fANFARE: Autonomic Framework for Service-based Pervasive Environment

Yoann Maurel*, Stéphanie Chollet†, Vincent Lestideau‡, Jonathan Bardin‡, Philippe Lalanda‡ and André Bottaro*
*Orange Labs Grenoble
28 Chemin du Vieux Chêne
F-38243 Meylan, France
{firstname.lastname}@orange.com
†Laboratoire de Conception et d'Intégration des Systèmes
F-26902, Valence cedex 9, France
stephanie.chollet@lcis.grenoble-inp.fr
‡Laboratoire d'Informatique de Grenoble
F-38041 Grenoble cedex 9, France
{firstname.lastname}@imag.fr

***Abstract*—The ability to react quickly to unpredictable changes in the environment is a key requirement in pervasive computing. This paper presents fANFARE, a framework for the autonomic management of service-oriented applications in pervasive environments. Specifically, it focuses on the configuration and optimization of pervasive applications deployed on OSGi platforms. We propose to handle runtime administration through a hierarchy of autonomic managers, that is a platform manager and a number of application managers and dependency managers. Our approach has been implemented and validated on pervasive use cases within the MEDICAL project funded by the French Ministry of Industry.**

***Keywords*-Autonomic Computing, Integration middleware, Formal Concept Analysis**

I. Introduction

As Mark Weiser's [1] vision is becoming more and more concrete, a new world of applications emerges with the proliferation of devices available in the home, building or cities. This comes along with an increasing number of interconnected computing platforms, characterized by their heterogeneity and volatility. These platforms include set-top-boxes, Internet boxes, smartphones, tablets, etc. They are more and more used to run applications orchestrating and sometimes managing a number of sensors/effectors. In a near future, applications will be more and more available for deployment on those platforms in order to manage energy, security, entertainment devices, etc.

The dynamic aspect of pervasive environments justifies the use of service-oriented approaches [2] as a base paradigm for the development and execution of pervasive applications. The use of services offers the possibility to create, deploy and manage multiple configurations of the same application. This certainly increases the complexity of the systems but, in return, it provides the flexibility demanded by context-aware pervasive applications.Pervasive applications raise however major challenges in terms of administration. This problem is made particularly severe in distributed environments made of a number of computing platforms and volatile devices. In addition, many applications are critic in the sense that they directly intervene in the physical environment. Humans have to be protected against inappropriate actions caused by incorrect administration (after badly controlled updates for instance). We believe that administration actions should be done automatically as much as possible. In particular, adaptations and optimisation due to context evolution should be dealt with programmatically. However, such optimizations can be expressed in the form of high-level objectives fixed by the users and/or administrators. It seems clear to us that before be distributed to the general public, pervasive platforms and applications should be managed in an autonomic way.

This paper presents fANFARE, a framework for the autonomic management of service-oriented applications in pervasive environments. Its purpose is to provide a solution to three major issues in pervasive computing: 1) the discovery and use of services seamlessly by hiding their technology heterogeneity, 2) the classification and the autonomic selection of services at runtime and 3) the supervision of resources and platform components to allow evaluation of the behaviour of services and applications. We believe that these three challenges are very much related. Indeed, being able to integrate all sorts of services, regardless of their technology, dramatically increases the number of usable services when dynamically building an application. This makes the basic "all-or-nothing" selection techniques ineffective and obsolete. In this context, service selection becomes a central issue for the configuration, optimization and repair of pervasive applications. Complementarily, services elected for execution must be carefully supervised. They come from multiple providers, multiple servers, possibly in the cloud, and cannot be completely and definitively trusted. Also, unforeseeable interactions between applications can result in undesirable actions. First thing to allow the dynamic composition of applications and their efficient monitoring

is, obviously, to detect the available services in the environment. To do so, we rely on an extensible integration middleware called RoSe [3]. RoSe is available in open source and used in many industrial projects. The extensibility feature is absolutely necessary. Indeed, pervasive environments are characterised by the number of available services but also by the heterogeneity of these services. New protocols, new formats are regularly needed and must be integrated rapidly in communication middleware like RoSe. As we will see, in the MEDICAL project, we extended RoSe to support such protocols as Modbus for instance.

