Managing the dynamism of the OSGi Service Platform in Real-time Java Applications

João Claudio Américo
Université de Grenoble
BP 53, 38041 Grenoble, France
Joao.Americo@imag.fr

Walter Rudametkin
Université de Grenoble
BP 53, 38041 Grenoble, France
Walter.Rudametkin@imag.fr

Didier Donsez
Université de Grenoble
BP 53, 38041 Grenoble, France
Didier.Donsez@imag.fr

ABSTRACT

Real-time features and software runtime adaptation are two requirements of modern software. On the one hand, the most important characteristics in real-time applications are their predictable behavior and deterministic execution time. On the other hand, runtime adaptive software are capable of being updated and reconfigured at execution time, making them more flexible and available. The OSGi Service Platform has become the *de facto* platform for developing flexible and modular software, due to its simple service-oriented component model. Many Java applications are being migrated to and developed for the OSGi Platform's component model. However, due to the popularization of real-time solutions such as the Real-Time Specification for Java, some of these applications may have timing constraints which cannot be respected because of the platform's dynamic behavior and the fact that service-oriented component-based application architectures may change at execution time. This paper proposes to delay reconfigurations for after critical processing, according to Service Level Agreements established between service providers and consumers. Our approach has been implemented as an extension to the iPOJO component model which freezes application's architecture and avoids introducing unpredictability caused by runtime adaptation during real-time applications' execution.

Categories and Subject Descriptors

D.4.7 [**Organization and Design**]: Real-time Systems and Embedded Systems; C.1.3 [**Other Architecture Styles**]: Adaptable Architectures.

General Terms

Design, Measurement.

Keywords

Real-time; Service-Oriented Architectures; Real-time Specification for Java (RTSJ); OSGi Platform; Dynamic Software Adaptation; Mode Change.

SAC'12, March 26-30, 2012, Riva del Garda, Italy.

1. INTRODUCTION

Dynamic adaptive behavior and real-time requirements are common needs of today's software. While the former primes for flexibility and unforeseen modifications in the environment at runtime, the latter concerns predictability and determinism of applications' response times. Many solutions for dealing with both aspects separately have been fairly recently developed for the Java platform. One of the most adopted real-time solutions for Java is the Real-Time Specification for Java (RTSJ) [1] and its implementations. At the same time, the popularization of component-based design and service-oriented computing concepts [20] for the development of flexible and modular applications in Java are responsible for the specification of service platforms.

One of the most popular service platforms is the OSGi Service Platform [27]. Its original intention was to become an open specification to develop and deploy services in home gateways, but it has become the *de facto* standard for developing general-purpose Java applications in a modular and flexible way. Its popularization in several domains is due to, among many other things, its adoption by the Eclipse Foundation for developing plug-ins for their IDE. The development paradigm of the OSGi framework is both service-oriented and component-based. This specification, that in former days addressed embedded systems, was extended to cover many other domains, such as mobile phones, industrial supervision, automobiles and more recently a whole set of Java Enterprise Edition application servers.

In general, runtime adaptive software and real-time software are disjoint sets due to the conflict between predictability and flexibility. However, two factors motivate us to find solutions to the growing number of systems which may be in the intersection of both application classes: First, the fact that even critical real-time software, which cannot have its execution interrupted, must be updated due to environment changes or maintenance; and secondly, the increasing popularity of service-oriented and component-based approaches to achieve runtime adaptation, which is leading industries and developers to migrate their applications to service and component frameworks. An example is the inclusion of the OSGi framework in the core of several application servers, such as JOnAS [8] and Oracle's (formerly BEA's) WebLogic Real-Time.

In this paper we identify the issues raised by the dynamic modification of service-oriented architectures in Real-time Java applications and suggest approaches to avoid the introduction of unpredictability in real-time software hosted on the OSGi Platform. We propose two approaches to tackle those issues:

1. an architecture freezing strategy, which blocks the application's architecture during real-time execution periods and delay reconfigurations after them;
2. and an SLA-based monitoring system that controls the reconfigurations performed at non-critical execution periods.

We have implemented and validated one of our propositions on Apache Felix, an open source OSGi platform.

This paper is organized as follows. Section 2 presents an overview of Real-time for Java. Section 3 discusses about OSGi and real-time applications. The architectural freezing approach and a real-time extension for an SLA model are presented in the Sections 4. Section 5 presents an implementation for the approach and the experimentation performed to validate it. Finally, Section 6 concludes the paper and presents our perspectives.

