume certain amount of memory and processor (Table IV).

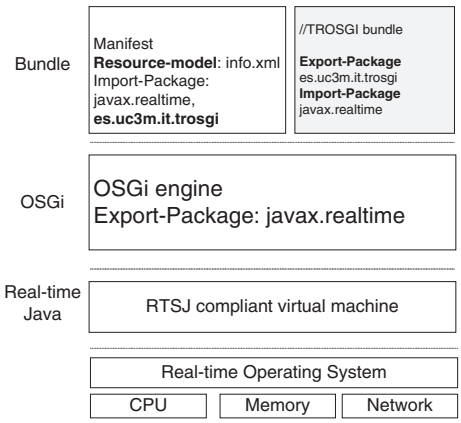

Figure 4. Software stack for the Level 1 integration in TROSGI.

DOI: 10.1002/spe

Table IV. Resource model for real-time bundles.

	Contents	Example
Application	Processor	<p1>
	Schedulable	<sch1><sch2>
	Lock	<lock1>
	Memory model	<mm1>
	GC	<gc1>
	Regions	<reg1>
CPU	ID	This example
	Processing group parameters schedulable	
	Schedulable	
	-Cost	- 1 ms
	-Deadline	- 10 s
	-Min_Period	- 10 sect
	-Release	- 15 Feb. 2010
	-Priority	- <10>
	Lock	
	-Ceiling	- <10>
Memory	Physical memory model	
	-ID	- Model A
	GC	
	-Allocation rate	- 100 MB/s
	- Memory required	- 100 MB
	Regions	
	-Size	- 100 MB
	-Type	- Immortal
Network	Not addressed yet	Not addressed yet

5.2.1. Programming perspective. This model may be accessed from a new service that accesses the manifest file and to the corresponding resource model. The service is able to obtain data dynamically using the identifier of a bundle. For instance Listings 3, 4, and 5 show how the thread may read its `mit` (minimum interarrival time) parameter using Level 1 facilities. Listing 3 corresponds to the implementation of the application, Listing 4 to the changes introduced in the manifest file, and Listing 5 to the `xml` file packaged with the bundle that stores the `<mit>` value (10 ms).

Any OSGi application that requires using a service has to search for the service in first place. In the example, such action is taken in lines 11–14 of Listing 3. The name given to the Level 1 facility is `'CharacterizationService'` and it is supposed to be launched from another bundle (e.g. `CharacterizationServiceImplementation`), which should export the characterization service using the OSGi framework (see Section 6 for the API of this service).

After getting the service, it may then be used. In the example analyzed, it is sufficient to know that it has a `getRealtimeCharacterization` method that returns a Document Object Model (DOM) tree (which may be accessed via the `org.w3c.dom` package) that corresponds to the contents of the resource model. Internally, the method reads the headers of the bundle and constructs the `xml` tree from the manifest file (which in this example corresponds to `resourcesmodel.xml`).

The DOM tree may be used by the application to set its configuration parameters. For instance, it may read the `mit` parameter exploring the tree. In this simple example, a call to the `getElementsByTagName` method gets the `<mit>` tag, which is used to read its associated integer value (i.e., 10,000). The application requires changes in its bundle descriptor, which should import the new libraries used. It should also add a new header, used by the characterization service, to define the link to the real-time characterization.

One should notice the main advantage of this approach: an increase in portability because the parameters of an application may be described in an `xml` file. In this example one change in the

DOI: 10.1002/spe

```
00: import javax.realtime.*;
01: import org.osgi.framework.*;
02: import es.uc3m.it.trosgi.*;
03: import org.w3c.dom.*;
04: public class ExampleRT implements BundleActivator{
05: BundleContext ctx;
06: RealtimeThread rt;
07: int mit=0;


08: public void start(BundleContext context) throws Exception {
09:    rt=new RealtimeThread(){
10:      public void run(){

11:       ServiceReference   c_rf1=   //   Ref   to   the   service
12:      Context.getServiceReference("CharacterizationService");
13:      RealtimeCharacterizationService client1=\\
14:   (RealtimeCharacterizationService) context.getService(c_rf1);
15:      Bundle bundle=context.getBundle();
16:      Document doc=client1.getRealtimeCharacterization(bundle);
17:      NodeListlist nlist=doc.getElementsByTagName("mit");
18:      Node naux=(nlist.item(0));
19:      naux=naux.getFirstChild();
20:      mit=Integer.parseInt(naux.getNodeValue());
21:      RelativeTime rmit= new RelativeTime(mit=mit/1000,0);
22:         try{
23:            do{ System.out.println("RT hello");
24:              RealtimeThread.sleep(rmit); //10 ms
25:          }while (true);
26:         }catch(Exception e){
27:            System.out.println("STOP called: bye, bye");}
28:       };
29:       rt.start(); //Launch the real-time thread
30:  }
31: public void stop(BundleContext context) throws Exception {
22:       rt.interrupt(); //Launch an exception
33:  }
34:.}
```

Listing 3. Reading the MIT parameter from the descriptor

```
00: Bundle-Name: Hello RT Threads
01: Bundle-SymbolicName:
es.uc3m.it.drequiem.trosgi.RTthreads
02: Bundle-Description: A Hello World bundle for RTSJ
03: Bundle-ManifestVersion: 2
04: Bundle-Version: 1.0.0
05: Bundle-ResourceModel: resourcesmodel.xml
05: Bundle-Activator: ExampleRT
06: Import-Package: org.osgi.framework,
javax.realtime, es.uc3m.it.trosgi, org.w3c.dom
```

Listing 4. Bundle descriptor for Listing 2 (updated)

```
00: <!--Bundle-Name: Hello RT Threads ->
01: <schedulable name="RTHello">
02:    <mit>10000</mit>
03: </schedulable>
```

Listing 5. Example of XML descriptor

DOI: 10.1002/spe

mit of an application (e.g., from 10,000 to 50) does not require a Java code recompilation; a change in the xml file is enough.

This approach enables third-party bundle integrators, which may modify the descriptor by using tools and add the bundle to their repositories. For instance, the bundle provider may define an initial period that may be modified later before being available in the OSGi platform by the bundle integrator.

5.3. Level 2 integration

The third level (software stack in Figure 5) adds three new services to the previous stack: an admission controller (trosgi.scheduler), a failure detection and recover mechanism (trosgi.recover), and a compositor service (trosgi.compositor). The first service carries out simple feasibility tests on a set of bundles. The second detects bundle failures and enables recovery policies. The third, which is based on Ref. [33], allows the definition of complex constraints.

5.3.1. Scheduler service programming perspective. The code included in Listings 6 and 7 shows how to use the scheduler service. As in the characterization service, the example assumes that there is a service implementation registered as 'SchedulerService' in the framework implementing the BundleSchedulerService interface (see its API definition in Section 6).

As in the previous example, before using the service, the client code has to get the reference from the framework (Lines 11–14 in Listing 6). After this initialization, the service may be invoked using its standard API, which contains a setIfFeassible method. This method carries out admission control on the bundle. If the schedulable entities of the bundle cannot be added to system, it returns false. The example uses this information to decide if the sporadic printout starts or not.

Complex bundles, that is, bundles where the logic of the application is not as simple as in the example, should opt for separating configuration issues from the logic implementation. For the given example, the code in lines 21–23 would be placed in an independent Runnable class, whereas lines 11–14 would be hidden in a generic class (e.g., SpecificBundleActivator) extending the current bundle activator.

To complete the example, Listing 7 shows a piece of code that corresponds to the information used by the scheduler service to access the parameterization of the bundle.

To parameterize the service, the information stored in the xml file has to describe the schedulable units used in the bundle. In this case, the bundle describes the sporadic printout release patterns and its priority scheduling parameters. These parameters are accessed from the scheduling service to decide if the task could be added to the tasks currently running on the service.