Second thing is the ability to evaluate and select the services dynamically discovered by RoSe. Here, our approach is based on FCA (Formal Concept Analysis) in order to classify and identify equivalent services sets regarding some high level goals expressed by a user/administrator. Specifically, we endow every service with a local, very efficient, manager in charge of the dynamic management of service dependencies. Such a manager uses FCA structures and can decide to replace a service at anytime to better meet the current objectives, which can be also changed anytime. It is noteworthy that this notion of high level goals can be rather complicated in the sense that goals can be in opposition. It is then needed to define context-based cost functions to make a decision.

Finally, our approach is complemented with supervision mechanisms. We have defined and implemented on top of OSGi a set of monitoring mechanisms that can activated on the fly. This allows a dynamic focus of the monitoring activities depending on the context (current bugs or misbehaviours to be handled for instance). It also permits to adjust the cost of monitoring to the situation. The paper is structured as it follows. First section II outlines our approach. In the three following sections, we detail the approach. Specifically, we discuss the discovery service in Section III, the dependency manager in Section IV and the monitoring tool in Section V. Before concluding, Section VII discusses related works.

II. Global Approach

Our work is based on OSGi platforms. This framework, supporting the dynamic execution of service-oriented components, is today recognized as a solution of choice for pervasive computing. OSGi platforms generally host business components and are able to integrate devices using diverse technologies. In a typical application, the business logic is implemented as OSGi bundles while most of the sensors use UPnP or DPWS technologies.

OSGi and service component models such as OSGi Declarative Services [4] or iPOJO [5] establish a clear distinction between bundles, components, service references, service factories and service instances. Typically a bundle contains several components that provide or use services. A service is associated with one or many service references. References refer to the Java interfaces provided by the service and service properties. The service is provided by a service object (Java object instance) that can be retrieved via the registry. For simplicity sake, we make no distinction here between a bundle providing a service and a service instance, between a bundle requiring a service and a service client, and between a service instance and a service reference.

This paper focuses on the administration of applications deployed on the OSGi framework. We propose to manage such applications through a hierarchy of managers (see (Figure 1)) including a platform manage and a number of application and service managers. This hierarchy is derived from the generic architecture described by [6].

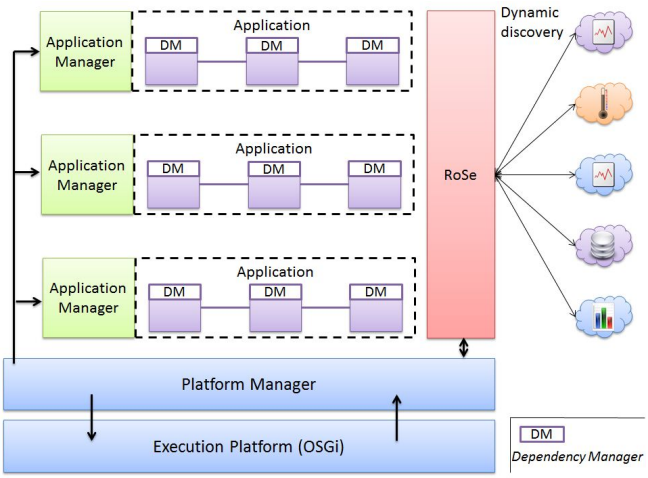

Figure 1. Service-based pervasive applications are managed by a hierarchy of managers.

Let us see the role and duties of these managers. The platform manager, first, has two main responsibilities. First, it manages the global behaviour of the platform through the join optimisation of the different applications. With respect of the scope of this paper, it influences the service selection policies of the dependency managers. To do so, it is able to dynamically modify the administration goals of the application managers, which, in turn, implies internal reconfigurations at the application level. For a given platform, an administrator can define mandatory and optional features, including non-functional aspects like cost per use, location, reliability, performances or security. The platform manager also holds a prioritized list of applications. Applications are ranked by the administrator depending on their criticality. Depending on these priorities and application specific goals, the platform manager configures each application managers' goals.

The platform manager is also in charge of the management of the available resources, like CPU or memory. Values characterizing the resources are collected by several monitoring mechanisms that have been implemented within the OSGi framework. CPU-monitoring is progressive and

localized: each mechanism can be enabled or disabled on-the-fly. When CPU-intensive applications are discovered, the platform manager sends an alert to the concerned application manager. This alert includes a list of suspected bundles that should be stopped or replaced. If the application manager is not able to solve the problem in a limited time, the platform manager can send an alert to the administrator or stop the suspected bundles depending on the application criticality.