2. REAL-TIME JAVA

Real-time systems differ from other information systems in the fact that their correctness depends on both functional and temporal aspects [25]. Timing correctness requirements come from the impact of a real-time system upon the real world. These requirements, in turn, may be expressed in the form of timing constraints for the set of cooperating tasks which compose the system [14]. Those timing constraints, also known as deadlines, can be relative to an event or absolute, precising a point in time for a task to complete its execution. Real-time systems do not necessarily have to be fast, but they must present two properties: they must be predictable, that is, it is possible to mathematically demonstrate, at design time, that all timing constraints will be met; and they must be deterministic, which means that the system has the ability of ensuring the execution of an application despite external factors that can introduce unpredictability. Schedulability analysis and formal verification are two techniques commonly used in order to verify the predictability of a system. The determinism of an application can be measured through its latency (time between an event and the system's response to that event) and its jitter (distribution and standard deviation of latency responses) [3].

In order to develop real-time applications, the entire underlying infrastructure must be real-time. This motivated the creation of real-time scheduling algorithms, operating systems and programming languages. In the context of the Java platform, an extension to standard API, as well as new rules for the Java execution environment, were defined in order to allow the development of real-time systems with the Java programming language. This extension was developed under the name Real-Time Specification for Java (RTSJ) in 2000, by members of companies such as Sun and IBM. The two main areas identified as requiring enhancements were:

- **Thread Scheduling and Dispatching:** RTSJ introduces the concept of schedulable objects (real-time threads, asynchronous event handlers and their subclasses), objects which the base scheduler manages. The RTSJ's base scheduler is priority-based, preemptive, with at least 28 unique priorities with higher execution eligibility than standard Java thread priorities, uses run-to-block scheduling[1], and can perform feasibility analysis for a schedule. Schedulable objects have parameter classes bound to them, representing resource-demand (scheduling, memory or release) characteristics;
- **Memory Management:** RTSJ supports memory management that avoids interfering with the deterministic behavior of real-time code. It allows the allocation of short and long-lived objects in memory areas that are not garbage collected.

Besides these areas, RTSJ also propose enhancements in the mechanisms for synchronization and sharing, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination and physical memory access.

As we have seen in this section, real-time software requires reliability and predictability. However, many current and future real-time applications are dynamic, that is, external conditions may require modifications and adaptations at runtime. In the next sections, we discuss about dynamic software adaptation, the OSGi service platform, which allows for dynamic reconfiguration, and how real-time software may benefit from their approach.

3. MOTIVATIONS FOR A REAL-TIME AWARE OSGI PLATFORM

3.1 Dynamic Adaptation of Real-Time Software

The first works about the importance of structuring software systems were originally led by Dijkstra [9], in the late 1960s. These were the basis for a software engineering discipline called Software Architecture. Software architecture studies ways of structuring software systems, by representing its software components, their interconnections and the rules concerning their design and evolution over time [11]. Structuring systems as interacting components is the result of years of research in software engineering and one of the solutions proposed in order to deal with scalability, evolution and complexity issues in software.

Dynamic software architectures are architectures in which the composition of interacting components changes during the system's execution. Advances in this field have been boosted by the emergence of ubiquitous computing [31] and the growing demand for autonomic computing [16]. The main motivation for runtime adaptive software is to avoid the risks, costs and inconveniences presented by the downtime of software-intensive systems because of environment changes [19]. An example is dynamic software update, in which the application is able to update itself to fix bugs and add new features without requiring a stop and a restart. Nonstop and critical systems, such as air-traffic control systems, enterprise and financial applications, which must provide continuous service, are examples of applications in which dynamic update is required [18]. However, this flexibility has a cost: safety. Although we can perform modifications which were not planned during the design phase, we cannot anticipate the effects of a dynamic modification.

Several techniques have been developed to enable dynamic adaptive behavior in applications:

- **Component-Based Design:** Software components are independent software units, which are composed in order to build a complete system, with contractually

[1] It means that a schedulable object in execution will continue running until it either finishes its execution or is preempted by a higher-priority schedulable object.

specified interfaces and explicit context dependencies. Dynamic adaptation can be performed using late binding mechanisms, which allows coupling components at runtime through well-defined interfaces [26]. The foundation of a component-based methodology lies on its software component model, which defines what components are, how they can be constructed, assembled, deployed, etc.