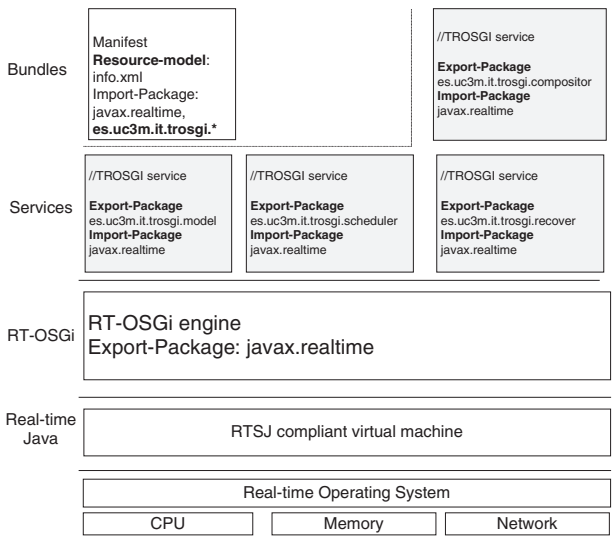

Figure 5. Software stack for the Level 2 integration in TROSGI.

DOI: 10.1002/spe

```
00: import javax.realtime.*;
01: import org.osgi.framework.*;
02: import es.uc3m.it.trosgi.*;
03: import org.w3c.dom.*;
04: public class ExampleRT implements BundleActivator{
05: BundleContext ctx;
06: RealtimeThread rt;
07: int mit=0;

08: public void start(BundleContext context) throws Exception {
09:    rt=new RealtimeThread(){
10:      public void run(){

11:  ServiceReference c_rf1= // Ref to the service
12:      Context.getServiceReference("SchedulerService");
13:      BundleSchedulerService client1=\\
14:      (BundleSchedulerService) context.getService(c_rf1);
15:      Bundle bundle=context.getBundle();
16:      boolean accepted=client1.setIfFeassible(bundle);
17:      if (!accepted){
18:            return;
19:         }
20:      Thread.getCurrentThread().setPriority(50); //Priority
21:         try{
22:            do{ System.out.println("RT hello");
23:              RealtimeThread.sleep(new RelativeTime(10,0)); //10 ms
24:          }while (true);
25:         }catch(Exception e){
26:            System.out.println("STOP called: bye, bye");}
27:       };
28:       rt.start(); //Launch the real-time thread
29:  }
30: public void stop(BundleContext context) throws Exception {
31:       rt.interrupt(); //Launch an exception
32:  }
33: }
```

Listing 6. Using the scheduler service

```
00:<!--Bundle-Name: Hello RT Threads ->
01: <bundlescheduler>
02:  <schedulable name="RTHello">
03:   <releaseparameters>
04:    <sporadic>
05:    <mit>10000</mit>
06:    <cost>1</cost>
07:    </sporadic>
08:  </releaseparameters>
08:  <scheduling>
09:    <priority>50</priority>
10:  </scheduling>
11:  </schedulable>
12: </bundlescheduler>
```

Listing 7. Parametrizing the scheduler service

The main advantage provided to the programmer is the possibility of ensuring that the bundle meets its desired real-time performance. Another advantage is that the application designer does not require a deep understanding of the scheduler algorithm used. Its implementation is hidden from the definition of the service using SOA tenets.

DOI: 10.1002/spe

5.3.2. Recovery service programming. The recovery service defines a fault detection and recovery mechanism working at bundle level. It allows subscribing the bundle to the service (as made in line 16 of Listing 8), which takes fault detection and recovery actions. The name of the policy used by the bundle is described in `xml` format and it is accessed by the service for service configuration.

The example shown in Listing 9 corresponds to a service where the service should check if the bundle is running every 10 ms. The `xml` file also binds the recovery cost: 200 μs. As in the previous cases, the resource descriptor is packed in the bundle `jar` file.

```
00: import javax.realtime.*;
01: import org.osgi.framework.*;
02: import es.uc3m.it.trosgi.*;
03: import org.w3c.dom.*;
04: public class ExampleRT implements BundleActivator{
05: BundleContext ctx;
06: RealtimeThread rt;
07: int mit=0


08: public void start(BundleContext context) throws Exception {
09:    rt=new RealtimeThread(){
10:      public void run(){

11:  ServiceReference c_rf1= // Ref to the service
12:      Context.getServiceReference("Recover");
13:      BundleRecoverService client1=\\
14:      (BundleSchedulerService) context.getService(c_rf1);
15:      Bundle bundle=context.getBundle();
16:      boolean accepted=client1.addBundle(bundle);
17:      if (!accepted){
18:             System.out.pritln("UNRECOVERABLE BUNDLE");
19:         }
20:         try{
21:            do{ System.out.println("RT hello");
22:              RealtimeThread.sleep(new RelativeTime(10,0)); //10 ms
23:          }while (true);
24:         }catch(Exception e){
25:            System.out.println("STOP called: bye, bye");}
26:       };
27:       rt.start(); //Launch the real-time thread
28:  }
29: public void stop(BundleContext context) throws Exception {
30:       rt.interrupt(); //Launch an exception
31:  }
32: }
```

Listing 8. Using the recovery service

```
00: <!-- Bundle-Name: Hello RT Threads -->
01: <bundlerecover>
02:   <policy name="restart">
03:      <mit>100000</mit>
04:      <cmax>200</cmax>
05:      <deadline>100000</deadline>
04:   </policy>
05: </bundlerecover>
```

Listing 9. Defining a restart policy in the descriptor

DOI: 10.1002/spe

5.3.3. Compositor service programming. The last service is the compositor service. It helps to develop complex interaction models, which can appear over time, and allow defining multiconstrained models. A clear example are systems where each task may have multiple costs (e.g., $C = (C_i, C_{2,}C_3) \quad | \; C_i \leqslant C_{i+1}$), which may be controlled from the system (i.e., the system may introduce degradation to meet deadlines).

Another example is an application that has different behaviors over time, depending on its internal state (called modes in the literature). For instance, in Listing 10 the application thread has two states:

- One running at priority 48 with a maximum cost of 2 μs, and a minimum sleep of 5 ms.
- Another running at priority 50, with a maximum cost of 1 μs and a minimum sleep of 10 ms.

By using the compositor service (Listings 10 and 11) the application may define its requirements with an alternative constraint. The constraint is checked by the underlying platform when the bundle

```
00: import javax.realtime.*;
01: import org.osgi.framework.*;
02: import es.uc3m.it.trosgi.*;
03: import org.w3c.dom.*;
04: public class ExampleRT implements BundleActivator{
05: BundleContext ctx;
06: RealtimeThread rt;
07: int mode=0

08: public void start(BundleContext context) throws Exception {
09:    rt=new RealtimeThread(){
10:      public void run(){
11:         ServiceReference     c_rf1=     //     Ref     to     the     service
12:      Context.getServiceReference("Compositor");
13:      BundleRecoverService client1=\\
14:      (BundleSchedulerService) context.getService(c_rf1);
15:      Bundle bundle=context.getBundle();
16:      boolean accepted=client1.addComposedBundle(bundle);
17:      if (!accepted){
18:             System.out.println("BUNDLE NOT COMPOSABLE");
19:              return;
20:         }
21:      Thread.getCurrentThread().setPriority(50); //Priority
22:         try{
23:            do{
24:                if (mode==0){
25:                  mode=1;
26:                  System.out.println("RT hello");
27:                  Thread.getCurrentThread().setPriority(48);
28:                  RealtimeThread.sleep(new RelativeTime(10,0)); //10 ms
29:                  else{
30:                  mode=0;
31:                  System.out.println("RT hello World! ");
32:                  Thread.getCurrentThread().setPriority(50);
33:                  RealtimeThread.sleep(new RelativeTime(5,0)); /*5 ms*/ }
34:          }while (true);
35:         }catch(Exception e){
36:            System.out.println("STOP called: bye, bye");}
37:       };
38:       rt.start(); //Launch the real-time thread
39:  }
40: public void stop(BundleContext context) throws Exception {
41:       rt.interrupt(); //Launch an exception
42:  }
43: }
```

Listing 10. Using the composed service

DOI: 10.1002/spe

```
00: <!-- Bundle-Name: Hello RT Threads -->
01:   <bundlecompositor>
02:    <alt type="all_required">
03:       <block>
04:          <schedulable name="RTHello">
03:            <releaseparameters>
04:               <sporadic>
05:                 <mit>10000</mit>
06:                 <cost>1</cost>
07:               </sporadic>
08:            </releaseparameters>
08:            <scheduling>
09:            <priority>50</priority>
10:            </scheduling>
11:          </schedulable>
12:      </block>
13:      <block>
14:          <schedulable name="RTHello">
15:            <releaseparameters>
16:               <sporadic>
17:                 <mit>5000</mit>
18:                 <cost>2</cost>
19:               </sporadic>
20:            </releaseparameters>
21:            <scheduling>
22:            <priority>48</priority>
23:            </scheduling>
24:      </block>
25:     </alt>
26:   </bundlecompositor>
```

Listing 11. Defining the bundle information for the compositor service

is added to the compositor service according to the service. In this case, the checks required to add a bundle have to check the feasibility of this behavior.

