Technical Report RT/33/2009

A-OSGi: A framework to support the construction of autonomic OSGi-based applications

João Ferreira
INESC-ID/IST
joao.elias.ferreira@ist.utl.pt

João Leitão
INESC-ID/IST
jleitao@gsd.inesc-id.pt

Luis Rodrigues
INESC-ID/IST
ler@ist.utl.pt

May 2009

Abstract

The OSGi specification is becoming widely adopted to build complex applications. It offers adequate support to build modular applications, where modules can be added and removed at runtime without stopping the entire application. This paper proposes A-OSGi, a framework that leverages on the native features of the OSGi platform to support the construction of autonomic OSGi-based applications. A-OSGi offers a number of complementary mechanisms for that purpose, such as: the ability to extract indicators for the performance of deployed bundles, mechanisms that allow to have a fine grain control of how services bind to each other and to gather this information in runtime, and support to a policy language that allows to define autonomic behaviour of the OSGi application.

Keywords: Autonomic Computing, OSGi, Service Oriented Computing

A-OSGi: A framework to support the construction of autonomic OSGi-based applications

João Ferreira
INESC-ID/IST
joao.elias.ferreira@ist.utl.pt

João Leitão
INESC-ID/IST
jleitao@gsd.inesc-id.pt

Luis Rodrigues
INESC-ID/IST
ler@ist.utl.pt

Abstract

The OSGi specification is becoming widely adopted to build complex applications. It offers adequate support to build modular applications, where modules can be added and removed at runtime without stopping the entire application. This paper proposes A-OSGi, a framework that leverages on the native features of the OSGi platform to support the construction of autonomic OSGi-based applications. A-OSGi offers a number of complementary mechanisms for that purpose, such as: the ability to extract indicators for the performance of deployed bundles, mechanisms that allow to have a fine grain control of how services bind to each other and to gather this information in runtime, and support to a policy language that allows to define autonomic behaviour of the OSGi application.

1 Introduction

The OSGi specification [22] (initials for the extinct Open Services Gateway initiative) defines a standardized component oriented platform for building Service Oriented Java^{TM} applications. OSGi provides the primitives and support that allows to build applications from small, reusable and collaborative components. The OSGi platform also provides the support for dynamically changing such compositions, without requiring restarts. To minimize the level of coupling, the OSGi provides a service-oriented architecture that enables components to dynamically discover each other for collaboration.

OSGi was first developed with a number of concrete application areas in mind, including ambient intelligence, automotive electronics, and mobile computing. However, its advantages made the technology appealing also to build flexible Desktop Applications [6], Enterprise Applications [21, 14] and also Web Applications [16, 23]. A key issue associated with the deployment and management of complex web applications is to ensure the performance of the application in face of changing workloads. The difficulties in forecasting accurately the demand and in estimating the interference among the deployed applications, makes the configuration of web applications a significant challenge [1, 25]. The concurrent execution of multiple OSGi bundles, possibly developed by different teams, that invoke each other in patterns which, due to the dynamics of the system evolution, are difficult to predict at design time, makes this challenge even more daunting.

Autonomic computing has emerged as a viable approach to manage complex systems such as the one described above [8]. The idea is that a system must own autonomic management components, able to offer self-configuration, self-optimization, self-healing and self-protection features to itself. The ability to adapt its own behavior in response to changes in the execution environment is the fundamental ability of an autonomic system. The OSGi platform, by allowing components to be removed, added, and replaced at runtime without stopping the system, is particularly appealing for building autonomic web applications.

This paper proposes, describes and evaluates A-OSGi, a framework to support the construction of autonomic OSGi-based applications. A-OSGi offers a number of complementary extensions to the basic OSGi framework that improve its autonomic capabilities. Namely, A-OSGi includes the following features: the ability to extract indicators for the performance of deployed bundles, mechanisms that allow to have a fine grain control of how services bind to each other and to gather this information at runtime, and support for the interpretation of a policy language, that allows to define the autonomic behavior of OSGi applications deployed over the A-OSGi framework.

The rest of the paper is organized as follows. Section 2 overviews related work. The design and implementation of A-OSGi is described in Section 3 and Section 4, respectively. The resulting system is illustrated and evaluated in Section 5. Section 6 concludes the paper, providing some pointers for future work.

2 Related Work

In this section we provide a brief description of the OSGi platform architecture. Then we describe the MAPE-K autonomic control loop in the context of the OSGi architecture and, finally, we present some works that have explored strategies to enrich the OSGi platform with mechanisms to assist in the creation of autonomic applications, for instance, by proposing adequate monitoring mechanisms.