- **Architecture Description Languages:** ADLs are languages used to describe software architectures. Common elements of ADLs are components, connections and configurations. ADLs may be used to specify points of variability in the architecture at runtime. For instance, Fractal [2] is a component model which contains an ADL. The architecture description of an application can be queried by means of an X-Path-based language called F-Path and reconfigurations can be performed through a DSL called FScript [7].
- **Dynamic Service-Oriented Architectures:** Service-Oriented Architecture (SOA) is an architectural style and a programming model based on the service concept [20]. A service is a software unit whose functionalities and properties are declaratively described in a service descriptor. Services can be composed and orchestrated to create more complex services. Service providers register the description of their services in a service register. Service consumers query the service register to discover and select services. Then, after negotiating and agreeing to service usage terms, the service consumer is bound to the service provider. Dynamic SOA [4] is an extension to the SOA model, which considers that services may appear, disappear or modify their contract at runtime.

Real-time adaptive systems may be used to implement real-time systems which need flexibility, adaptive systems whose interactions with other software entities must meet real-time requirements, or systems which present both characteristics. Several works in the literature present approaches for dealing with the dynamic reconfiguration of real-time applications. Among these approaches, we find the use of mode change protocols [21]. An operating mode is characterized by a goal and a set of functionalities, which in turn are provided by different sets of tasks. Changes in the application state may request a mode change, transitioning the system from one operating mode to another. During the transition, the set of tasks may include both old- and new- mode tasks. In this approach, it is crucial to assure that the overload produced by the transition will not interfere on the system's predictability and cause any deadline to be missed. Other approaches for the development of real-time adaptive systems include real-time extensions to object request brokers [30][6] and the componentization of objects representing quality of service attributes [24].

3.2 The OSGi Service Platform

The OSGi Service Platform is a service platform which addresses the lack of support for modularity in Java applications [13] and applies SOA principles to the design of Java applications. The OSGi specification defines a way to create modules (bundles, in OSGi terminology) and to make them interact at runtime. Bundles are actually Jar files with meta-data specifying their symbolic name, version and dependencies. The central idea of OSGi modularization is that each bundle has its own class-loader, and consequently, its own class path. In order to allow interactions among bundles, OSGi uses a mechanism of explicit package imports and exports. In addition, the OSGi platform allows bundles to be dynamically installed, updated and uninstalled, without requiring the platform to stop and restart. Besides the deployment mechanisms, the specification defines a non-distributed service platform for Java, which allows services to be dynamically published and consumed in a single process or memory space. OSGi services do not introduce overhead when being invoked thanks to direct references. They are also known as micro services (*μServices*) [17] or light-weight services in opposition of well-known and heavy-weight Web Services.

The OSGi Service Platform has been a widely adopted technology for home automation, pervasive environments and enterprise contexts, due to its dynamic service component model, flexible remote management and its continuous deployment support. However, it lacks support for real-time applications, which restricts its application to environments where real-time requirements do not have to be guaranteed. Indeed, the continuous deployment support allows bundles to be installed, started, stopped and uninstalled at anytime, thus the static system configuration assumption is no longer valid, because the system will evolve during the application's life-cycle.

3.3 Challenges

When we consider the execution of real-time applications in the OSGi Service Platform, several potential issues must be taken into account. The most evident of them is that the OSGi Service Platform was not conceived as a real-time application. So far, all of its implementations were written in standard Java and its classes will need to interact with real-time components, possibly written in the RTSJ or another real-time Java technology. Considering RTSJ specifically, this may lead to problems such as:

- **Memory leaks:** In the RTSJ, class objects and static objects are stored automatically in the immortal memory area, which is never garbage-collected. Consequently, this may complicate uninstalling components and class unloading.
- **Starvation:** The OSGi framework is based on standard Java. Consequently, it uses ordinary Java threads. RTSJ components have real-time priorities, which are higher than the ordinary ones. Thus, due to the fact that RTSJ's scheduler uses a run-to-block scheduling policy, real-time component threads may lockout system threads and keep the administrator from issuing commands to the framework.

In this study we will focus mainly on ways to deal with real-time constraints in the OSGi service platform, where dynamic adaptability is a required property. An example of an application with soft real-time requirements which requires dynamic adaptations to its architecture is a digital security camera monitor system (see Figure 1). In this application, the intrusion detection system is connected to several security cameras which provide either a still image frame or motion frame service, or both. The number of security cameras connected to the intrusion detection system is unknown at design time. At runtime, once a camera component is installed, thanks to the system's Plug-and-Play deployer, it is automatically connected to the intrusion detection module, which will process frames in order to detect human presence. Indeed, cameras in this system have the dynamic

availability property, being able to appear and disappear at any time, in order to facilitate installation and maintenance and provide fault-tolerance to device and network failures. For instance, if we assume that frames are sent regularly to the motion detection module, image processing time must be bounded in order to allow the system to react as soon as possible to a human presence. Four points must be observed concerning the impact of dynamic availability in the architecture of the real-time system:

