Adaptive Monitoring of End-user OSGi-based Home Boxes

Yoann Maurel, André Bottaro, Radu Kopetz, Kouther Attouchi
Orange Labs Grenoble
28, Chemin du Vieux Chêne, 38243 Meylan
firstname.name@orange.com

ABSTRACT

In-production performance monitoring is required for dynamic and modular systems open to third-party applications such as the OSGi-based smart home box that home actors envision today. Existing approaches are not suitable for in-production monitoring as they generally induce a strong persistent overhead. This paper proposes a progressive and localized monitoring system that is able to dynamically activate/deactivate and tune the accuracy of monitoring mechanisms depending on detected performance issues. In particular, it proposes to build a proxy-aware service registry in order to inject proxies on-the-fly without stopping bundles and it advocates the use of localized sampling. Our evaluation shows that the overhead of our monitoring system is only 2% when idle and comparable with traditional systems when completely active (around 20%).

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics—Performance measures; D.2.11 [Computer Systems Organization]: Performances of systems—Measurement techniques, Reliability, availability, and serviceability

General Terms
Management, Performance, Design, Reliability

Keywords
self-adaptive, monitoring, OSGi, SOC, proxy, sampling, autonomic computing, smart home

1. INTRODUCTION

A new world of applications emerges in the home thanks to the growing variety of sensors and actuators available. Several application domains are considered, e.g., security, energy efficiency, comfort, ambient assisted living, multimedia communication. However, the Smart Home market is still divided into niche markets involving dedicated device manufacturers providing proprietary solutions.

One great challenge lies in the deployment and maintenance of a software environment embedded on a Home box open to third party applications. The openness of this execution environment calls for programming language simplicity and for a particular compromise between software component sharing and isolation. These reasons explain why the Smart Home ecosystem shows an overwhelming support for Java programming language and OSGi software modularity. This support is claimed by leading actors [20][23], and leading standardization organizations in this domain [4][1][2]. OSGi technology brings the required modularity, dynamics and performances: (i) it provides fine-grained sharing and isolation mechanisms between deployment units called bundles — bundles internal code is private and they interact via direct method calls through contracted service interfaces (Java interfaces); (ii) it enforces a service oriented approach [21] and supports the on-the-fly deployment, execution and removal of bundles; and (iii) inter-bundle calls cost no more than local method calls and the overall memory footprint of applications is limited.

We are developing such an environment on an inexpensive embedded box (figure 1) with constrained performances. It dynamically pairs devices provided by third-party manufacturers by supporting their seamless discovery on-the-fly. As the whole range of devices and applications is not foreseeable, the execution environment supports the runtime deployment of third-party signed bundles.

Figure 1: OSGi-based Home Box pairing devices and supporting third-party applications (bundles).

The openness to third party applications raises the level of needed security guarantees beyond what is provided by Java and OSGi, i.e., JVM native resource isolation, OSGi code sharing and isolation mechanisms. Sharing the same
software execution environment by competing actors requires that the execution of software modules from an actor does not harm the execution of other modules. Thus, hardware resource management has to be provided at software module level above the JVM. There is clearly a need to protect the box against poorly written bundles [12,22]. Given the wide range of deployable modules and their interactions, it is hard to test exhaustively all the possible combinations and even a rigorous testing of every bundle before deployment is not sufficient. In-production administration is required and, given the number of deployed boxes, administration should be autonomic [14].

However, achieving this vision first requires to develop efficient in-production monitoring. This is the scope of this paper. Monitoring an OSGi-based platform is challenging [15]: the specification [3] does not define any means to isolate or monitor bundles. Existing Java tools cannot be used as is because gathered information is too fine-grained and thus irrelevant. Existing OSGi tools are not suitable for embedded in-production environment because (i) they target development environments [17] or rich platforms [9], or (ii) they require heavy modifications of the JVM or underlying operating system [12], and (iii) they generally induce a persistent strong overhead of at least 20%.

The goal of this work is to build a monitoring system inducing the lowest monitoring overhead whenever possible. The idea is to use resource-friendly monitoring mechanisms (less than 1% of CPU overhead) most of the time and to trigger the more accurate monitoring mechanisms to monitor the suspected part of the system only when a problem is detected. This way the system will be almost unaffected most of the time. We have three constraints. First, the system must be compatible with any OSGi-based application without any legacy bundle modification. Second the modification of signed code is forbidden: otherwise guarantees given by editors will hold no more. We finally exclude the modification of the operating system or the JVM since maintaining specific versions is expensive and error-prone.

This paper proposes to build a progressive and localized monitoring system that can enable/disable monitoring mechanisms on-the-fly on any bundle depending on detected performances issues. The monitoring is guided by a controller that dynamically tunes the information granularity and accuracy to fulfill the autonomic manager’s needs. For that purpose, the paper proposes a manager architecture and discusses the necessary modifications to make monitoring mechanisms activable/deactivable and localized. In particular, it (i) refines existing techniques to attach threads to bundles, (ii) proposes 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.