Summarizing, the integration proposed is made at three levels that take into account implementation efforts and impact on the programmer. Level 0 is just access to the underlying RTSJ API, Level 1 refers to real-time resource characterization and Level 2 to scheduling facilities, fault recovery mechanisms, and bundle compositors.

Section 6 introduces the API for the services informally introduced in the examples shown in this section.

6. AN API FOR TROSGI

This section describes an API for the services introduced in the previous section. All the services described in the previous section are created on the `es.uc3m.it.trosgi` package. The API is complemented with an `xml` description of the services currently implemented in the current prototype (see Section 7).

6.1. Service definition

All services have a common template that allows attaching event handlers (Listing 12) to every real-time service. The meaning of the events and the condition that raise them is application-dependent and should be taken into consideration in the application and the implementation of the service.

The real-time characterization service (Listing 13) has two methods: one to read the DOM tree (shown in the previous section) and another to write it. In a case where the DOM tree cannot be accessed, it calls all subscribed `handle` methods to notify an exception.

The real-time scheduler service (Listing 14) has two methods: one to add a bundle (shown in the previous section) and another to remove it. In case the bundle cannot be added or a new bundle is accepted in the service, it calls the event handler method (`eh`) of the subscribed clients.

DOI: 10.1002/spe

```
00: public interface RealTimeService{
01:   boolean addHandler(EventHandler eh);
02:   boolean removeHandler(EventHandler eh);
03: }
```

Listing 12. Shared API among all real-time services

```
00: import org.osgi.framework.*;
01: public interface RealtimeCharacterizationService
02:                 extends RealTimeService{
03:   public Document
04:          getRealtimeCharacterization(Bundle b);
05:   public boolean setRealtimeCharacterization(
06:                Bundle bnd, Document doc);
07:}
```

Listing 13. Real-time characterization service API

```
00: import org.osgi.framework.*;
01: import org.w3c.dom.*;
02: public interface BundleSchedulerService
03:                 extends RealTimeService{
04:    public boolean setIfFeassible(Bundle bnd);
05:    public boolean removeFromScheduler(Bundle bnd);
06: }
```

Listing 14. Bundle scheduler service

```
00: import org.osgi.framework.*;
01: public interface BundleRecoverService
03:                  extends RealTimeService{
04:    public boolean addBundle(Bundle bnd);
05:    public boolean removeBundle(Bundle bnd);
}
```

Listing 15. Bundle recovery service

```
00: import org.osgi.framework.*;
01: public interface BundleCompositorService
03:                 extends RealTimeService{
03:    public boolean addComposedBundle(Bundle bnd);
04:    public boolean removeComposedBundle(Bundle bnd);
}
```

Listing 16. Bundle compositor service

The recovery service (Listing 15) has two methods: one to add a bundle to the service and another to remove it. In case it cannot add/remove the bundle to/from the service, it calls the corresponding event handler previously subscribed.

As in the previous service, the bundle compositor service has a method to add and another to remove bundles from the feasibility (Listing 16). Each time a bundle is added or removed, it raises an event, which is transferred to the subscribed event handlers. Enhanced implementations may use the event mechanism to notify runtime changes to the subscribed bundles. The events generated and their meaning (i.e., actions taken by the bundle) is an application-dependent characteristic.

6.2. Extensible markup language document type definition

All services require parameterization in an `xml` format. The DTD included in Listing 17 defines a valid structure for the descriptor passed to a service. Any service parameter (`<bundlescheduler>`, `<bundlerecover>`, `<bundlecompositor>`) has a `<resourcemodel>` as a direct parent.

DOI: 10.1002/spe

```
00: <!DOCTYPE resourcemodel[
01: <!ELEMENT resourcemodel(bundlescheduler|
02:       bundlerecover|bundlecompositor)
03: <!ELEMENT bundlescheduler(schedulable*)>
04:  <!ELEMENT schedulable (releaseparameters?,scheduling?)>
05:   <!ATTLIST alt name (#PCDATA)>
06:   <!ELEMENT releaseparameters (periodic|sporadic
07:                               |aperiodic)>
08:    <!ELEMENT periodic (start?,period?,cost?,deadline?)>
09:     <!ELEMENT sporadic (start?,mit?,cost?,deadline?)>
10:     <!ELEMENT aperiodic>
11:     <!ELEMENT start (#PCDATA)>
12:     <!ELEMENT period (#PCDATA)>
13:     <!ELEMENT cost (#PCDATA)>
14:     <!ELEMENT deadline (#PCDATA)>
15:     <!ELEMENT mit (#PCDATA)>
16: <!ELEMENT scheduling (priority?)>
17:    <!ELEMENT priority (#PCDATA)>
18: <!ELEMENT bundlerecover (policy+)>
19: <!ELEMENT policy (mit?, cost?,deadline?,priority?)>
20:   <!ATTLIST policy name #PCDATA>
21: <!ELEMENT bundlecompositor (alt*)>
22:  <!ELEMENT alt (block+)>
23:   <!ATTLIST alt type (#PCDATA)>
24:  <!ELEMENT block (schedulable*)>
25: ]>
```

Listing 17. Unified DTD corresponding to all services

The scheduler is defined as a list of independent schedulables. Each schedulable may have a release pattern and scheduling parameters. Following the RTSJ description model, each release pattern may be periodic, aperiodic, and sporadic. Periodic tasks may be specified with its release time (start), period, cost, and deadline. The sporadic release pattern substitutes the period by a minimum interarrival pattern.

The bundle recovery service may define a list of policies. Each policy is defined as a sporadic task. In addition, the policy may receive a `name`, which may be used to setup the recovery service.

Lastly, the bundle compositor mechanism is very similar to the scheduler service. Both describe the bundle as schedulable entities with release and scheduling parameters. The main difference between this service and the scheduling service is that it may offer alternatives and blocks. Each alternative requires at least one block, which is the entity that holds a list of schedulables.

This section presented a basic API subsystem for adding the services described in the previous section in OSGi, as new platform services. Section 7 shows implementation details for each specific service implementation and their corresponding performance patterns.

7. EMPIRICAL EVALUATION OF TROSGI

This section introduces the patterns obtained on a reference implementation of the architecture described in the previous sections. For each level feature, the section describes performance patterns stemming from each mechanism in terms of cost and obtained predictability.

A Level 2 implementation was carried out to evaluate the performance that may be expected from a real-time OSGi engine. In this implementation, the authors used Sun's JRTS [16], which offers a modified virtual machine with support for real-time RTSJ-compliant applications (see Figure 6). The underlying OSGi software used for the implementation was Knopflerfish [34]. Benchmarks run on a personal computer (1.5 GHz) with a standard real-time Linux kernel (2.6.28-3-rt) on which the TROSGi implementation was deployed.

DOI: 10.1002/spe

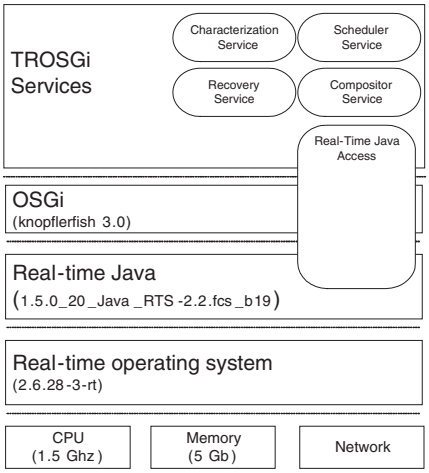

Figure 6. Software stack used in the empirical evaluation of TROSGi.

On this architecture there is one service implementation for each service:

- `Impl.RealtimeCharacterizationServiceImpl`.
  The basic implementation of this service allows access to the `xml` descriptor associated with the bundle. The basic implementation does not allow overriding the basic configuration file.
- `Impl.BundleSchedulerServiceImpl`.
  The basic implementation of this service allows adding bundles to the set of tasks running in the system.