Application managers are in charge of the management of groups of bundles. The mapping between bundles and application managers is done manually by the administrator at deployment time or automatically on the basis on meta-information provided by bundles (providers and dependencies). The bundles may change groups during the execution if required by the administrator. In particular, since applications are typically distributed among multiple platforms and that bundles and services are shared across multiple applications, an application may in fact be managed by several application managers. For instance, when a bundle is shared by multiple applications, the administrator can assign it to a new application manager. This allows avoiding conflicts in the management. The main advantage of grouping bundles into consistent sets is to enable the application of specific policies to a particular group of bundles or applications.

Application managers also deal with a large part of service selection. They play the role of smart registries for their associated bundles and dependencies managers. Depending on the required services, they subscribe and discover services in the OSGi registry using LDAP filters. This allows the selection of relevant services only. The application manager first ensures the consistency between properties. Each property of the small subset of interesting services are then discretized if necessary depending on the application context. Discretization is performed using a set of thresholds defined by the administrator. Properties interpretation is thus application-dependant. For instance the interpretation of ”high-cost per use” may vary between critical and comfort applications. The application manager then organizes services using the FCA algorithm. This structure is divided into substructure and subgoals that are sent to dependency managers. One substructure (or decision lattice) is computed per services bindings. These structures help dependency managers to find the most suited services depending on the goals.

Components are endowed with a dependency manager in charge with the proper selection of services using the FCA structure and goals provided by the application manager. Decision structures allow to easily partition discovered services into equivalence classes. Goals include a set of mandatory properties (*i.e.*, the desired functionality) and an ordered set of optional properties (*i.e.*, cost, reliability, UPnP). The dependency manager then uses the equivalent classes provided by FCA to choose the most suitable services. Benefits of this approach are that the decision structures are small and their inspection to choose services is fast. This allows the fast substitution of services when a service becomes unavailable. Additionally this approach avoids the selection of no service by providing equivalence classes. Finally as the structures are updated by application managers only when needed, that is on a context change or when the list of available services has undergone too many changes. For instance, changing the order of priorities of optional properties does not require changing the structure.

Next sections are organized as follows. Section III describes the RoSe middleware for managing service heterogeneity and discovery. Section IV explains why FCA is a method of choice for service selection and how this technique is used by application and dependencies managers. Finally, Section V focuses on the resource management and how the platform manager carries the progressive monitoring of the system.

III. RoSe: Heterogeneity Management

Service discovery is based on the integration middleware named RoSe [3]. RoSe handles in a transparent way services distribution and heterogeneity. It reacts to the arrival and departures of services (Figure 2). It is an OSGi-based open source middleware[1].

It allows service discovery, the automatic instantiation of a specific proxy with different strategies, the management of the proxies life-cycle and the ability to automatically generate a proxy for well-identified services. Proxies representing remote services are directly inserted in the OSGi registry. Consequently, remote services are accessible as regular OSGi services, that is to say programmers are able to access remote services just like local ones.

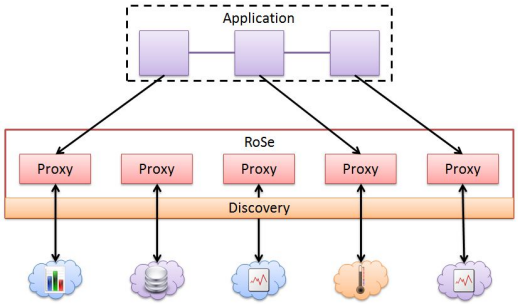

Figure 2. RoSe in action.

Concretely, when RoSe discover an available service (published by a device for instance), a proxy is dynamically created providing a local delegate of this service. Different protocols are supported here, including Web Service, UPnP, DPWS, etc. Finally when a remote service is no longer available, its corresponding proxy is destroyed. To describe the expected features and find corresponding services, RoSe

[1]http://wiki.chameleon.ow2.org/xwiki/bin/view/Main/Rose

provides a query language to define configurations as pairs of properties/values. By applying the different configurations, we obtain the matching service(s). It is possible to define LDAP filters among the set of available configurations to refine the search. In addition, RoSe allows the dynamic modification of filters and configurations at runtime.

IV. Selection through the use of FCA

We propose to use the Formal Concept Analysis method to classify the dependencies between components. This classification is stored in each dependency manager as a decision structure. First, we present the computation of this decision structure. Then, we detail the use of this structure in an autonomic context.

A. Formal Concept Analysis Foundations

Formal Concept Analysis (FCA) [7] is a theoretical and mathematical framework used to classify items. We very shortly define the main concepts of FCA. The purpose of FCA is to build a partially ordered structure, called concept lattice, from a formal context.