This paper is organized as follows. Section 2 introduces the general architecture and discusses the design of each necessary monitoring mechanism. Section 3 describes the implementation of each mechanism and section 4 provides performance measurements. Section 5 describes related works. Section 6 concludes the paper and discusses future works.

2. ADAPTIVE MONITORING

This section first describes the principles and the architecture of a progressive monitoring system. Then, it discusses three important mechanisms used in our system for the monitoring of the threads of bundles and their service and package dependencies.

2.1 Principles and Architecture

Our approach is to build a self-optimizing monitoring system that can dynamically activate specific monitoring mechanisms when issues are detected. A controller 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. This information can then be used by an autonomic manager to take decisions, e.g., stopping a bundle, changing implementation, or restarting a device.

In this paper, we focus on CPU usage monitoring. To achieve this goal, we use a flexible autonomic manager architecture following the principles described in [16]. The idea is to decompose the manager into simple, dynamically deployable and activable modules called administration tasks. Each task is responsible for a dedicated simple management activity. Typically tasks will (figure 2): monitor specific system/context information (e.g., M1, M2), transform or aggregate information (e.g., aggregation), decide action to be taken (e.g., decision), and apply solutions (e.g., execution). Communication between tasks is direct and follows the service oriented pattern.

![Figure 2: Modular autonomic manager administering the box. The controller refines monitoring by enabling/disabling monitoring tasks.](image-url)

Task configuration, activation and collaboration is carried out by a controller. The controller is responsible for optimizing the manager. As explained in [16], the controller can act on any task — e.g., tuning monitoring tasks, replacing ineffective tasks, using competing analysis tasks. In this paper, we solely discuss the adaptation of monitoring tasks.
The manager is composed of several monitoring tasks specialized on gathering and computing particular system metrics such as system CPU usage, load average or service usage. The monitoring tasks offer an abstraction of the mechanisms provided by the underlying framework. They provide standard high-level interfaces to configure and manage the lifecycle of sensors and to aggregate reported information. This eases the development by limiting the modifications of the manager.

Collected data are aggregated by a dedicated task (aggregation task) into a model of the environment. This model describes a graph of bundles with their bundle properties, their package and service dependencies. It is inspired by the OSGi capability model [3]. This information is used by decisions tasks to identify suspect bundles.

The controller monitors information provided by decision tasks and the model to decide whether, how and when to increase or lower the accuracy of the monitoring by enabling, reconfiguring or disabling monitoring tasks. As a result, the set of activated monitoring tasks and the associated overhead vary at runtime. We use the following criteria to choose different mechanisms to be implemented in tasks and determine on-the-fly controller activation policy:

- **granularity**: the nature of monitored elements (objects, threads, bundles, applications, middleware, systems). One goal of the monitoring process is to raise the granularity level to allow meaningful interpretation of reported values. We promote mechanisms that provide information at the application or bundle level.

- **locality**: although side effects of monitoring cannot be completely avoided, parsimonious monitoring should favor localized observation and instrumentation to minimize the impact on other applications.

- **accuracy**: accuracy is necessary to determine whether a bundle consumes more than another during a given period of time. It can be leveraged to limit CPU usage.

- **activation cost/idle cost/execution cost**: we consider three costs for the monitoring. The instrumentation cost is the cost of activating the monitoring system. The idle cost indicates the impact of the monitoring strategy when idle, e.g., starting the JVM in debug mode has a persistent impact on performance. Execution cost refers to the cost of running the full monitoring system. Keeping the idle cost as low as possible ensures good system performance when no monitoring is required.

Based on these criteria, we define the following activation levels and monitoring tasks:

1. **global load average and system CPU** (M1): this task is always activated and uses system calls to make the system compute load average. The collect frequency is increased depending on previous values.

2. **bundle CPU usage monitoring** (M2): this mechanism, described in section 2.2, provides an estimate of CPU usage per bundle. This task is activated only when load average is high.

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. One reason may be the result of faulty interactions. This mechanism is described in section 2.3.

5. **monitoring package dependencies** (M5): the CPU consumption may be the result of the usage of a poorly coded library. The section 2.4 discusses ways to implement a monitoring task to identify such problem.

The realization of M1 and M3 is trivial: collected values are provided directly by the system or the OSGi framework. Next sections discuss conceptual choices to realize M2, M4 and M5. Implementation of these tasks is detailed section 3.2.