- **Binding to a new camera component:** The system's underlying mechanisms which perform the binding between components may add an unpredictable delay in the image processing time. Moreover, depending on the number of cameras to which the intrusion detection system is connected, processing time may become longer than the time attributed to the execution of the intrusion detection system.
- **Removal of a camera component:** In the same way as binding mechanisms, unbinding mechanisms may introduce an unpredictable delay in image processing time. Furthermore, we must ensure that the removed camera component is not currently being used by the intrusion detection system.
- **Update/reconfiguration of a camera component:** Updating a camera component in the OSGi platform makes the component stop and return to the Installed state. Component dependences are then resolved and afterwards the component is restarted. The duration of this process cannot be predicted. In addition, the new camera component may have different properties. Thereby, a feasibility analysis which may have been performed before is no longer valid and therefore must be redone.
- **Binding a camera component to another component:** Another factor that must be considered is the case where the camera component provides the image frame service to more than one consumer. Sending the frame to one of the components while the other waits may introduce an unpredictable wait time for the former component.

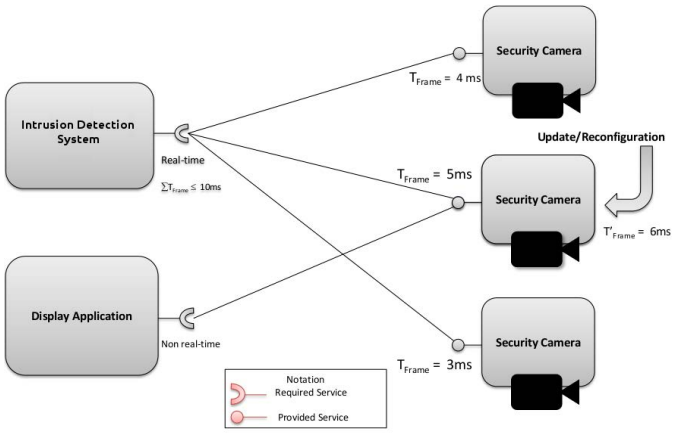

Figure 1. Video monitoring application

To summarize, besides the real-time issues inherent to the Java Platform, the OSGi Service Platform presents several other shortcomings which make it inappropriate for deploying and developing real-time applications. These shortcomings may have two different reasons. The first one is the fact that the OSGi platform was not conceived for being used in the context of real-time applications. The other one comes from the dynamism provided with its service-oriented component model. Indeed, component dynamic availability features allow components to appear and disappear unexpectedly at runtime. In this study, we will focus on the dynamism aspect.

4. PROPOSITIONS

We propose an architecture freezing policy when the platform is in a real-time processing state, holding all reconfigurations until the end of execution of the critical code. Then, modifications may be performed if they respect an agreement established between the service consumer and the service provider.

4.1 Architectural Freezing

We consider that in a real-time application every component is able to perform its tasks within the real-time requirements of the application. Thus, the issues we are interested in lie in the bindings between components and how they change over time. Suppose that an application is represented by a set of states. Each state corresponds to a given architecture of the application[2], and transitions between states correspond to the arrival, the departure or the update of a component in the application during runtime. In consequence, in order to respect the application timing constraints, just as in a mode change protocol, we must define rules for the transitions between states.

We have included a new mode to the OSGi Platform, called "Real-time Mode". As shown in the Figure 2, contrarily to the non-real-time mode, once the system enters the real-time mode, no modifications are performed until it returns to the corresponding non-real-time state, in order to ensure that real-time requirements will be met. The mode change is explicitly requested by the bundles before starting to execute a critical piece of code. After the real-time processing, the bundle itself communicates the platform that its critical code period has finished. Many bundles may request the mode change at the same time. Once the system has entered the real-time mode, it only switches back to the non-real-time mode when no bundle is executing critical code.

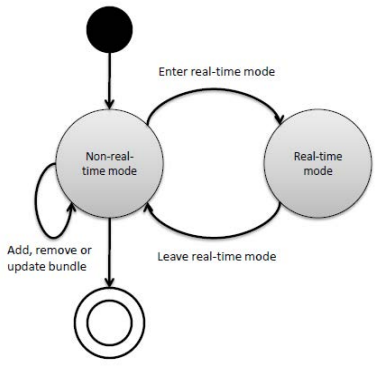

Figure 2. OSGi Platform state diagram with the inclusion of a real-time mode.