  The basic theory implemented is the efficient bound described in [35], previously used in [23, 36]:

$$\forall i, R_i^{\mathrm{ub}} = \frac{C_i + \sum_{j=1}^{i-1} C_j(1-U_j)}{1-\sum_{j=1}^{i-1} U_j} \leqslant D_i \tag{1}$$

Equation (1) assumes n tasks, which are ordered with increasing priorities. Each task (τ_i) has a maximum utilization ($U_i = C_i/T_i$) and a maximum cost (C_i). For each task, the equation allows to calculate an upper bound response time $\left(R_i^{\mathrm{ub}}\right)$, which should be lesser than the desired deadline (D_i).

From these values, it is possible to decide on the schedulability of a set of tasks in a $O(n)$ time, with n the number of tasks in the system. One way to solve the equations is to start with $i = 1$ and iteratively check the n inequations required to check that all tasks meet their deadlines. An efficient implementation may use the calculations carried out for the i^{th} schedulable in the $(i+1)^{th}$ schedulable.

The main difference between the implementation carried out and [23, 36] is that [23, 36] include server technology to integrate the concept of application. [23, 36] also change the priorities assigned to the bundle, while our approach maintains the scheduling parameters declared by the client, which simplifies the admission control carried out.

- `Impl.BundleRecoverServiceImpl`.

The default recovery module allows adding and removing bundles to or from the system. The default service supports a pooling mechanism that detects if a bundle (subscribed to the system) is active or not. In a case where the bundle is not active, it is activated.

The implementation carried out consists of one-to-one mapping between the subscribed bundles and a substitute thread in the service. The thread is allocated when the bundle is accepted in the service. The period, cost, and other scheduling parameters of the thread are taken from the descriptor. Periodically, the thread checks if its associated bundle is active or not; if it is not active, the thread

DOI: 10.1002/spe

activates the bundle by using the OSGi framework API. Likewise, when the bundle is unsubscribed, the thread is released.

- `Impl.BundleCompositorServiceImpl`.
  The default bundle compositor uses the default bundle scheduling implementation to check every combination. By default, the implementation explores all the combinations and checks the feasibility of the system for each combination. Our implementation is able to validate applications with several constraints that must be fulfilled.

Regarding the prototype implementation, an eventual implementation note is written concerning life-cycle management and the new services introduced by TROSGi. There three sources for life-cycle operations in TROSGi: (i) the system console, (ii) another bundle, and (iii) the recovery service. In TROSGi, when a life-cycle operation is launched from the console, the priority of the operation is lower than the real-time priorities of the system. If it is launched from another bundle, the operation inherits the priority of the calling thread, as is made in RTSJ when calling a method in the API. Lastly, the recovery service allows setting up the priority of the recovery technique by configuring its descriptor.

7.1. Level 0 integration: real-time performance in OSGi

Level 0 provides OSGi with real-time performance (access to the `javax.realtime` package) so that the first experiment compares the performance given by a traditional OSGi, which lacks real-time threads, against the new OSGi bundles that may use real-time threads. The goal of this experiment is to show the performance delivered by having access to the real-time Java API versus current OSGi implementations (which lack such type of support).

The experiment considers two activities (i.e., two threads). One requires 0.5 s every 0.7 s and another that varies the CPU consumption from 3 s to 1 ms in each invocation. Figure 7 shows the response time of the second task for different works. In this figure work = 1000 refers to a response time by 3 s. The relationship between the time demanded and the work axis follows an x^3 function.

The figure shows the plain OSGi performance (labeled Non RT) where all tasks share the same priority and the real-time case (RT). For the real-time case, two situations are displayed: in the first case (RT(high_prio)) the displayed task has a higher priority than the 0.5 s task, so that it does not have to suffer its interference. When the priority of the application is lower than the priority of the second real-time task, the response-time increases above the nonreal-time case.

7.2. Level 1 integration: resource characterization service performance

One metric very important for the characterization service is the time required to create the DOM representation allocated in memory from the file descriptor. Because this service is also used by other services allocated in the platform, the evaluation case varies the number of the schedulable

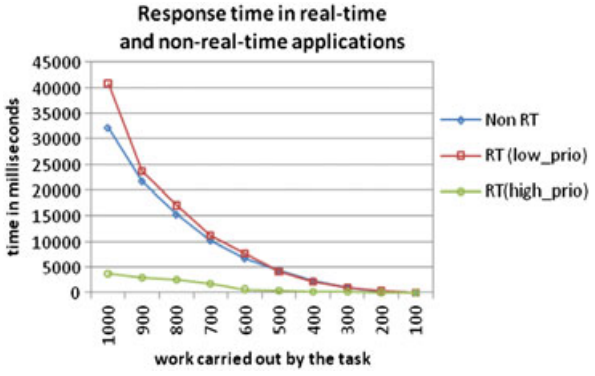

Figure 7. Nonreal-time against real-time performance (1.5 GHz).

DOI: 10.1002/spe

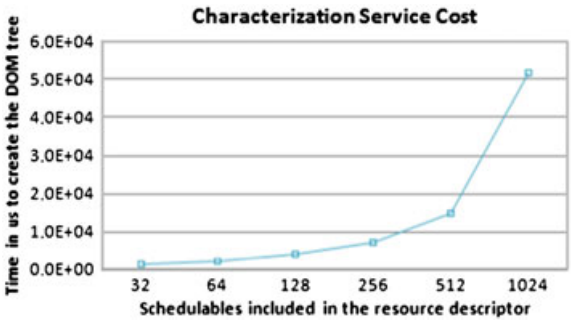

Figure 8. Time consumed creating the DOM tree from the bundle descriptor.

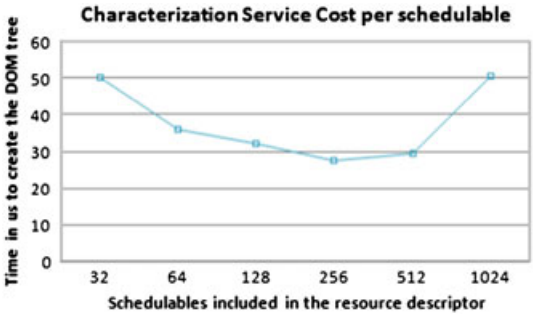

Figure 9. Average time consumed per schedulable in the descriptor.

entities included in the file. All schedulables are defined as periodic with a cost, a period, a deadline, and a priority (following the grammar shown in Section 6.2).

The results (Figure 8) showed that the cost grows with the number of tasks in the system as expected. The results obtained are in the 1 ms (with 32 schedulables) and 51 ms (with 1024 schedulables) range.

In the empirical evaluation of the characterization service (Figure 9), the cost per schedulable varies from 50 μs for 32 schedulables and 1024 schedulables to a minimum of 28 μs for 256 schedulables. Ideally, from 256 schedulables the graph move towards an asymptotic value around 27 μs per sample. The difference between the ideal and the measured case is in the implementation of the library that creates the DOM tree, which cannot maintain the linear cost.

The main conclusion obtained after the evaluation is that the service has a strong dependency on the mechanisms used to access the file descriptor and to process the `xml` file.

7.3. Level 2 integration: scheduler service implementation performance

7.3.1. Basic behavior. By using the same benchmark (i.e., 32–1024 schedulables) the authors evaluated the performance of the scheduler service. In this case, the worst case happens when all schedulable entities are stored in one bundle and there are no other bundles registered in the service. For this configuration, the overhead of having to read the `xml` file goes from 2.1 ms for 32 schedulables to 53 ms with 1024 schedulables (see Figure 10).

7.3.2. Scheduling algorithm cost and XML processing overhead. Figure 10 shows the cost of the scheduling algorithm in the same scenario shown in Figure 11. The figure shows that the cost increases with the number of schedulable entities in the system. This cost is much more reduced than in Figure 11 because of the overhead of having to read the `xml` file.