### 2.2 Monitoring CPU per Bundles (M2)

When system’s CPU is overloaded, the monitoring is refined by activating a dedicated mechanism in order to identify the responsible bundles. For this purpose, it is necessary to compute the CPU usage of each bundle. As Java provides the means to monitor individual thread CPU usage, e.g., via Java Management Extensions (JMX) [18], this typically requires attaching threads to bundles. Depending on how the matching between the threads and the bundles is made, the accounting of CPU usage varies. [17] identified two ways to perform it in an OSGi context:

- **direct accounting**: the resources consumed during service interaction are accounted to the service provider. In other words, the CPU used by a code that belongs to bundle A will be accounted to A, even if it is the bundle B that called this code through a public interface of a service.

- **indirect accounting**: all the resources consumed by the threads belonging to a bundle are accounted to this same bundle. Therefore in service interaction there is no resource consumption accounted to service providers.

There are advantages and disadvantages for each method. For instance, in the case of direct accounting, if a service is called in an endless loop, the CPU usage will be accounted to the service provider and not to the responsible user that executes the loop. Similarly, if the service is poorly implemented, in indirect accounting the user of the service will be unfortunately identified as the responsible. We further present different techniques to perform direct or indirect accounting on the OSGi platform.

#### 2.2.1 Instrumenting the bundles

The first obvious way to associate threads to bundles is to use aspect-oriented programming (or low level bytecode instrumentation libraries like [6]). It is then possible to search for every instantiation of the Thread class in the code of the bundles before deployment, and modify the code in order to give the threads a name that can be easily related to the bundle. There are however two main shortcomings to this approach. First, there is no way to capture the creation of threads that occurs in third party libraries that we do not modify or that are in the JDK. Moreover, the preferred way to create threads since Java 1.5 is no longer by
instantiating the Thread class but rather using the Executor Framework [5]. Therefore, without modifying the JDK, one could not capture the thread creation in Executor Framework. Second and more importantly, it is highly intrusive and irreversible: third party code would have to be modified in multiple places. As explained before the modification of signed code is hazardous and raises too much responsibility issues.

2.2.2 Modifying the framework

Another option is partially described in [17]. It is based on the use of ThreadGroup objects. Thread groups were introduced as a means to group threads together and allow to apply some Thread primitives to multiple threads at once. Threadgroups are organized in a tree structure, with system and main groups at the top of the tree. When a new Thread object is created, it is by default put in the group of the thread that spawned it, unless the developer explicitly indicates a new thread group. If it is the case, the new thread group becomes a child of the current thread group. In general, developers do not specify a thread group and all the threads in an application are typically in the main thread group.

The main idea is to perform the activation of each bundle by a thread that is in a specific ThreadGroup. This way, all the threads that will be further spawned will automatically belong to either this bundle-specific ThreadGroup, or to a child of it. Further on, using the methods of ThreadGroup class and JMX we can associate the threads with the bundles and monitor their CPU usage.

One way to do this is to modify the OSGi framework so that for every bundle activation a new activation-thread is created, with a ThreadGroup object that has a name derived from the name of the bundle. All the threads created during the activation will belong to this new thread group. This provides an indirect accounting: if a consumer C calls a method M on a provider P, and if the method M creates a new thread T during the call, then T is attached to C.

This modification is suitable for lifecycle methods, i.e., start and stop, that are typically called a few times during the bundle lifecycle by the framework. However, the framework is also responsible for triggering events – e.g., service changes or bundle changes – to listening bundles. Threads created during these calls should also be associated with the bundles. These methods are often ignored in existing work ([17], [9]) but they should also be modified since doing otherwise leads to the non-accounting of several threads.

The problem is that event listening methods are called frequently and creating one thread per event will induce a heavy overhead. A rigorous way to alleviate the problem would be to create one thread per bundle and reuse it for dispatching events. This way the cost of thread creation will be almost eliminated. However in our case, the cost of creating a thread per bundle still appears high. A way to improve the performance is to temporarily change the framework ThreadGroup, i.e., without spawning new threads so that every thread called during the dispatching of events is eventually associated with the listening bundle. Actual Oracle’s JVM does not provide such possibilities. It can however be realized using Java reflection on the Thread class. Given the performance gains, this method was chosen as presented in implementation (section 3.2).

2.3 Monitoring service dependencies (M4)

2.3.1 Motivation

Based on the previous results, it can be tempting to uninstall bundles consuming most CPU time. Doing so is effective when immediate response is needed. However, many times, this symptom-based response does not tackle the real issues and, in case the uninstalled bundle is not the actual culprit, another bundle will eventually start consuming too much CPU time. For instance, let P be a bundle providing a service S that is not running on a dedicated thread. If a consumer bundle C invokes S, resource consumption of S will be attached to C with indirect accounting. Typically, if S is poorly implemented, C will appear the source of the overhead. Uninstalling C will remove the overhead, but once another consumer uses S, the same problem arises. Symmetric arguments apply for direct accounting. For this reason, complementary information at the service and application level is required for the self-optimizing system to be able to analyze the interaction between bundles.