[2] By architecture we mean a set of bundles and the interconnections among them. Consequently, a bundle update leads the application to a new state, even if the system configuration seems to be the same.

Figure 3 shows the bundle life-cycle for bundles on both non-real-time and real-time modes. Bundles cannot be started or stopped in the real-time mode, so we removed the arrows which represent this transition. Consequently, the life-cycle becomes a graph with two separated components: Bundles which are resolved but not started cannot be started during a real-time mode, which is represented by the Installed, Resolved and Uninstalled states; Active bundles cannot be stopped and remain in the Active state during the real-time mode. We call this approach architecture freezing, due to the fact that the system holds all the architecture changes until the system quits the critical code period. This solution addresses mainly systems whose majority of the code is non-real-time but with some critical pieces of code which are executed periodically.

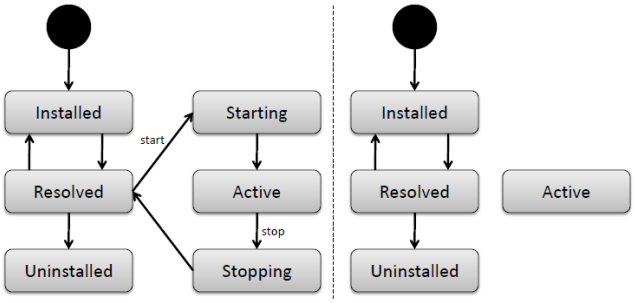

Figure 3. Bundle life-cycle diagram for bundles in both standard (on the left) and real-time (on the right) modes.

This approach is based on mode change, as we added a real-time mode for the OSGi platform and defined rules for the transitions to and from this mode. However, mode change protocols are focused on tasks, while the OSGi platform handles bundles, which may contain and execute multiple tasks simultaneously.
Nevertheless, avoiding components to be started, stopped or updated during real-time processing is far from being enough. We must ensure that the modifications that will be carried out at the non-real-time mode will not affect the predictability of the real-time components hosted by the platform. Therefore, we suggest in the next section a SLA-based mechanism in which architecture modifications may be performed if they respect an agreement established between the service consumer and the service provider.

4.2 Real-Time Dynamic Service Level Agreement

Service Level Agreement (SLA) [29] is a negotiated part of the contract established between the service provider and the service consumer which formally defines the level of service and the penalties applied when commitments are not met by either party. These commitments are specified in order to reach a given quality of service. A SLA contains information such as the parts engaged in the agreement, the service provided, service utilization time, service availability, service reliability, service utilization price and dates for renegotiating the agreement. SLAs are monitored by Service Level Management (SLM) modules, which are also responsible for applying the penalty policies in case of non commitment of the agreement. In order to avoid equity issues, a third party (the service certifier), chosen by the service provider and consumer, takes measures periodically in order to verify that the contract clauses are not violated.

In this work, we have extended the Dynamic SLA model proposed by [28]. This DSLA model handles dynamic availability and service disruption, by adding meta-data which will be used by the monitor module for accepting new components and handling real-time constraints. Our SLA model is depicted on the Figure 4 (for the sake of clarity, model restrictions are not represented).

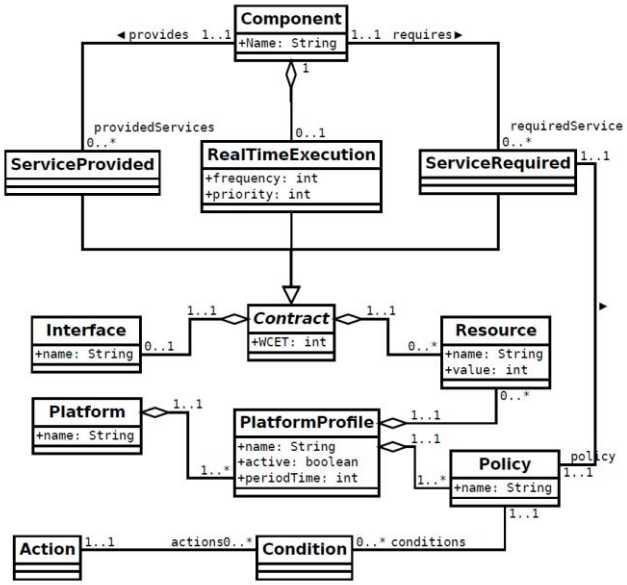

Figure 4. Our real-time SLA model.