Figure 12 shows the overhead introduced by parsing the `xml` file versus the scheduling algorithm cost. The empirical results show that the cost of the feasibility test decreases as the number of

DOI: 10.1002/spe

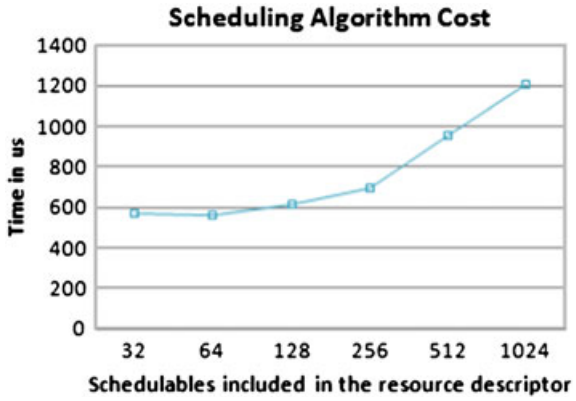

Figure 10. Time required carry out the feasibility test.

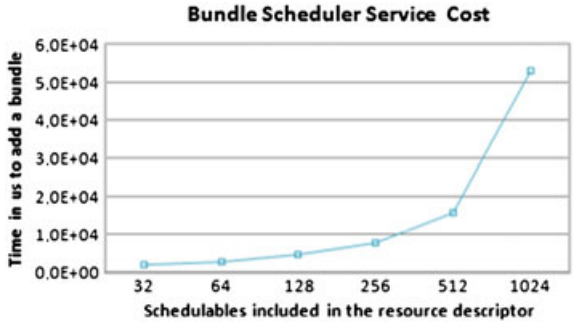

Figure 11. Time consumed to read the configuration and to carry out the feasibility test.

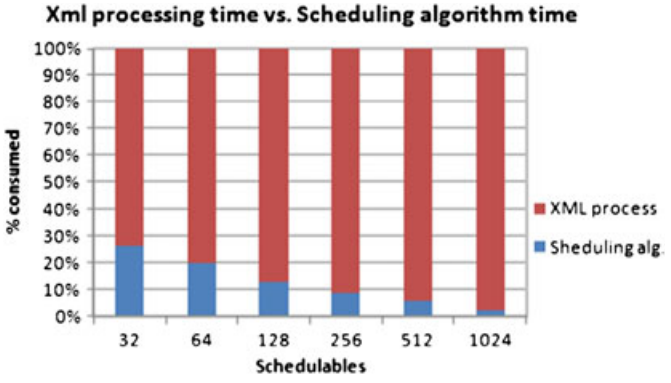

Figure 12. Cost because of xml file reading and scheduling algorithm.

schedulable entities in the file increases. The empirical overhead because of `xml` processing ranges from 75% for 32 schedulables to 97% for 1024 schedulables.

7.3.3. Cost with other bundles allocated in the service. The next set of experiments analyzes the cost of the service when there are other bundles in the system, which have a certain amount of schedulable entities running in the system. Three different cases are analyzed: (i) 25%, which means that the bundle wants to insert 25% new schedulable tasks; (ii) 50%, which means that half of the tasks are already in the system; and (iii) 100%, which means that the system is empty (as in the experiments described in Figures 11 and 10).

DOI: 10.1002/spe

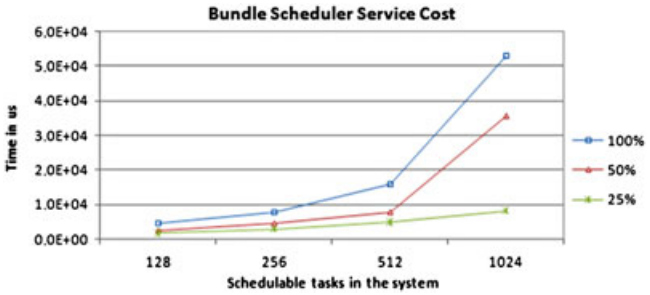

Figure 13. Bundle scheduler service cost with tasks already allocated in the system.

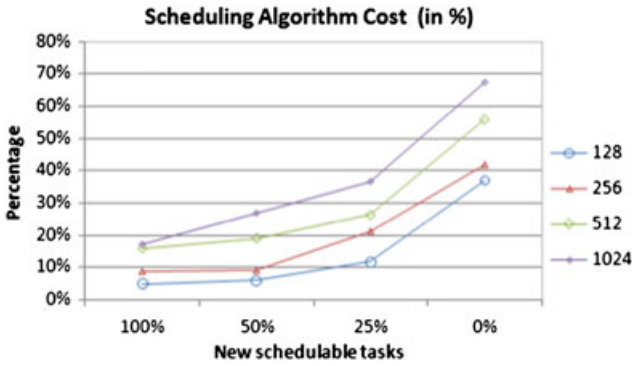

Figure 14. Overhead introduced by the scheduling algorithm in the feasibility test.

The results obtained showed that having tasks running in the system is advantageous, in terms of cost. For instance (Figure 13), with 1024 schedulables the difference between having to allocate all of them is to change the feasibility algorithm cost from 53 ms to less than 8 ms. This reduction in time is very significant in applications where tasks are distributed in different bundles, which may reduce their worst-case admission costs.

One consequence of this reduced admission cost is that the overhead introduced by the scheduling algorithm increases. As Figure 14 shows, the cost of executing the scheduling algorithm when the descriptor contains a moderate number of schedulables is more significant than in the previous example. This is mainly due to the reduction in the time required to process a shorter `xml` file of the new bundle.

The main conclusion stemmed from the empirical evaluation of this service is the high overhead because of the `xml` processing introduced in the system. Future optimizations in this service should try to develop optimized representation formats (e.g., binary or other formats that speed up data access).

7.4. Level 2 integration: detection and recovery service performance

For the detection and recovery service, the empirical evaluation considered two types of experiments. The first set is directed to determine the reconfiguration cost. The reconfiguration requires stopping and starting the subscribed bundles that are not running. The second part of the experiments analyzes the overhead introduced by the service (in utilization terms) and its relationship with the operational frequency of the service.

7.4.1. Detection and recovery cost. The service models the cost of a failure as the cost of stopping (carried out by the application) and starting a bundle which is stopped. The cost of these two strategies is in Table V. In addition the table provides the worst-case recovery cost and a soft bound for soft real-time applications, which require meeting 99% of their deadlines. They show that there is

DOI: 10.1002/spe

Table V. Recovery hard and soft cost depending on the policy (10,000 samples per case) for a simple bundle.

Approach	Worst case cost (ms)	Soft (99%)case cost (ms)
Including installation cost (full installation)	257	157
Without installation cost (start only)	177	137

an important difference depending on the technique used to recover the application (full installation vs start only) and the type of guarantee offered (hard vs soft).

7.4.2. Cost in utilization because of recovery frequencies. The range of frequencies in which the service may operate was evaluated in the four cases described in Table VI and it is based on the frequencies defined in [37] and [38]. The operational frequencies evaluated range from 100 Hz to 10 mHz (Figures 15 and 16). In all cases and in the described platform, the system does not have enough resources to carry out activations with frequencies higher than 6 Hz. However, it offers a good performance (utilization required per bundle subscribed <1%) when the maximum recovery frequency is 20 mHz.

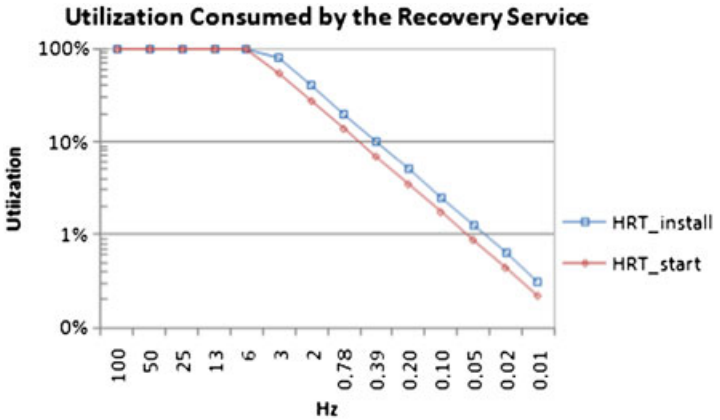

Figure 15. Utilization consumed by the recovery service with hard policies (1 bundle subscribed).