OSGi bundles mainly communicate using services. The main idea is to instrument bindings between a consumer and its providers to identify whether one or more providers are overloading the system. The goal is to obtain a comparable estimate of the time spent by the consumer calling the services. This implies surrounding every service call by measurement code, which can classically be achieved in three ways: instrumenting the provider, instrumenting the consumer or using proxies. We used proxies for several reasons. First, as explained modifying signed code is prohibited and instrumentation of providers and consumers should be avoided. Second, proxies do not require any modification on applications and can be easily automatically generated and injected/removed by modifying the framework as explained hereafter. Consequently when monitoring is deactivated, this approach induces no overhead. This way the overhead is located only on monitored bundles, which induces a globally lower overhead. It also provides individual values per service binding. This more detailed information enables the identification of suspect bundles more accurately.

2.3.2 Proxy-aware registry

Typically, bundles publish and subscribe to services. The framework provides means for providers to (un)register services and for consumers to search for them and track changes to service registrations. These features are provided by a central registry. The framework is dynamic: providers may register or unregister services anytime. To ensure that consumers always use valid services, they register service listeners. These listeners are notified of the arrival or departure of services matching a given filter.

OSGi and service component models such as OSGi Declarative Services[3] or IPOJO[8] 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 to 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.

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[3], 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.

To avoid these inconveniences, 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. These steps are refined by the sequence diagram (figure 3):

1. before monitoring activation, when the consumer C1 searches for services, the registry returns a list of providers and calls to services are direct.

2. after activation, the registry pretends there are no more services. This prevents C1 from binding to other providers during the unbinding process. This is mandatory to ensure that c1 is rebound to the same subset of providers after the unbinding process.

3. the registry pretends by sending a notification to c1 that providers in use are leaving. C1 consequently releases all services. Once done, C1 uses no more services. If dependencies were optional (e.g., log services), C1 continues its execution. Otherwise, it is waiting for substitutes.

4. the registry notifies that previously used services are coming back. C1 then retries to rebind to services by requesting providers.

5. instead of returning the providers, the registry generates and returns proxies. One proxy is created per service binding.

6. C1 therefore starts to use the proxies instead of services.

7. all future requests to services will return proxy objects instead of original service objects. The use of proxies is restricted solely to monitored consumers.

Deactivation of the monitoring follows a symmetrical process.

A proxy implements all the interfaces of the original service and pretends to provide strictly the same services. It forwards the requests to the original service and simply gathers statistics (time per calls, number of calls, mean time). The more information is collected, the bigger is the impact on performance.

The information provided by proxies helps to identify abnormal relations. In particular, it is possible to evaluate the time spent by the consumer’s threads in calling the code of a provider and the time of the service response. If those times are too long, the manager forbids the binding between the consumer and the faulty provider by providing a substitute when possible. If the CPU problem is not solved, the provider is placed under surveillance or is stopped. Two future works include (a) using this information to rank services and lower the probability to use a suspect provider and (b) sharing the information between several homes to refine the ranking.

2.3.3 Advantages and Limitations

This modification of the framework brings dynamism with on-the-fly activation/deactivation. This allows localizing the monitoring on a subset of bundles solely. Only a few bindings will be monitored and only a few applications will be affected by the monitoring. At best, if dependencies expressed by the monitored consumer are optional, the monitoring will have almost no consequence on its state. Moreover, proxies have no impact on bundle internal code execution and only affect inter-bundle calls; typically the overhead on the global application will be limited. This mechanism provides complementary information to previous methods: using both allows having direct and indirect accounting information.

The use of proxies is however not innocuous. First, it may affect bundle states: when its dependencies are mandatory, the monitored bundle will be in a pending state until proxies becomes available – and potentially the whole application. Nevertheless measures provided by proxies are necessary and using traditional approaches, the bundle would have been stopped and its state affected anyway – or worse, the proxies would have been always active. Second, the nature of information collected by proxies highly depends on the accuracy of information provided by the underlying system and JVM. Gathering accurate information may entail a
This approach relies on an assumption. Proxy injection requires that the monitored bundle listens to service registry events, that it correctly releases object references when a service is leaving and that it waits for registering services and binds to them after the release of previous services. This assumption is reasonable since it is a strong requirement of the OSGi framework. Besides many tools have been developed to check this constraint (see [10] for instance).

2.4 Monitoring package dependencies (M5)

CPU usage can also be the consequence of faulty shared libraries. A known technique is to use (time-)sampling. Sampling is performed by an external task observing system threads activity on a regular basis. The subset of threads to be monitored is determined using the M3 methods: the task monitors threads attached to the monitored bundle. At a fixed pace, the task requests the JVM to generate the stack traces for monitored threads. Each 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 information can be matched up with package dependencies. Usage statistics are then calculated.