2.1 OSGi Platform

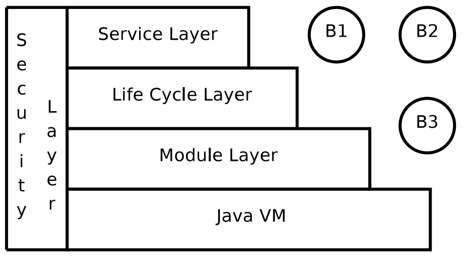


Figure 1. OSGi Architecture

The OSGi platform [22] provides support for deploying extensible Java-based applications composed by reusable modules, usually named *bundles*. The basic architecture of the platform is depicted in Figure 1. The platform allows to install, update, and remove bundles without stopping or restarting the system. Moreover, bundles are loosely coupled, and interact through service interfaces. In more detail, a bundle can register with the OSGi platform a number of services that it makes available to other bundles; the platform offers a service discovery mechanism that allows a bundle to dynamically find, at runtime, services that it requires to operate. The platform functionality is divided into the following four layers: i) The *Security Layer* extends the basic Java language security layer with a secure package format for bundles; ii) The *Module Layer* defines the modularization model employed by the platform, including the Java packages visibility and sharing rules among bundles, iii) The *Life Cycle Layer* provides the runtime support for bundles, including the mechanisms to install, update, remove, start, and stop individual bundles. It operates on top of the Module Layer; iv) The *Service Layer* owns the responsibility of providing the mechanisms to support the decoupling between service specification (interface) and implementation. Moreover, it allows individual bundles to register services provided by them as well as locate and bind bundles that provide other required services. The use of the service layer for bundle communication is recommended in the OSGi best practices.

The OSGi platform was initially oriented to embedded systems and network devices, however with its inclusion in the Eclipse IDE, OSGi is now widely used for both desktop and server applications [6, 21], namely for developing web applications [16, 23]. OSGi based applications have increased in complexity over the years, however the OSGi platform still lacks support for developing autonomic applications. Namely, the platform does not provide mechanisms to monitor the operation of individual bundles and to take advantage on distinct service implementations that potentially present different trade-offs between quality of service provided to the clients and resource consumption required to provide that service.

2.2 MAPE-K Control Loop

Many autonomic systems are modeled through a MAPE-K autonomic management control loop [9]. This loop consists on the following operations: monitoring (M), analysis (A), planning (P), and execution (E). The K stands for a shared knowledge base that supports these operations. We now provide a brief description of each MAPE-K component and discuss how they can be implemented in the context of the OSGi platform.

Monitoring The monitoring component is responsible for managing the different sensors that provide information regarding the performance of the system. In the OSGi context, sensors can capture the current consumption of critical node resources (such CPU and memory) but also other performance metrics (such as the number of processed requests per second and the request process latency). The monitoring granularity is the bundle. Sensors can also raise notifications when changes to the system configuration happen. Such sensors can be implemented using the notifications provided by the OSGi platform during the life cycle of bundles and services, and when clients bind and unbind to services.

Analysis The analysis component is responsible for processing the information captured by the monitoring component and to generate high level events. For instance, it may combine the values of cpu and memory utilization to signal an overload condition in the OSGi platform.

Planning The planning component is responsible for selecting the actions that need to be applied in order to correct some deviation from the desired operational envelope. The planning component relies on a high level policy that describes an adaptation plan for the system. These policies may be described using Event Condition Action (ECA) rules that are defined by a high level language. A ECA rule describes for a specific event and a given condition what action should be executed. In the context of OSGi, the actions may affect the deployed bundles and the bindings among these bundles.

Execution The execution component applies the actions selected by the planning component to the target components. In OSGi, we consider three main actions types, as follows: *i)* specify rules for service bindings, in such a way that a specific bundle is prohibited, or obliged, to use some specific service implementation; *ii)* change service properties, for instance change a parameter associated with a service implementation; and *iii)* control the life cycle of a bundle, by either starting or stopping bundles.

Knowledge Base The knowledge base component maintains information to support the remaining components. In the context of OSGi, it maintains information about managed elements, specificaly which services a bundle is using, which services a bundle provides, and other information about the dependencies among services.

2.3 iPOJO

One of the useful properties of OSGi, that can assist in developing autonomic applications, is the fact that the binding among different services can be established in runtime, to reflect the system dynamics. For instance, bindings may change when services become available or unavailable, as a result of bundle activation or deactivation. Still, the basic OSGi framework offers poor support to manage this dynamism. iPOJO [3] is a tool that is mainly aimed at simplifying the management of service dynamism and of non-functional service properties. iPOJO creates a clear separation between the business logic and the service oriented mechanisms, allowing business logic objects to be implemented as simple POJOs: **P**lain **O**ld **J**ava **O**bjects. This is achieved using a component container that manages all dynamism related to the business logic POJOs. In [3], the authors specifically apply the iPOJO solution over an OSGi platform. Although this approach can ease the management of services binding in runtime, unlike A-OSGi, it lacks the remaining components to build a MAPE-K control loop.

2.4 OSGI Monitoring

Several previous works have addressed the topic of monitoring OSGi applications [13, 5]. Most of these solutions have focused on providing an adequate per-bundle CPU consumption isolation. The work presented in [13] employs a thread-based approach to monitor each OSGi bundle, by creating threads that are internally associated with an individual bundle. Another approach can be found in [5], where the authors employ Isolates (or other execution environment objects) to achieve the required isolation (unfortunately, this solution only works in specific, modified, JVMs). Other tools could also be applied to monitor the resources, such as bytecode instrumentation for CPU accounting [7].