Definition 1: A **formal context** $\mathbb{K}$ is a set of relations between objects and attributes. It is denoted by $\mathbb{K} = (O, A, R)$ where O and A are respectively sets of **Objects** and **Attributes**, and R is a **Relation** between O and A (Figure 3).

Definition 2: A **formal concept** C is a pair (E, I) where E is a set of objects called **Extent**, I is a set of attributes called **Intent**, and all the objects in E are in relation R with all the attributes in I.

Thus, the Extent of a concept is the set of **all** objects sharing a set of common attributes, and the Intent is the set of **all** attributes shared by the objects of the Extent. Formally:

- $E = \{o \in O, \forall i \in I, (o, i) \in R\}$,
- $I = \{a \in A, \forall e \in E, (e, a) \in R\}$.

Consequently, a formal concept $C = (E, I)$ is made of the objects in E which are exactly the set of objects sharing the attributes in I. For example, $(\{1, 2, 3\}; \{c\})$ is a formal concept in the context of Figure 3.

Let X a set of attributes. We define the function $Closure_{\mathbb{K}}(X)$ which associates to X the concept made of the set of objects sharing X and the other attributes shared by this set of objects. Note that the computation of a formal concept from a set of attributes X of size n has a complexity of $\mathcal{O}(n \times m)$ where m is the number of objects.

Definition 3: A **concept lattice** is defined as the pair $(\mathcal{C}(\mathbb{K}), \leq_{\mathcal{C}})$. Let two concepts (E_1, I_1) and (E_2, I_2) we say that (E_2, I_2) is a successor of (E_1, I_1) if $(E_1, I_1) <_{\mathcal{C}} (E_2, I_2)$. Given I_1 a subset of A, we note by **successors**(I_1) the set of successors of the concept (E_1, I_1). The concept lattice can be represented by a particular graph called Hasse Diagram (Figure 3).

Note that the computation of a concept lattice from a formal context has a complexity of $\mathcal{O}((n+m) \times m \times |\mathcal{C}(\mathbb{K})|)$ where n is the number of attributes and m is the number of objects [8]. Most of the time we have $n << m$ and the complexity becomes $\mathcal{O}(m^2 \times |\mathcal{C}(\mathbb{K})|)$.

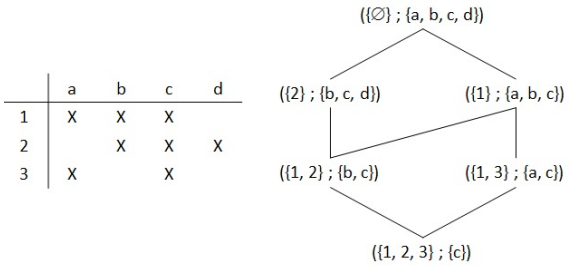

Figure 3. Formal Context and Hasse Diagram.

The set $\mathcal{C}(\mathbb{K})$ of all concepts induced by a context can be ordered using the following partial order relation: $(E_1, I_1) <_{\mathcal{C}} (E_2, I_2)$ if $E_2 \subset E_1$ and $I_1 \subset I_2$.

B. Computation of Decision Structure

In [9], [10], we propose to apply FCA to service selection in pervasive environment. The usability of the FCA is based on the computation of the interesting concepts of the concept lattice. First, we transform the data extracted from the OSGi registry in a formal context where: service functionalities and properties are attributes, available services are objects.

All the computed concepts of the lattice are not interesting. Two exclusive groups are proposed:

- **concepts with no real meaning.** These concepts contain in their intent a set of properties which is not usable.
- **concepts with sense.** Contrary to the previous group, the intent of the concepts makes sense, *i.e.* the intent contains coherent information according to the application context.

In addition, the computation of the lattice is evaluated with a significant complexity. Consequently, we propose to compute only interesting concepts according to the selection request (Figure 4). The ordered interesting concepts, named decision structure, are a sub-set of the concept lattice. Obviously, a concept lattice contains many decision structures and these structures can share common formal concept(s).

The interest to compute formal(s) concept is that the extent contains all the services providing the properties of the intent. For example, in the concept $(\{S_1, S_4, S_7\}; \{Temperature, DPWS\})$, all the services provides the functionality $Temperature$ implemented in $DPWS$. This allows to define equivalent classes of services, *i.e.* a service can be replace by another in case of failure (departure, breakdown...). The decision structure expresses the difference between the services, *i.e.* services are classified by refinement.