The monitoring is not continuous and the accuracy highly depends on the interval between inspections. To be accurate enough sampling induces a persistent overhead (since it causes threads to be paused and involve computation). Collected information is much less precise than the one collected on services bindings and granularity is high. Furthermore the impact will be constant regardless of the number of outside calls to libraries. A bundle that does very little use of its libraries will be affected as much as a very dependent bundle is affected.

3. IMPLEMENTATION

The current implementation relies on Apache Felix, a popular open-source implementation of the OSGi specification.

3.1 Overall Architecture Implementation

The prototype implements the monitoring part of the manager and a naive analysis task. No decision is taken by the manager itself. The manager uses 7 tasks: (a) the five monitoring tasks described in the approach (global load average and CPU, bundle CPU usage monitoring, dependency graph analysis, service monitoring, package dependency monitoring); (b) an aggregation task that gathers all the information in a single model; (c) an analysis task that ranks bundles based on the model and triggers alerts when CPU-intensive bundles are detected. Communication is service-oriented: tasks publish and use services — this ensures a weak coupling between them. Each task has been implemented as an OSGi bundle and provides external configuration facilities.

When a change is detected and on a regular basis, the monitoring tasks push data to the registered listeners, e.g., the change of bundle properties and bundle dependencies. Monitoring data listeners are automatically discovered by monitoring tasks in the registry according to the WhiteBoard Pattern\(^1\). The data push frequency can be tuned by


the controller to adjust the monitoring quality and the monitoring overhead. Monitoring tasks are stateless and do not store information.

Data are aggregated in a single model held by an aggregation task. This task registers monitoring data listeners for every type of changes and holds a history of changes. The task provides a service to browse the model. The model is refined by the analysis task, which can add new information. For instance, the analysis task can (i) tag a bundle to indicate that it has been previously suspected or (ii) associate a rank to the bundle according to its reliability. This way all analysis tasks can benefit from the expertise of other tasks.

In this prototype, the manager has only one analysis task. Based on the model, it ranks the bundles: bundles with high rankings are considered guiltier than the others. The implementation is very simple. Ranking mainly depends on bundle CPU usage and bundle criticality - system's bundle have a lower ranking than the ranking of applications. This information is used by the administrator to take decisions. The analysis is highly coupled with the controller. The controller offers a service for the configuration of monitoring tasks. When more precision is required the analysis asks the controller for more information. The actual analysis is static and empirical. For instance, when load average of the past 5 minutes is above 1, the bundle CPU mechanism is activated (M2) and CPU usage is then monitored for a configurable time (2 minutes by default); if a CPU-intensive bundle is found, bundle dependencies are analyzed (M3). The activation of tasks M4 and M5 depends on bundle criticality and the number of dependencies. Future work will refine this analysis process.

The implementations of M1, M3, and M5 rely on Java and OSGi. It does not need framework modification. The monitoring of load average, memory and global CPU-usage (M1) is obtained by parsing Linux /proc files. The monitoring of dependencies (M3) relies on OSGi. When possible, information on dependency optionality is gathered — this information is provided by known OSGi-based component model automating service dependencies, e.g., OSGi Declarative Services [3] and iPOJO [8]. An optional service binding could be broken without stopping bundle main activity. As there is no way to be notified of the time a service is in use in the OSGi specification [3] (no hook for getSer-
vice), the dependencies are analyzed on a regular basis. The sampling tasks M5 rely on information provided by M2 and M3 and uses standard Java Thread.getStackTrace() calls.

The following subsections describe the necessary modification to the framework to implement bundle CPU usage monitoring (M2) and service monitoring (M4).

3.2 ThreadGroups modifications (M2)

Due to performance considerations, we chose to implement our approach using reflexion as mentioned in section 2.2. The bundle lifecycle management methods as well as event dispatching methods have been modified using the Java reflexion. We also modified two popular automatic service binding systems: Delarative Services[3] or iPOJO[8] which take precedence over the framework for the activation of their specific bundles. The figure 4 provides an example of a modification (modification of method start()).

Once the modifications in place, the accounting can be easily performed with JMX, which is included in Java SE platform since its 1.5 version. Using the ThreadMXBean ob-
formances were very poor. For this reason, proxy classes
prototype used Java Dynamic Proxy and reflection but per-
cated and error-prone code to monitor services. One first
are leaving and must not be used, there is no reason for the
reference. As the framework pretends that original services
One way to avoid this problem is to give the proxies a new
framework should continue to return original instance when
using and the instance that is leaving. For this reason, the
mapping between service instances and service references.
However, when the listener notifies a service change, it uses
dependency monitoring (M5). Then it shows the overhead of
adapted to fast application prototyping. This section first
4. RESULTS
All benchmarks were performed on a computer embed-
ding a 1GHz ARM processor and 512MB of RAM. This en-
vironment is slightly faster than the targeted one but well
adapted to fast application prototyping. This section first
provides measures of the overhead of using proxies for ser-
vice dependency monitoring (M4) and sampling for package
dependency monitoring (M5). Then it shows the overhead
of each monitoring configuration on a typical environment.