The model contain two main entities: Component and Platform. **Components** explicitly declare its required and provided services in a meta-data file, alongside of a description of its timing constraints (represented by the class RealTimeExecution), when necessary. Provided and required services, and timing constraints are considered as contracts, which must be monitored by a Real-time Service Level Manager (SLM). Contracts are composed by an interface and a list of resources, whose role depend on the type of contract: for provided services and timing constraints, it means the resources required by the service provider or real-time component to execute; for service consumers, it acts as a filter, that is, only services which use less or equal the given quantity of resource can be selected. Service consumers also declare a policy, which must be applied in case of contract violation.
A **Platform** is formed by a list of profiles, but only one profile at a time can be active. In turn, a platform profile has a period time (during which it must schedule all the executing components), a list of resources available to the components it executes and a list of policies to be applied in case of contract violation. The resources declared by the components on its contracts must obviously be a subset of the resources declared by the platform.

5. IMPLEMENTATION AND VALIDATION

We have implemented both approaches by means of an extension to the iPOJO component model [10], which intercepts modifications in the application's life-cycle. Components which execute critical code are connected to a Real-time Manager, whereby they can request a real-time mode change and notify the platform when its real-time processing is over. Our iPOJO

extension is also connected to this manager, and queries whether the system is in a real-time mode or not before the execution of a modification; in a positive case, the object which performed the query is blocked. Once there is no component executing a critical code, a notification is sent to the manager, that unblocks all the blocked object. This notification module was developed by means of RTSJ's asynchronous events and event handlers.

When the system is at non-real-time mode, our Real-time Manager iPOJO extension performs several verifications accordingly to the currently established SLAs, before allowing the modification:

- In case of a component addition, it verifies whether this component contains a real-time execution profile, with timing constraints; if so, it will only start the component if component's worst case execution time (WCET) plus the sum of the WCET of its required services fits on the active platform's period time and if all the required resources can be reserved;
- In case of a component removal, it updates the platform's current profile, by releasing the occupied resources and updating the remaining period time; additionally, it performs the same action for all components which depends on the given component and cannot be bound to any other service provider. For those whose provider can be substituted, it updates the platform's profile with the new data.
- In case of a component update, as the platform stops the currently used component and starts the updated one, the previously described actions are taken.

In order to evaluate the determinism of our solution, we adapted the benchmark provided with Oracle/Sun's Java Real-time System 2.2 to the OSGi Platform and to the iPOJO component model, as shown at Figure 5. Tests were performed on an architecture composed by an Intel Core i5 2.4 GHz processor and a 4 GB RAM memory, running the version 2.6.31 of the RT Linux Kernel for Ubuntu 11.04. The benchmark consists of a client which performs a given number of calls towards a class which performs CPU-intensive work and measure the jitter and standard deviation of the set of execution times[3].

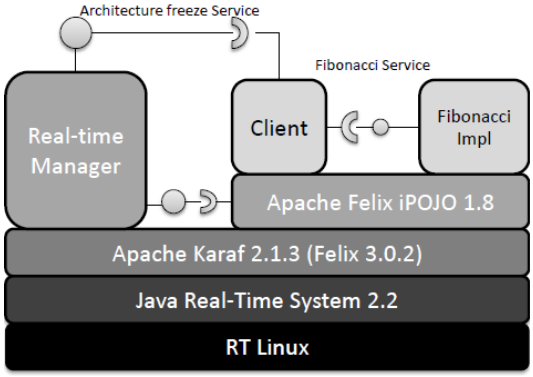

Figure 5. Benchmark application architecture.

[3] We are currently working on a service-oriented adaptation of the CDx benchmark [15], which is used to evaluate Real-time Java virtual machines through a collision detection application.

The service provider, in turn, was modified to start and stop stochastically according to a given probability. Figures 6 and 7 show the jitter and mean execution type respectively obtained for different values of the probability parameter (in figure 7, the red error bar represents the standard deviation for the mean of the execution time with the real-time manager, as the blue error bar is referent to the execution time without the manager). For this benchmark, the client executes 1000 calls to the FibonacciService. The Fibonacci Implementation calculates recursively the 40th term of the Fibonacci series and returns its value. This way, we simulate a service which unpredictably appears and disappears (this may also be considered as an update), but whose availability is controlled by means of software (e.g. we do not consider services attached to physical devices which may be powered off and disconnected from the network, as there is no way to avoid this type of service disruption through software). For the moment, we assume that components do not have a malicious behavior, and will not block the whole platform in a real-time state. However, this case can be easily covered by implementing a timeout mechanism.