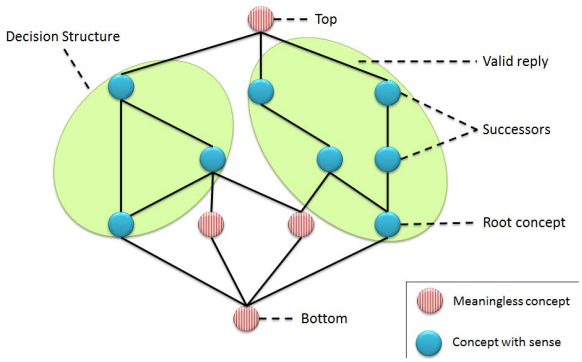

Figure 4. Decision Structures in Concept Lattice.

The computation of decision structures naturally contributes to reduce the computation time and also the number of computed concepts in spite of theoretical complexities. Figure 5 shows the using such structure is feasible. We have evaluated the number of computed concepts according to the available services and the size of the request. The request contains:

- No constraint, *i.e.* equivalent to compute the entire lattice,
- One functional constraint, *i.e.* the minimal use case because the functionnality is always known to select a service.

For this experimentation, we count the number of computed concepts from a context composed by 24 attributes (11 functionalities and 13 properties).

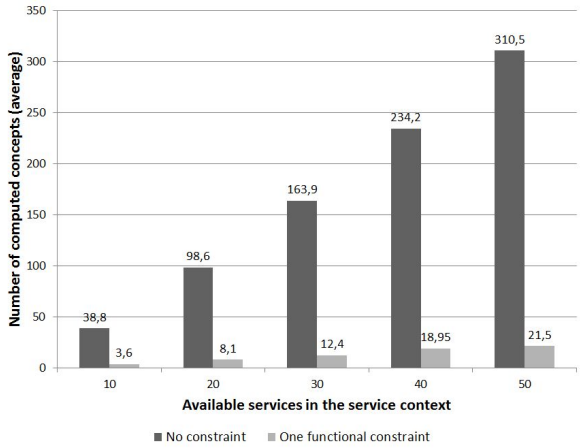

Figure 5. Size of decision structure.

We note that the computation of only interesting concepts (decision structure) largely decreases the number of computed concepts. For a request based on one functionality, the decrease is 92% in average.

C. Autonomic Dependency Management

At a given time, an application is executed in order to address a given goal. This goal can change and, as a result, it may be necessary to change one or several components or dependencies. It is the role of application manager and dependency managers to handle these evolutions. Figure 6 illustrates the relations between application manager and dependency manager.

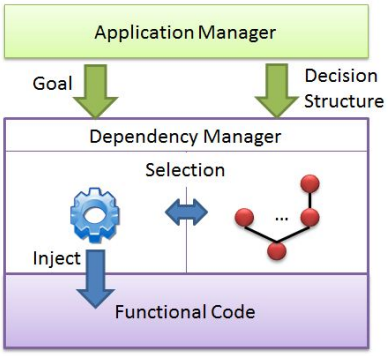

Figure 6. Component Details.

Application Manager. In our approach each application comes with its own application manager. It provides all services that can be used in the application, somehow playing the role of a local registry. The application manager use the RoSe ”Service Discovery” feature in order to get the list of available services. The expected functionalities to be searched are described using the RoSe descriptors. Thus, this local registry is a view of the available services filtered according to the current goals of the application. We propose to use the formal context approach to store the services characteristics. These characteristics are aligned with ontologies [11] and discretized to get numerical properties [12], [13]. The advantage of this definition is that the characteristics are defined by application. Consequently, one application can express that the cost is high even if it is more than 100$ while for another application a cost of 100$ is low. Each application is composed by a set of related services (components). For each dependency (represented in the form of a binding in OSGi), application managers generate a decision structure. The chosen level of granularity (dependency service) ensures that the decision structure is relevant (meaning and size). For each dependency, the global goal is transformed into a more specific goal sent to the dependency manager concerned.

Dependency Manager. Each service consumers (i.e. component or bundle) is endowed with a local dependency manager. The dependency manager is used to dynamically change the dependencies between components. To help make ”good” decisions, it has a goal and a structure of decision provided by the application manager. A goal is an ordered list of mandatory and optional properties. For instance, we

can express that we want a service providing temperature with a reduced cost and possibly with reliability characteristics and in DPWS technology. As previously explained, the decision structure allows to classify services and to group them into equivalent classes. To resolve a dependency, a dependency manager searches the decision structure and it can thus select the "best" service related to this goal.