4.1 Proxies overhead
The table 1 evaluates the overhead of proxies for different
methods. Compared to direct method call, the overhead of
calling an empty method with no parameters is huge
(13.5x). However, this factor should be put into perspec-
tive since the cost is constant (3µs) for the same method
signature. This implies that calling a method performing
some work induces a significantly lower overhead compared
to time consumed inside the method. This is shown on the
third column: calling a method performing a simple opera-
tion (calling Math.sqrt(9)) is only 1.8x slower than direct
call. It is thus reasonable to think that the overhead will
be under 2x most of the time. As show in column 2, the
overhead when more parameters are involved is only slightly
higher (3.4µs). This might result of the optimization of the
work called ASM[6]. A generated proxy implements every
Java interface provided by the original service instance. One
attribute is used to store the proxy creation date and two at-
tributes are created for each method: one to count calls and
one to compute total time spent to call the method. This
allows gathering statistics per methods with very little over-
head; statistics are then easily derived. One limitation of
the current implementation is that it uses System.nanoTime()
calls to estimate the time of each call. The main conse-
quence of using System.nanoTime() is that the monitoring is
scheduling-dependant and error-prone. Reported values are
however comparable on a long period and a high number of
calls.

The proxy generator is modular and uses a transformation
chain. It is thus possible to change the proxy generation to
compute more or less information. For instance, one way to
significantly reduce the overhead is to make the proxy com-
pute the time of only a limited number of calls or compute
statistics once per n calls. In such case, the accuracy is af-
fected but remains sufficient if the number of call is high.
This improvement has however not been automated.

The proxy class is generated only the first time a proxy
is necessary for a specific service. The proxy class is loaded
into the provider’s classloader. This way, the proxy class is
automatically unloaded when the provider is uninstalled. In
that purpose, we modify the loadClass(...) method : when a
class Hello$Proxy is required, the classloader returns the
proxy implementation of the class Hello. An instance of
proxy is created and returned only when a bundle requires
to get a service instance. This avoids the creation of useless
instances. The unloading of proxies requires special care.
Many bundles do not use ungetService(...) when they are
releasing a service reference. For that reason and to avoid
keeping unused strong references to proxies, the framework
use weak references (of the WeakReference Java class). When
the proxy is not in use anymore, the reference is garbage
collected.

3.3 Proxies implementation (M4)
The proxy injection requires modifications of the frame-
work to (a) change the relation between bundles and the
service registry, (b) to generate and load proxy implementa-
tions, (c) to create/release proxies. To ease maintenance,
modifications are limited to a small number of classes (4 clas-
se classes changed, 5 classes added).

Modifications are required to hijack the registration pro-
cess. The registration of services listeners and the request of
services (e.g., getService(...) or getServiceReference(...) calls)
are modified in BundleContext class. Listeners are used to
send “false” register (respectively unregister) events to force
the services to bind (resp. unbind) to proxies (resp. origi-
nal services) when needed. Due to the distinction between
service references (a description of the service) and service
instances (the actual instance providing the service), close
attention must be paid to the unbinding process. Many bun-
dles keep and use Java service instances but do not keep the
mapping between service instances and service references.
However, when the listener notifies a service change, it uses
the service reference. The consumer thus needs to get the
instance (via getService(...)) to compare the instance it is
using and the instance that is leaving. For this reason, the
framework should continue to return original instance when
bundle request for original references (at least the first time).
One way to avoid this problem is to give the proxies a new
reference. As the framework pretends that original services
are leaving and must not be used, there is no reason for the
consumer to reuse them unless for unbinding.

Service proxies are dynamically generated. This is a re-
requirement since it is not reasonable to provide every service
provider with a proxy implementation. Proxy generation
is completely automated: there is no need to write dedi-
cated and error-prone code to monitor services. One first
prototype used Java Dynamic Proxy and reflection but per-
formances were very poor. For this reason, proxy classes
are generated using the popular bytecode engineering frame-
JVM for method calls and call stack setup — i.e., the call stack is most likely not fully rebuilt inside the proxy method. What should also be stressed here is that the impact of proxies highly depends on the coupling between bundles. Internal calls are unaffected and the global overhead for typical bundle is likely to be lower than those in the tests.

Compared to bytecode instrumentation (row 2), there is only a slight advantage to instrumentation: 12.5x against 13.5x for empty methods, 1.78x against 1.8x for √9 method. The result column 2 can also be explained by compiler optimization (stack setup is most likely measured by instrumentation and skipped by proxies). The method delegation cost is negligible for typical methods and using ASM for implementing proxies is performance efficient. This result validates the use of proxies that is less intrusive.

Overall, however, the uses of proxies induce a significant overhead and their systematic use is unreasonable. This is why they are used sparingly on suspect bundles only.