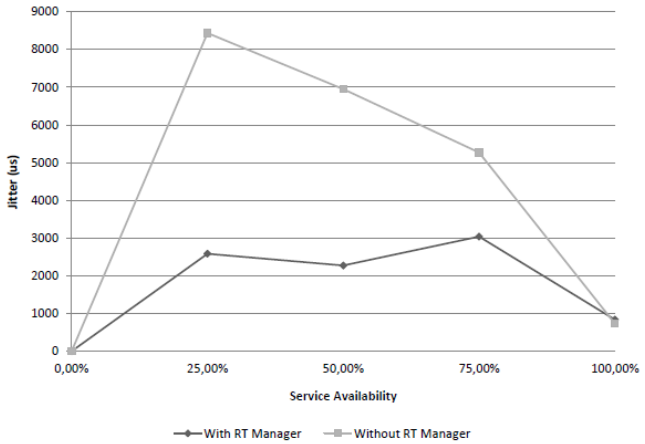

Figure 6. Jitter for a 0%, 25%, 50%, 75% and 100% service availability

As we may see, by using architectural freezing we obtain more deterministic results, regardless of the service unavailability probability. Additionally, it does not incur overhead costs in the mean execution time. Both results can be explained by the same effect: by avoiding the service provider to leave, we do not have to wait until it returns (or wait for a substitute), what makes the response time shorter and more homogeneous.

6. RELATED WORKS

Few works have been dedicated to provisioning real-time support in OSGi, and most of them deal with the isolation of components instead of the issues raised by the dynamism of the platform [5][12][23]. The first study concerning dynamics of real-time applications on the OSGi Platform was developed by [22]. It proposes an admission control protocol for applications over a real-time OSGi platform. We also address this issue through our SLA-based approach, preventing the installation of components which may compromise other components' determinism.

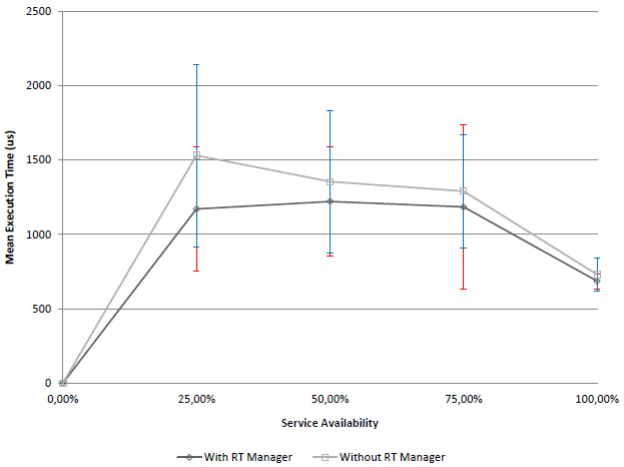

Figure 7. Mean execution time for a 0%, 25%, 50%, 75% and 100% service availability.

7. CONCLUSION AND PERSPECTIVES

In this paper, we have focused on the conflict between the predictability required by real-time applications and the dynamism provided by dynamic software adaptation frameworks, such as the OSGi Service Platform. Our motivation to approach this conflict comes from the growing use of the OSGi Framework for developing applications and the popularization of Real-time Java technology. In order to deal with dynamism issues in real-time applications hosted in the OSGi Service Platform, we suggested the distinction between real-time processing periods, where no architecture modification is allowed (architectural freezing), and non real-time processing periods, where architecture modifications are allowed under the condition of respecting service level agreement established between service provider and service consumer components in the platform. An implementation of the architectural freezing approach was developed by means of iPOJO handlers. We chose iPOJO handlers as the base of our implementation in order to increase the portability of our solution and avoid modifications to the standardized OSGi platform and iPOJO core's corresponding source code. The prototype functioned as expected: when components entered the real-time state, the real-time manager component held all the dynamic component modifications, performing them once the execution of real-time code had terminated. Regarding future work, we intend to explore the issues raised by the cohabitation of real-time and non-real-time service providers and consumers in the same OSGi platform.

The three open source implementations of the OSGi Service Platform (Apache Felix, Equinox and Knoplerfish) are already mature, as they date from around 10 years old. Our proposition is not to create a new whole specification (and consequently implementation), but to be compatible with the existent platforms with a minor impact on the developers' code. We see our proposed solutions and our prototype as a first step towards the development of a real-time OSGi platform and of extensions for OSGi's component model and other real-time service-oriented component models.