Note that the evolution of the goal and the decision structure is independent. A goal change does not imply a new computation of decision structure. Consequently, this allows fast reactivity to the context modifications and to priority changes regarding the global goals of the application.

Dependency managers have been implemented by extending actual iPOJO dependency manager [5] as suggested by [6]. For legacy bundles (*i.e.* not using iPOJO) we relied on the new services hook provided by OSGi [4] – this way it is possible to control the service discovery.

V. Resources Management

The share of the same software execution environment by competing applications requires that the execution of software modules from an actor does not harm the execution of other modules. Hardware resources management has thus to be provided at software module level above the JVM. Given the wide range of deployable modules and their interactions, it is hardly possible to test exhaustively all the possible combinations and even a rigorous testing of every bundle before deployment is not sufficient. This problem is exacerbated by the use of remote services in the cloud whose evolution can have unpredictable consequences on the behavior of applications hosted on the platform. For instance, we had trouble on a fleet of platform when performances of a cloud services have improved significantly: some applications have been so overwhelmed that they caused CPU consumption problems.

Our approach is to build a self-optimizing monitoring system that can dynamically activate specify monitoring mechanisms when issues are detected. The platform manager tunes monitoring mechanisms accuracy and frequency and decides when to enable/disable monitoring mechanisms on-the-fly. By sparingly using resource-intensive monitoring mechanisms, it is possible to get the necessary accuracy while limiting the average resource consumption. This helps to detect performance issues with a minimal overhead in the long run. These information can then be used by an autonomic manager to take decisions, *i.e.*, stopping a bundle, changing implementation, or restarting a device.

In this paper, we focus on CPU usage monitoring. To achieve this goal, we have developed several mechanisms that are activated in the following order :

1) global load average and system CPU (M1): this mechanism is always activated and uses system calls to make the system compute load average. The collect frequency is increased depending on previous values.
2) CPU usage per bundle (M2): this mechanism provides an estimate of CPU usage per bundle. It is activated only when load average is high. This mechanism has been implemented following ideas described in [14]. The framework has been modified so that each new created threads are attached to the proper bundle.
3) building dependency graph (M3): when a suspect has been found, a dedicated task determines bundle dependencies. This information is used to determine the impact of uninstalling a bundle on other bundles.
4) monitoring service dependencies (M4): this mechanism uses service proxy injection to refine the analysis and try to determine the actual source of CPU load. Dynamically injecting proxies in OSGi raises some issues. It requires to force the consumer to release the providers it uses so that it uses proxies instead. However, according to the specification [15], once a consumer holds a reference on a service object, the only ways to force it to discard the services are (a) to stop the providers or (b) to stop the consumer. Stopping the providers has an impact on all their consumers and potentially on applications or the whole system. Stopping the consumer may also have an impact on other bundles and may lead to bundle state loss. We propose some modifications in the OSGi framework so as to create a proxy-aware registry. We take advantage of the loose coupling offered by the SOA by modifying some mechanisms and in particular the way bundles are notified of the arrival and departure of services. Basically, service binding monitoring implies three steps: (a) unbinding existing services by pretending they are no longer available, (b) pretending they are available again, (c) substituting original providers by proxies when consumers ask for them.
5) monitoring package dependencies (M5): the CPU consumption may be the result of the usage of a poorly coded library. Sampling is performed by an external mechanism observing system threads activity on a regular basis. The subset of threads to be monitored is determined using the M3 mechanism: the task monitors threads attached to the monitored bundle. At a fixed pace (10ms in our case), we request the JVM to generate the stack trace for monitored threads. This stack trace contains information on the stack frame. It is thus possible to infer the time spent calling a class by comparing stack traces. This gives an estimate of the time taken calling an outside package. This is then matched up with package dependencies. Usage statistics are then calculated.

The Figure 7 shows the average monitoring impact on CPU usage for different monitoring configuration on a system running typical bundles (50 bundles). Mechanisms (M1 to M5) are activated progressively from C1 to C5 respec-

tively. In C4 and C5, services and packages are monitored on a single bundle. In C6 all bundles are monitored. Overall, these results confirm that using progressive monitoring is generally more CPU-efficient than using always-enabled traditional monitoring systems. When idle, the impact of monitoring is way below the one with traditional systems. Most of the time, the platform is not charged by useless computations. When active, the impact is significant but localized on a single bundle. Additionnaly the overhead on non-monitored bundle is limited. The benefits of localization is visible when comparing C5 (local monitoring on a single bundle) and C6 (monitoring all bundles). Moreover, the difference is even more pronounced on a platform with more bundles. Therefore, always-enabled monitoring, done by many OSGi monitoring systems, is not reasonable on end-user systems.