### 4.2 Packages monitoring overhead

The overhead of sampling mainly depends on the sampling frequency (we use 10ms) and the number of threads. An additional overhead is induced by the monitoring threads that perform necessary computations: package and bundle identification. The latter overhead was approximately 5% of CPU usage in our experiment. Several experiences showed that the influence of the number of threads is negligible when there are less than 10 threads per bundle. As shown table 2 the cost of using sampling is approximately 2x. However, contrary to proxies that induce an overhead for inter-bundle communications only, the sampling induces a persistent overhead no matter where the code is executed (inside or outside the bundle). Consequently the global overhead might be higher than with the previous method. Just as for proxies, sampling is used sparingly.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Empty method</th>
<th>√9 method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>240ns</td>
<td>590ns</td>
</tr>
<tr>
<td>Instrumented</td>
<td>3020ns</td>
<td>4720ns</td>
</tr>
<tr>
<td>Proxies</td>
<td>3250ns</td>
<td>4110ns</td>
</tr>
</tbody>
</table>

Table 1: Average cost of an inter-bundle call depending on the used monitoring techniques, the work done by methods and parameters.

### 4.3 Overall approach overhead

The CPU monitoring overhead depends on many factors and can hardly be evaluated independently of the targeted environment. For instance, as explained previously, the global overhead of proxies mainly depends on the coupling between bundles and on the work performed by methods. To be significant, the results depicted by the figure 5 were measured on a platform running a typical set of 35 bundles (including typical home automation applications, administration GUI and web services). The number of threads ranges from 5 to 15. In these conditions, the order of magnitude is more meaningful than the numbers themselves. To show the impact of monitoring, bundles were chosen to consume only a small and stable fraction of the processor (9±1%). For similar reason, each monitoring task was triggered manually and the average CPU usage for each configuration was computed on a significant period of time. Several experiences showed that the order of magnitude remains the same on production platforms.

Figure 5: Average monitoring impact on CPU usage for different monitoring configuration on a system running typical bundles. Mechanisms (M1 to M5) are activated progressively from C1 to C5 respectively. In C4 and C5, services and packages are monitored on a single bundle. In C6 all bundles are monitored.

When idle (C1), the monitoring system induces a very low overhead. The M1 task, computing the load average every second, consumes less than 1% of the CPU. C2 to C5 are the active monitoring configuration. When monitoring bundles’ CPU usage (C2), M2 task overhead remains quite steadily around 2% although it is linearly dependant on the number of threads. M3 task (gathering dependencies information) also induces an overhead around 2% and depends on system changes and number of bundles. Overall, using the first three monitoring tasks altogether (C3) induces a 5% overhead.

The real interest of the approach is shown by the pronounced gap between C3 and C4/C5. The use of proxies (C4) on a single bundle induces 10% overhead - this was expected given the preceding results on proxies performances and given that this bundle performs a lot of requests. The sampling induces a 17% overhead (C5). The difference with proxies is mainly due to the computations performed every 10ms, but it is also dependent on the number of threads (for sampling) and the coupling (for services). It is therefore hard to determine which of the two mechanisms (M4 or M5) induces the most overhead in general. To keep the global overhead under 20%, it is possible to activate these tasks separately (C3+M4 then C3+M5).

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 signifi-
cant but localized on a single bundle. As previously mentioned, the overhead on non-monitored bundles is limited. The benefit 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, performed by many OSGi monitoring systems, is not affordable on end-user systems.

5. RELATED WORK

As a result of its ever-growing popularity, many tools have been developed to support the monitoring of Java systems. We draw the same conclusion as [17] and [9] concerning the standard Java tools such as JMX[18] or JVM-TI [19]. All these solutions are designed to monitor low granularity elements: e.g., threads, classes, objects or methods. As such, these data are of limited interest and there is clearly a need to raise the abstraction to the primitive deployment unit in OSGi, services and applications. Moreover, most of existing tools require heavy instrumentation [13] and are not designed to be dynamically stopped or started. These tools however provide the necessary foundation for the techniques described in this paper.

The work presented by Miettinen et al. [17] and refined by [9] is probably the closest to our proposition. As in the present work they focus on CPU usage monitoring of bundles. The M2 mechanism described in section 2.2 is inspired by their work. As explained, they only focus on activation methods which are only a subset of methods that should be instrumented. Our work explains why and how to modify and implement efficiently the notification mechanisms. Interestingly the authors of both papers have modified the framework to introduce proxies. However, their implementation suffered many limitations and is conceptually different from the one advocated here. First, they do not ensure that a proxy can be injected and they define no way to properly unbind/rebind bundles. This implies that there is no way to monitor a bundle without stopping it and loosing its state – this has huge consequences on applications. Additionally, they offer no guarantee that the bundle will reuse the same provider after the activation of monitoring. Moreover, the gathering of monitoring data is not clear and it is not clear whether there is one proxy per binding or one proxy per provider. We advocate the use of one proxy per binding which provides more relevant information. Finally, the authors do not specify whether the proxies are generated automatically or not; this generation is required to be generic. The novelty of the method M4 described here, besides automatic generation, is to provide real support for the on-the-fly activation/deactivation of bindings monitoring without stopping bundles and loosing states.