8. REFERENCES

[1] G. Bollella, K. Jeffay, and J. Gosling. The real-time specification for java. *IEEE Computer*, 33:47-54, 2000.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36:1257-1284, September 2006.

[3] E. J. Bruno and G. Bollella. *Real-Time Java Programming: With Java RTS*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2009.

[4] H. Cervantes and R. S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 614-623, Washington, DC, USA, 2004. IEEE Computer Society.

[5] G. Coates. Real-time OSGi. www.osgi.org/wiki/uploads/VEG/Aonix-RT-OSGI.ppt, 2007.

[6] A. Corsaro. Cardamom: A next generation mission and safety critical enterprise middleware. In *Proceedings of the Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 73-74, Washington, DC, USA, 2005. IEEE Computer Society.

[7] P.-C. David, T. Ledoux, M. Léger, and T. Coupaye. Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures. *Annales des Télécommunications*, 64(1-2):45-63, 2009.

[8] M. Desertot, D. Donsez, and P. Lalanda. A dynamic service-oriented implementation for java ee servers. In *Proceedings of the IEEE International Conference on Services Computing*, SCC '06, pages 159-166, Washington, DC, USA, 2006. IEEE Computer Society.

[9] E. W. Dijkstra. The structure of the "the"-multiprogramming system. *Commun. ACM*, 11:341-346, May 1968.

[10] C. Escoffier, R. Hall, and P. Lalanda. ipojo: an extensible service-oriented component framework. In *Proceedings of the 2007 IEEE International Conference on Service Computing*, pages 474-481, july 2007.

[11] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1-39. Publishing Company, 1993.

[12] N. Gui, V. De Flori, H. Sun, and C. Blondia. A framework for adaptive real-time applications: the declarative real-time osgi component model. In *Proceedings of the 7th workshop on Reactive and adaptive middleware*, ARM '08, pages 35-40, New York, NY, USA, 2008. ACM.

[13] R. Hall, K. Pauls, S. McCulloch, and D. Savage. Osgi in Action: Creating Modular Applications in Java. Manning Pubs Co Series. Manning Publications, 2010.

[14] D. Isovic and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In Proceedings of the 21st IEEE conference on Real-time systems symposium, RTSS'10, pages 207-216, Washington, DC, USA, 2000. IEEE Computer Society.

[15] T. Kalibera, J. Hagelberg, P. Maj, F. Pizlo, B. Titzer, and J. Vitek. A family of real-time java benchmarks. Concurrency and Computation: Practice and Experience, 23(14):1679-1700, 2011.

[16] J. O. Kephart and D. M. Chess. The vision of autonomic computing. Computer, 36:41-50, January 2003.

[17] P. Kriens. µservices. http://www.osgi.org/blog/2010/03/services.html, Mar. 2010.

[18] J. Magee and J. Kramer. Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes, 21:3-14, October 1996.

[19] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: framework, approaches, and styles. In Companion of the 30th international conference on Software engineering, ICSE Companion '08, pages 899-910, New York, NY, USA, 2008. ACM.

[20] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing. Communications of the ACM, 46:25-28, 2003.

[21] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. Real-Time Syst., 26:161-197, March 2004.

[22] T. Richardson and A. Wellings. An admission control protocol for real-time osgi. In Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 217-224, 2010.

[23] T. Richardson, A. J. Wellings, J. A. Dianes, and M. Díaz. Providing temporal isolation in the osgi framework. In Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES '09, pages 1-10, New York, NY, USA, 2009. ACM.

[24] P. K. Sharma, J. P. Loyall, G. T. Heineman, R. E. Schantz, R. Shapiro, and G. Duzan. Component-based dynamic qos adaptations in distributed real-time and embedded systems. In Proceedings of the International Symposyum on Distributed Objects and Applications (DOA), volume 3291 of Lecture Notes in Computer Science, pages 1208-1224. Springer, 2004.

[25] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. Computer, 21:10-19, October 1988.

[26] C. Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[27] The OSGi Alliance. OSGi service platform core specification, release 4.3. http://www.osgi.org/Specifications, 2011.

[28] L. Touseau, D. Donsez, and W. Rudametkin. Towards a sla-based approach to handle service disruptions. In Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 1, pages 415-422, Washington, DC, USA, 2008. IEEE Computer Society.

[29] D. Verma. Supporting Service Level Agreements on IP Networks. Macmillan Technical Publishing, 1999.

[30] N. Wang and C. Gill. Improving real-time system configuration via a qos-aware corba component model. In Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack. HICSS, 2003.

[31] M. Weiser. Ubiquitous computing. Computer, 26:71-72, 1993.