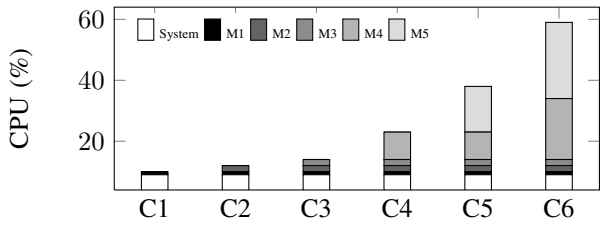

Figure 7. Guided progressive activation of monitoring reduces overhead.

With all the reported values, it is easy to build simple heuristic for the detection of CPU-intensive bundles. Automatically, the platform manager controls the activation of monitoring mechanisms and fires alerts to the concerned applications manager. The latter are then responsible for the interruption and replacement of suspect bundles. We tested these policies on simple applications on sample pervasive applications. The manager has been implemented using the Ceylon [16] framework. This proves to be effective to avoid platform crashes caused by CPU-intensive applications. A future work is to use the information provided by the system to automatically rank bundles and services and therefore influences the selection process depending on this ranking.

VI. Related Work

The problem of service selection, depending on service classification and FCA, has been studied by a few authors. Bruno et al. [17] propose an approach based on machine-learning techniques to support service classification and annotation. Peng et al. [18] classify Web Services in a concept lattice. Services are classified according to their functional operations regardless of non-functional aspects. Azmeh et al. [19] classify Web Services by their calculated QoS levels and composability modes. In these approaches the concept lattice is computed only once whereas in the pervasive domain, services regularly appear and disappear, which means recalculating the lattice. Moreover, these approaches can not manage simultaneously different technologies (UPnP, DPWS...). Ait Ameur [11] proposes to adapt the registry to a semantic registry in which semantic Web Services are stored. The introduction of ontologies allows to define a subsumption relationship between services that expresses a substituability relationship between these services. In our approach, ontologies can be added in the filter of the service registry in order to minimize the number of attributes in the context model.

Monitoring an OSGi-based platform is challenging [20]: the specification [15] does not define any means to isolate or monitor bundles. Existing Java tools (*e.g.*, JVM-Ti[2] or TPTP[3])cannot be used as is since gathered information is too fine-grained and thus not relevant. Existing OSGi tools are not suitable for embedded in-production environment because (i) they target development environments [14] or rich platforms [4], or (ii) they require heavy modifications of the JVM or underlying operating system [21], and (iii) they generally induces a persistent strong overhead of at least 20%. Our main contributions in the proposal of flexible monitoring mechanisms. In particular, we refine existing techniques to attach threads to bundles, (ii) propose a novel approach to inject proxies on-the-fly without stopping bundles by building a proxy-aware registry, (iii) proposes a method to monitor package dependencies by using localized sampling. The proposed system competes well with traditional monitoring systems: the overhead when idle is under 2% and is comparable when full active (20% on a typical system). Moreover, the overhead is localized: it mostly impacts the targeted bundles and has limited consequences on the others.

VII. Conclusion and Future Work

This article presents a framework dedicated to the execution of pervasive service-based applications and their automated management. This framework addresses three major overlapping challenges related to service-based pervasive environments: (i) the management of heterogeneous distributed services is performed by RoSe, an integration tool that generates the necessary bridges for the publication or integration of services, (ii) the autonomic service selection through the use of FCA to classify available services and through the addition of an autonomic dependency manager to each service consumers, (iii) finally the evaluation of applications and the autonomic resource management has been made possible via the use of several monitoring mechanisms that are activated on-purpose and on-the-fly by a platform manager. This platform is controlled by a hierarchy of managers that allows the division of administrative objectives into sub-goals and managing the preoccupations of applications and the platform. This platform has been implemented

[2]http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/
[3]http://www.eclipse.org/tptp/

using OSGi technologies and in particular via the extension of the associated service-oriented component model. Each contribution has been evaluated separately and the platform is currently being used in the MEDICAL project. Future works include the management of a fleet of platforms and the inter-platform communication between application managers and platform managers so as to coordinate the management of the multiple platforms.