Unlike the present paper that is targeting in-production environments, their solution has been made for development environment[17] only or rich platforms (Intel Core duo processor)[9]. There are also technical differences on proxy implementations. In their solution all calls performed by a proxy are made by a dedicated thread; their reported performances made us decide to use System.nanoTime that is less accurate but a lot faster. This is reflected by the overall cost of their approach that is claimed to be more than 20%, which shows how important it is to use proxy sparingly. This result shows the importance of the progressive approach advocated here for in-production environment.

Several projects share with this paper the ambition to bring autonomic management (such as [9] or [11]). These projects show the feasibility and importance of autonomic management for OSGi platform by providing complete autonomic managers implementing complete loops.

A disruptive approach can be found in [12]. The authors modify the JVM and name it ‘I-JVM’. They employ isolates (or Java processes) [7] to achieve the required bundle isolation while preserving the OSGi bundle interaction model, which relies on direct method calls. Every bundle runs in a dedicated isolate, which is built from a class loader and is associated a private copy of special JVM entities, i.e., static variables, strings and java.lang.Class objects. Isolates run in the same execution stack to enable thread migration between isolates in a single address space and to enable passing an object reference in inter-isolate method calls. Thread migration used is used in inter-isolate communication to make CPU usage and memory (direct) accounting possible. Memory accounting is performed by the garbage collector that charges object memory to threads. This approach solves three known OSGi issues: (i) JVM static variables, strings and Class objects are isolated, (ii) resource accounting is done on a per-bundel basis, (iii) bundle termination is finally made deterministic by implementing a mechanism similar to known thread.stop deprecated method.

However, this approach brings several overheads, mainly a CPU overhead on every intra-bundle and inter-bundle call (said to be respectively 14% and 16%), a CPU overhead on object allocation due to resource accounting (said to be about 18%), and a memory overhead calculated to be about 16% on known OSGi platforms. The overheads are substantial and are always turned on whereas they can be turned off in our approach. However, it can not be simply compared to the overheads of our approach since the approach delivers several features that our approach does not address. One main difference is that I-JVM features are implemented in the JVM itself – this requires to modify and maintain a full JVM . Similar approach can be found in [15].

OSGi Service Hooks specification [3] enables a privileged actor on the OSGi platform to filter the services that a bundle can request or is informed from the service registry. The actor must use a Find Hook to intercept services requests and filter the service references that are returned to clients. They must then use an Event Listener Hook to intercept and filter service events before they are delivered to the client. This specification could then be considered useful to proxy an existing service to specific service clients. However, specified mechanisms enables the proxying only in the case the client is not started earlier. In our case, service proxying is required for service clients that are running and their running make them suspects. Our proposition avoids the restart of the client - that would have been necessary with OSGi Service Hooks - by simulating a service removal. More guarantees are therefore given for service continuity.

6. CONCLUSION

We described the design and implementation of a progressive and self-adaptive monitoring tool for in-production OSGi systems. It enables lightweight monitoring of OSGi resources with a low overhead most of the time. Monitoring is guided by a controller, performed only when necessary, and selectively on suspect bundles or applications. To achieve this goal, we discuss existing monitoring techniques
for OSGi platforms and explain how to adapt them to be dynamically activable/deactivable. In particular, we propose to benefit from bundle loose coupling by modifying the OSGi framework registry so as to inject monitoring proxies dynamically with little incidence on bundles states. We also use localized sampling to monitor the use of libraries. Overall, the accuracy of collected information provides a strong basis for automatic detection of CPU-intensive code. This is required to automate the management of the platform. As shown by the evaluation, being able to easily and automatically disengage the monitoring system can greatly limit the impact of supervision on the long run. When idle, the overhead is under 2%, and when activated the system behaves like most competing traditional monitoring systems. Thanks to localization, the impact is mostly limited to the suspect bundles. In practice, the techniques presented here proved effective in identifying common problems – such as bugs or unoptimized implementations – on our platform that hosts over a hundred bundles.

The main limitation of these techniques is that they may not be applicable to other resources than the CPU (such as memory) without modifying the JVM or the operating system. Another limitation lies in the different actions that are available to solve problems – in particular, current JVM does not allow stopping threads. This, however, does not question the process of progressive monitoring that proves to be efficient.

We plan as future works to implement a complete autonomic management of the platform. This implies building effective analysis of the collected data and proposing heuristics to solve problems adequately. Another interesting development would be to use information gathered on many platforms to identify failure patterns.

7. REFERENCES