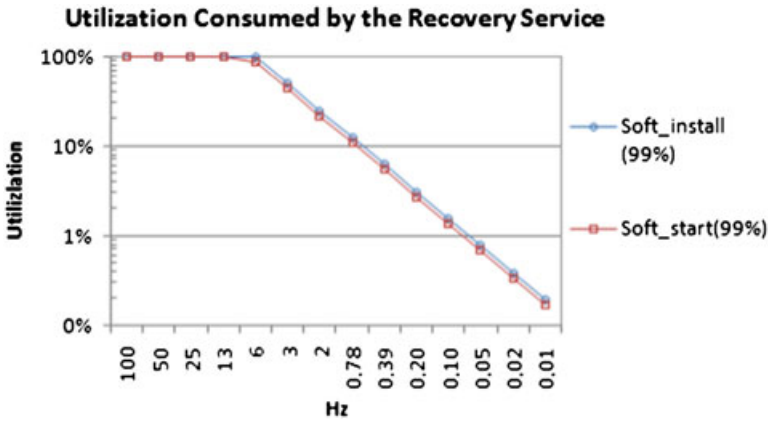

Figure 16. Utilization consumed by the recovery service with soft policies (1 bundle subscribed).

DOI: 10.1002/spe

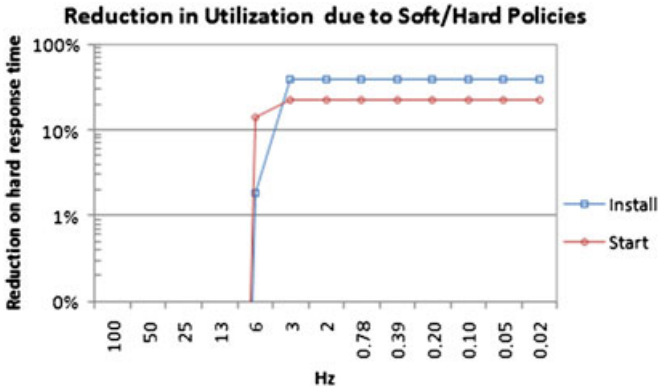

Figure 17. Utilization ratio reduction in soft versus hard policies.

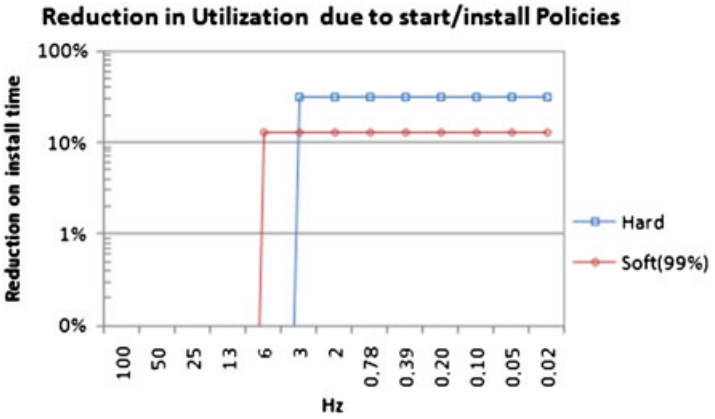

Figure 18. Utilization ratio reduction in start versus installed policies.

7.4.3. Soft versus hard real-time costs in operational frequencies. Analyzing the same operational range Figure 17 shows the ratio between the soft and hard policies in the full install (install) and start only strategies. The reduction in the cost because of the possibility of using a soft policy is 40% in the case of having to install the bundle from the local hard disk and 25% when it is only started.

7.4.4. Installed modules versus uninstalled modules. The last analysis refers to the difference in overhead because of having to install the bundle from the local hard disk to the case where the bundle is in the installed state. In this case (Figure 18) the reductions offered by having the bundle installed are 15% for the soft bound and 35% for the hard bound.

The main conclusion obtained from the evaluation of the recovery implementation carried out is that applications should carefully choose their minimum period and recovery strategy (full installation vs only start) to prevent the service from consuming all the available bandwidth.

7.5. Level 2 integration: bundle compositor service performance

The last service analyzed is the compositor service. All the experiments quantify the time required to admit a bundle in the system. Because the bundle compositor is (in some sense) an extension of the bundle scheduler service, the evaluation uses the same range of schedulable entities and quantifies the overhead introduced by having to evaluate multiple alternatives.

DOI: 10.1002/spe

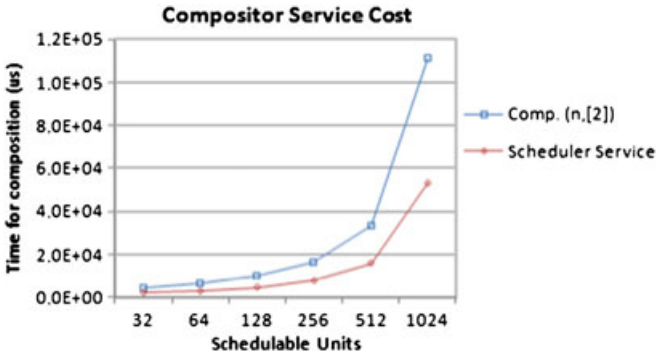

Figure 19. Compositor service cost versus scheduler service cost in a simple case.

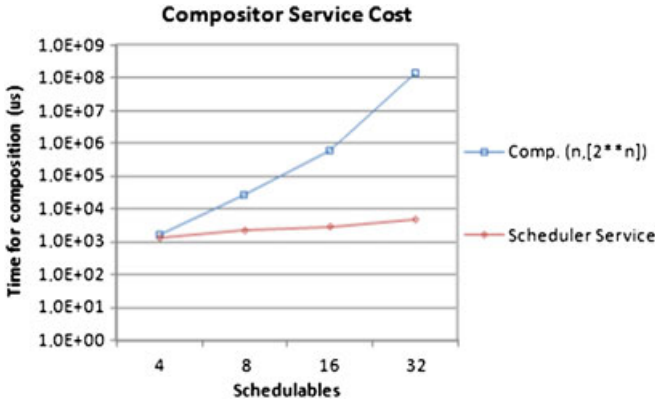

Figure 20. Compositor service cost versus scheduler service with applications with two alternatives per schedulable.

7.5.1. Basic performance pattern in the compositor service. As a general rule, the higher the number of blocks included in the resource file, the higher the number of combinations that have to be explored. Figures 19 and 20 exemplify this behavior in simple applications.

Figure 19 shows a bundle that has two alternatives. Each alternative has n schedulable entities (with n ranging from 32 to 1024) that have real-time constraints that have to be checked by the scheduler. In this example the number of schedulable tests that have to be carried out by the bundle compositor is $2 \cdot n$ with n the number of running schedulable units.

The graph shows the performance of the scheduler service for the same amount of the schedulable entities. For the same amount of schedulable entities, the scheduler service admission control consumes less CPU as expected.

The gap between the two services increases as the number of alternatives increases as exemplified in Figure 20. In the case shown there is an application with a certain number of threads (x-axis) with multiple constraints (2 constraints per each schedulable unit) that have to be evaluated by the compositor service. For this configuration the number of combinations to be checked is by 2^n where n is the number of schedulables in the system. In this case 32 schedulable units produce an increase of 5 orders in magnitude in the cost of the schedulables.

7.5.2. Several simple composed applications cost patterns. The last experiment carried out with the compositor service explores how having a certain amount of tasks already included in the service may reduce the admission cost of the bundle compositor service. To this end, the example described in Figure 21 continues the results included in Figure 20. In both cases, the number of checks (i.e., the number of explored combinations) is the same. The only difference is in the processing of the

DOI: 10.1002/spe

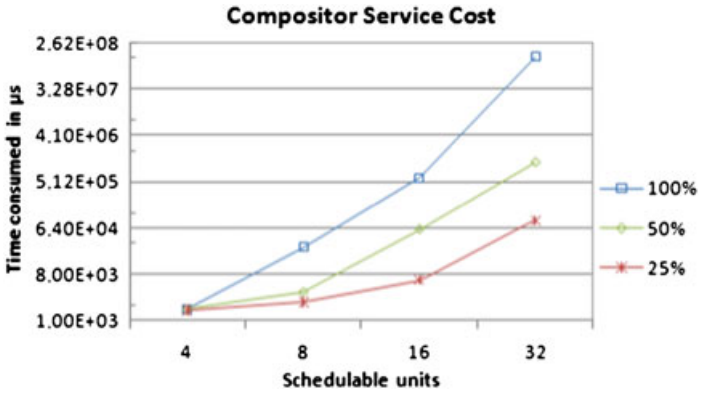

Figure 21. Bundle compositor cost with tasks already allocated in the system.