REFERENCES

[1] M. Weiser, "Human-computer interaction," R. M. Baecker, J. Grudin, W. A. S. Buxton, and S. Greenberg, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, ch. The computer for the 21st century, pp. 933–940.

[2] M. P. Papazoglou, "Service-Oriented Computing: Concepts, Characteristics and Directions," in *Proceedings of the fourth International Conference on Web Information Systems Engineering*, Los Alamitos, CA, USA, December 2003, pp. 3–12.

[3] J. Bardin, P. Lalanda, and C. Escoffier, "Towards an Automatic Integration of Heterogeneous Services and Devices," in *Proceedings of IEEE Asia-Pacific Services Computing Conference*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 171–178.

[4] J. Ferreira, J. Leitão, and L. Rodrigues, "A-OSGi: a framework to support the construction of autonomic OSGi-based applications," *Int. J. Autonomous and Adaptive Communications Systems*, vol. 5, no. 3, pp. 292–310, 2012.

[5] C. Escoffier, R. S. Hall, and P. Lalanda, "iPOJO: An extensible service-oriented component framework," in *IEEE International Conference on Services Computing, 2007. SCC 2007*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 474–481.

[6] J. Bourcier, A. Diaconescu, P. Lalanda, and J. A. McCann, "Autohome: An autonomic management framework for pervasive home applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 6, no. 1, pp. 8:1–8:10, Feb. 2011. [Online]. Available: http://doi.acm.org/1921641.1921649

[7] B. Ganter and R. Wille, *Formal Concept Analysis - Mathematical Foundations*. Berlin, Heidelberg: Springer, 1999.

[8] L. Nourine and O. Raynaud, "A fast incremental algorithm for building lattices," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 14, no. 2-3, pp. 217–227, 2002.

[9] S. Chollet, V. Lestideau, P. Lalanda, Y. Maurel, P. Colomb, and O. Raynaud, "Building FCA-Based Decision Trees for the Selection of Heterogeneous Services," in *Proceedings of the 2011 IEEE International Conference on Services Computing*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 616–623.

[10] S. Chollet, V. Lestideau, Y. Maurel, E. Gandrille, P. Lalanda, and O. Raynaud, "Practical Use of Formal Concept Analysis in Service-Oriented Computing," in *Proceedings of the International Conference on Formal Concept Analysis*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 61–76.

[11] Y. Ait-Ameur, "A semantic repository for adaptive services," in *IEEE Congress on Services*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 211–218.

[12] G. Polaillon, "Interpretation and reduction of Galois lattices of complex data," in *Advances in Data Science and Classification*, Springer-Verlag, Ed. A. Rizzi and M. Vichi and H.-H Bock, 1998, pp. 433–440.

[13] Z. Assaghir, M. Kaytoue, W. Meira, and J. Villerd, "Extracting decision trees from interval pattern concept lattices," in *Concept Lattices and their Applications*, 2011.

[14] T. Miettinen, D. Pakkala, and M. Hongisto, "A method for the resource monitoring of osgi-based software components," in *Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference*, 3-5 2008, pp. 100 –107.

[15] "OSGi service platform core specification release 4 version 4.3," OSGi Alliance, Tech. Rep., April 2011.

[16] Y. Maurel, A. Diaconescu, and P. Lalanda, "CEYLON: A Service-Oriented Framework for Building Autonomic Managers," *Engineering of Autonomic and Autonomous Systems, IEEE International Workshop on*, vol. 0, pp. 3–11, 2010.

[17] M. Bruno, G. Canfora, M. D. Penta, and R. Scognamiglio, "An Approach to support Web Service Classification and Annotation," in *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05) on e-Technology, e-Commerce and e-Service*, ser. EEE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 138–143.

[18] D. Peng, S. Huang, X. Wang, and A. Zhou, "Management and Retrieval of Web Services Based on Formal Concept Analysis," in *Proceedings of the The Fifth International Conference on Computer and Information Technology*, ser. CIT '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 269–275.

[19] Z. Azmeh, M. Driss, F. Hamoui, M. Huchard, N. Moha, and C. Tibermacine, "Selection of Composable Web Services Driven by User Requirements," in *Proceedings of the 2011 IEEE International Conference on Web Services*, ser. ICWS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 395–402.

[20] C. Larsson and C. Gray, "Challenges of resource management in an OSGi environment," in *OSGi Community Event 2011, Darmstadt, Germany*, September 2011.

[21] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot, "I-JVM: a java virtual machine for component isolation in osgi," in *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2009, pp. 544–553.