`xml` file, which only refers to a part of all schedulables in the system. The results show an important reduction when the system contains 75% and 25% new schedulables. The reductions in time are due to the overhead introduced by the `xml` processing and transformation processes, which are alleviated by having a more reduced number of schedulables in each bundle.

The results shown for the compositor service and those previously obtained for networked service-based applications [39] suggest that applications should avoid the state explosion problem by limiting the alternatives that are explored. Another possibility is to have some type of heuristic included in the descriptor that helps to reduce the number of states explored. To this end, the figures of merit (described in [39]) may help reduce the number of combinations to be explored remarkably.

7.6. Main conclusion from the empirical evaluation and future optimizations

The main contribution of this section was to provide initial performance patterns for the real-time OSGi model in terms of performance that may be expected from a reference implementation. These performance patterns showed that real-time performance in applications may help developers to integrate real-time performance in their applications. The results also showed the performance pattern shown by each service implementation. For the characterization service, results establish an empirical relationship between the information stored in the configuration file and the cost of accessing these data. For the scheduler service, the contribution was to study the evolution between the number of schedulables included in the bundle and the cost of performing the acceptance test. For the recovery service, the main contribution is to bind the cost of the recovery strategy proposed. Lastly, for the compositor service implementation, the evaluation established a parallelism between the performance of the scheduler service and the compositor service.

All results regarding performance could improve changing the representation format. A common approach to reduce the overhead of the scheduling, compositor, recovery, and characterization services is to introduce enhanced representation formats, which do not require the complex processing of an `xml` file. For the analyzed cases, the main source of overhead is the access to the configuration file stored in the bundle. Nevertheless, it should be noticed that the use of a standard `xml` format reduces the implementation (e.g., tree navigation) efforts remarkably. Therefore, future format optimizations should evaluate the efficient format versus standard format trade-off carefully.

8. RELATED APPROACHES

In the state-of-the-art two different sets of work are relevant. The first set refers to generic enhancements to OSGi infrastructures and the second to specific pieces of work related to real-time OSGi.

DOI: 10.1002/spe

The first set of approaches may be used to enhance the real-time infrastructure defined in this article. The work described in [40] describes a (TV) middleware, which may be integrated in the architecture described in this paper. Additional support described for Zigbee [41] and universal plug-and-play [42] in an OSGi engine is also interesting from the perspective of an enhanced distributed real-time Java architecture with access to multiple networks.

Remote communications in the home using OSGi as deployment infrastructure is addressed in remote OSGi (R-OSGi) [43]. The main difference between the model proposed in this article and R-OSGi is the real-time performance introduced by the real-time Java infrastructure. R-OSGi may profit from the techniques currently included in our work to improve its predictability.

Another relevant work that refers to another piece of functionality included in current infrastructures is described in [44]. The authors described a composition service for OSGi, which is similar to the composer described in our architecture (Level 2). Once more, the main difference between both efforts is in terms of goals. In service composition QoS is a primary goal, while in the effort defined in [44] the goal is more at system interoperability and service detection.

Lastly, there are other pieces of work that target predictability in OSGi. Two of them deal with integration of low-level RTSJ techniques like temporal isolation [23] and admission control [24] in OSGi. From the point-of-view of the model described in this paper, the architecture proposed may be used to deploy these two services as Level 2 implementation services in the bundle scheduler service.

In addition, there is a real-time adaptive framework, which combines C and Java programming to offer real-time performance [8]. There are many commonalities among our Level 1 bundle descriptor and their component model, which is also described in `xml` files. The main difference between them is that the adaptive component model is implemented in C (for real-time applications) and Java (for general-purpose) while our approach is 100% pure Java.

The last piece of work [45] refers to the use of resource reservation protocols to enhance the predictability of distributed applications that use OSGi. The technique described is based on Resource Reservation Protocol (RSVP) and may be integrated in the distributed framework proposed for the Level 2 integration, as a new type of network.

9. CONCLUSION AND FUTURE WORK

This paper introduced an architecture for enhancing OSGi with real-time Java support. The proposed architecture has three integration levels, which put different requirements on the OSGi framework. The paper also proposed a set of services for real-time OSGi, namely the real-time characterization service, the scheduler service, the recovery service, the real-time characterization service and the compositor service. Our empirical evidence targeted to the performance delivered by each level. The basic Level 0 integration results showed that having fine control on resources may improve the response-time of applications remarkably. Results obtained for Level 1 refer to the cost of accessing to the description of the contents of the bundle and analyzed the overhead introduced by the abstraction. Lastly, the results for Level 2 showed the overhead introduced by each service implementation.

Our on-going work is focused on the integration of DRTSJ with the framework proposed in this article. The basic implementations defined for each service do not consider distribution, that is, they are centralized. Therefore, our efforts are going to be directed to produce distributed versions for these services taking into account the remote services introduced by the OSGi framework since version 4.2. The authors also consider the integration of dynamic deployment [39], enhanced real-time reconfiguration policies [46, 47] proposed in the iLAND project, and enhanced distribution Java support [22, 28–30, 48–55]) as new service implementations for OSGi.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the many valuable contributions made by the anonymous reviewers of the article. This work has been partly funded by the iLAND project (ARTEMIS-JU 100026) funded by the ARTEMIS JTU and the Spanish Ministry of Industry, Commerce, and Tourism. Also, this work has been partly funded by the ARTISTDesign NoE (IST-2007-214373) of the 7th EU Framework Programme and by the Spanish national project ‘DDS Gateway for Web Services’ (TSI-020501-2008-159) and

DOI: 10.1002/spe

by REM4VSS: Desarrollo de middleware para la Reconfiguración en Tiempo real de Sistemas Distribuidos de Video-Vigilancia (TIN 2011-28339) CICyT.

REFERENCES

1. OSGi Service Platform Core Specification, 2010.
2. Nilsen K. Issues in the design and implementation of real-time java. *Java Developer's Journal* 1996; **1**(1):44.
3. *R*equirements for the real-time extensions for the java platform, September 1999. Available at: http://www.nist.gov/itl/div897/ctg/real-time/rtj-final-draft.pdf.
4. Garcia-Valls M, Basanta-Val P, Estevez-Ayres I. Adaptive real-time video transmission over DDS. *2010 8th IEEE International Conference on Industrial Informatics (INDIN)*, 2010; 130–135.
5. Garcia-Valls M, Estevez-Ayres I, Basanta-Val P. Dynamic priority assignment scheme for contract-based resource management. *International Conference on Computer and Information Technology*, Bradford, UK, 2010; 1987–1994.
6. García-Valls M, Basanta-Val P, Estévez-Ayres I. Real-time reconfiguration in multimedia embedded systems. *IEEE Transactions on Consumer Electronics* 2011; **57**(3):1280–1287. DOI: 10.1109/TCE.2011.6018885.
7. Lin C, Lin C, Hou T. A graph-based approach for automatic service activation and deactivation on the OSGi platform. *IEEE Transactions on Consumer Electronics* 2009; **55**(3):1271–1279.
8. Gui N, Florio VD, Sun H, Blondia C. A framework for adaptive real-time applications: The declarative real-time OSGi component model. *ARM*, Leuven, Belgium, 2008; 35–40.
9. Rajkumar R, Lee I, Sha L, Stankovic J. Cyber-physical systems: The next computing revolution. *2010 47th ACM/IEEE on Design Automation Conference (DAC)*, Anaheim, CA, USA, 2010; 731.
10. Basanta-Val P, Garcia-Valls M, Estevez-Ayres I. Towards a cyber-physical architecture for industrial systems via real-time java technology. *International Conference on Computer and Information Technology*, Bradford, UK, 2010; 2341–2346.
11. Richardson T, Wellings AJ. RT-OSGi: Integrating the OSGi framework with the real-time specification for java. In *Distributed and Embedded Real-Time Java Systems*, Higuera-Toledano MT, Wellings AJ (eds). Springer: Spring street, New York, NY 10013, 2012.
12. Sha L, Abdelzaher TF, Arzen K, Cervin A, Baker TP, Burns A, Buttazzo GC, Caccamo M, Lehoczky JP, Mok AK. Real time scheduling theory: A historical perspective. *Real-time Systems Journal* 2004; **28**(2-3):101–155.
13. Foley S. Tactics for minimal interference from class loading in real-time java ™. *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems*, Vienna, Austria, 2007; 23–32.
14. RTSJ version 1.1, 2005.
15. The real-time specification for java, 2001.
16. Bollella G, Delsart B, Guider R, Lizzi C, Parain F. Mackinac: Making hotspot real-time. *8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, Seattle, USA, 2005; 45–54.
17. IBM WebSphere real-time, 2006.
18. JTime virtual machine. Available at: http://www.timesys.com,2004.
19. The jamaica VM, 2004.
20. Aphelion, 2004.
21. Distributed real-time specification, 2011. Available at: http://www.jcp.org/en/jsr/detail?id=50.
22. Wellings AJ, Clark R, Jensen ED, Wells D. A framework for integrating the real-time specification for java and java's remote method invocation. *Symposium on Object-Oriented Real-Time Distributed Computing*, Washington, DC, USA, 2002; 13–22.
23. Richardson T, Wellings A. An admission control protocol for real-time OSGi. *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, Carmona (Seville)- Spain, 2010; 217–224.
24. Richardson T, Wellings A. J, Dianes JA, Diaz M. Providing temporal isolation in the OSGi framework. *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, Madrid, Spain, 2009; 1–10.
25. Bollella G, Canham T, Carson V, Champlin V, Dvorak D, Giovannoni B, Indictor M, Meyer K, Murray A, Reinholtz K. Programming with non-heap memory in the real time specification for java. *OOPSLA Companion*, Anaheim, CA, USA, 2003; 361–369.
26. Bacon DF, Cheng P, Rajan VT. The metronome: A simpler approach to garbage collection in real-time systems. *OTM Workshops*, Catania, Sicily, Italy, 2003; 466–478.
27. Tofte M, Birkedal L, Elsman M, Hallenberg N. A retrospective on region-based memory management. *High Order and Symbolic Computing* 2004; **17**(3):245–265.
28. Basanta-Val P, Anderson JS. Using real-time java in distributed systems: Problems and solutions. In *Distributed and Embedded Real-Time Java Systems*, Toledano TH, Wellings AJ (eds). Springer: Spring street, New York, NY 10013, 2012.
29. Borg A, Wellings AJ. A real-time RMI framework for the RTSJ. *Proceedings of the 15th Euromicro Conference on Real-Time Systems 2003*, Porto, Portugal, 2003; 238–246.
30. Basanta-Val P, Garcia-Valls M, Estevez-Ayres I. No-heap remote objects for distributed real-time java. *ACM Transactions on Embedded Computing Systems* 2010; **10**(1):1–25.

DOI: 10.1002/spe

31. Basanta-Val P, García-Valls M, Estévez-Ayres I. No heap remote objects: Leaving out garbage collection at the server side. *OTM Workshops*, Larnaca, Chiprus, 2004; 359–370.
32. Basanta-Val P, Garcia-Valls M, Estevez-Ayres I. Towards the integration of scoped memory in distributed real-time java. *ISORC '05: Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, Seattle, USA, 2005; 382–389.
33. García-Valls M, Estévez-Ayres I, Basanta-Val P, Delgado-Kloos C. CoSeRT: A framework for composing service-based real-time applications. *Business Process Management Workshops 2005*, Nancy, France, 2005; 329–341.
34. Open source OSGi service platform. Available at: http://www.knopflerfish.org.
35. Bini E, Nguyen THC, Richard P, Baruah SK. A response-time bound in fixed-priority scheduling with arbitrary deadlines. *IEEE Transactions on Computers* 2009; **58**(2):279.
36. Richardson T, Wellings AJ, Dianes JA, Diaz M. Towards memory management for service-oriented real-time systems. *JTRES '10: Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, Prague, Czech Republic, 2010; 128–137.
37. Basanta-Val P, Garcia-Valls M, Estevez-Ayres I. Non-functional information transmission patterns for distributed real-time java. *Software: Practice and Experience* 2011.
38. Basanta-Val P, Almeida L, Garcia-Valls M, Estevez-Ayres I. Towards a synchronous scheduling service on top of a unicast distributed real-time java. *RTAS '07.13th IEEE Real Time and Embedded Technology and Applications Symposium*, Bellavue, Seattle, US, 2007; 123–132.
39. Estevez-Ayres I, Basanta-Val P, Garcia-Valls M, Fisteus JA, Almeida L. QoS-aware real-time composition algorithms for service-based applications. *IEEE Transactions on Industrial Informatics* 2009; **5**(3):278.
40. De Lucena VF, Filho JEC, Viana NS, Maia OB. A home automation proposal built on the ginga digital TV middleware and the OSGi framework. *IEEE Transactions on Consumer Electronics* 2009; **55**(3):1254–1262.
41. Ha Y. Dynamic integration of zigbee home networks into home gateways using OSGI service registry. *IEEE Transactions on Consumer Electronics* 2009; **55**(2):470–476.
42. Kang D, Kang K, Choi S, Lee J. UPnP AV architectural multimedia system with a home gateway powered by the OSGi platform. *IEEE Transactions on Consumer Electronics* 2005; **51**(1):87–93.
43. Wu J, Huang L, Wang D, Shen F. R-OSGi-based architecture of distributed smart home system. *IEEE Transactions on Consumer Electronics* 2008; **54**(3):1166–1172.
44. Diaz Redondo RP, Vilas AF, Cabrer MR, Pazos Arias JJ, Lopez MR. Enhancing residential gateways: OSGi service composition. *IEEE Transactions on Consumer Electronics* 2007; **53**(1):87–95.
45. Topalis E, Mandalos L, Koubias S, Papadopoulos G, Nikiforakis I. A novel architecture for remote home automation e-services on an OSGi platform via high-speed internet connection ensuring QoS support by using RSVP technology. *IEEE Transactions on Consumer Electronics* 2002; **48**(4):825–833.
46. Garcia-Valls M, Basanta-Val P, Estevez-Ayres I. Supporting service composition and real-time execution through characterization of QoS properties. *Proceeding of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, Waikiki, Honolulu, HI, USA, 2011; 110–117.
47. Garcia-Valls M, Basanta-Val P, Estévez-Ayres I. Supporting service composition and real-time execution throught characterization of QoS properties. *In Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems (SEAMS '11)*, ACM, New York, NY, USA, 2011; 110–117.
48. Basanta-Val P, García-Valls M, Fernandez-Gonzalez J, Estevez-Avres I. Fine tuning of the multiplexing facilities of Java's Remote Method Invocation. *Concurrency and Computation: Practice and Experience* 2011; **23**:1236–1260. DOI: 10.1002/cpe.1701.
49. Basanta-Val P, Garcia-Valls M, Estevez-Ayres I. Using switched-ethernet and Linux TC for distributed real-time java infrastructures. *Proceedings of the IEEE RTAS Work-in-Progress 2010*, 2010.
50. Basanta-Val P, Estevez-Ayres I, Garcia-Valls M, Almeida L. A synchronous scheduling service for distributed real-time java. *IEEE Transactions on Parallel and Distributed Systems* 2010; **21**(4):506–519.
51. Basanta-Val P, Garcia-Valls M, Estevez-Ayres I. Simple asynchronous remote invocations for distributed real-time java. *IEEE Transactions on Industrial Informatics* 2009; **5**(3):289–298.
52. Basanta-Val P, García-Valls M, Estévez-Ayres I, Fernandez-Gonzalez J. Integrating multiplexing facilities in the set of JRMP subprotocols. *IEEE (Revista IEEE America Latina) Latin America Transactions* 2009; **7**(1):107–113.
53. Basanta-Val P, Garcia-Valls M, Estevez-Avres I. Extending the Concurrency Model of the Real-Time Specification for Java. *Concurrency and Computation: Practice and Experience* 2011; **23**:1623–1645. DOI: 10.1002/cpe.1675.
54. Basanta-Val P, Garcia-Valls M, Estevez-Avres I. Real-time distribution support for residential gateways based on OSGi. *11^{Th} IEEE Conference on Consumer Electronics*, Las Vegas, 2011.
55. Basanta-Val P, Garcia-Valls M, Estevez-Ayres I. A dual programming model for distributed real-time java. *IEEE Transactions on Industrial Informatics* 2011; **7**(4):750–758. DOI: 10.1109/TII.2011.2166796.