3 The A-OSGi Framework

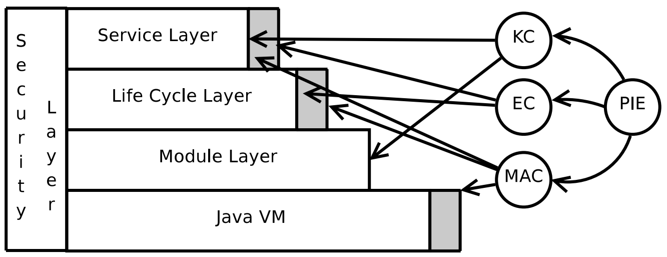


Figure 2. A-OSGi Architecture

The A-OSGi framework offers a number of extensions to the OSGi platform to support the development of autonomic applications. In this section, we provide an overall overview of the A-OSGi architecture followed by a detailed description of each of its components.

The A-OSGi architecture follows the general MAPE-K model (introduced previously in the Section 2.2). More specifically, we have augmented the OSGi platform with functionalities that support monitoring, analysis, planning, execution, and knowledge aspects of that model. As depicted in Fig. 2

these functionalities are provided by three main components, namely: A-OSGi Monitoring and Analysis component (MAC); A-OSGi Execution component (EC); A-OSGi Knowledge component (KC); and A-OSGi Policy Interpreter and Enforcer (PIE).

A-OSGi Monitoring and Analysis Component (MAC). The MAC component is responsible for retrieving information from sensors; it interacts with the OSGi service and module layers, as well as with the JVM. The MAC component monitors resource consumption, performance metrics, and changes to both bundle and service availability, as well as the binding of services by individual bundles.

Whenever the MAC detects a relevant change in the system, it generates an event to alert any interested component. Such events are routed to all components that have previously subscribed them. In our current architecture, only the PIE component subscribes all provided events. However, by exposing a publish-subscribe interface, we facilitate the extension of our architecture with additional functionalities.

The MAC component also runs any required analysis objects. Analysis objects subscribe the (low-level) events required to derive higher level notifications. In the current prototype, there is no explicit support to specify analysis objects using some form of domain specific language constructs: analysis components have to be programmed directly in Java. This pragmatic design choice allowed us to build a running prototype of the A-OSGi architecture that has been used to assess the merits of our approach. As future work we will enrich the analysis component, for instance, integrating previous work by others, such as the Event Distiller described in [11].

A-OSGi Execution Component (EC). The EC component is responsible for executing actions over bundles, individual services, and the OSGi kernel. Its interface exports the primitives that allow to start and stop bundles, change service binding rules in run-time (by adding or removing binding obligations and prohibitions), and also change properties of individual services (for instance by changing parameters associated with the operation of such services). In order to perform these actions, EC interacts with both the service and the life cycles layers of the OSGi architecture. In the current version of the architecture, only the PIE component uses the services of the EC component.

A-OSGi Knowledge Component (KC). The KC component provides a set of mechanisms that allow other components to consult information regarding the state of the A-OSGi execution evironment. In more detail, this component maintains, and exports, information concerning the set of installed bundles and registered services, and also on existing dependencies among bundles and services. To maintain such information available, the KC component interacts directly with the module and service layers of the OSGi architecture. In our current architecture the information maintained by the KC is accessed by the PIE component, which uses it to compute adaptation plans.

A-OSGi Policy Interpreter and Enforcer (PIE). The PIE component interprets the system policy, which is described by a set of ECA rules. The activity of PIE is driven by events received from the MAC component, that notify the need to perform adaptations. To select the best course of action, PIE uses the the information about the system provided by the KC component. As a result of its activation, PIE may request to the EC component the execution of one or more actions.

4 Implementation of A-OSGi

In this section we describe in some detail the implementation of A-OSGi architecture. The components of the A-OSGi architecture are implemented, themselves, as OSGI bundles. Naturally, these bundles need to be deployed to support the autonomic behavior of the OSGi system. However, some of the functionality required to implement these bundles requires small changes to the standard OSGi framework. More precisely, we had to augment the life cycle and service layers of the basic OSGI framework. These changes were necessary to support the monitoring and execution components of the MAPE-K cycle.

In the following paragraphs, we first enumerate the technologies that we have used to build our prototype of the A-OSGi framework and, subsequently, describe in more detail the implementation of each component.

4.1 Underlying Technologies

The OSGi specification has several implementations, some of the most well-know are: Eclipse Equinox [2], Apache Felix [4] and Knopflerfish [12]. For the work presented in this paper we have selected the Apache Felix 1.6.0 implementation. Notice however that changes performed over this implementation, and described in this paper, can easily be ported to other existing implementations. Other important component of our architecture is a HTTP server/container that permits the registering of resource and servlets to support the deployment of web applications. In this work we used the Pax Web [15] implementation of the OSGi HTTP service specification [23], that uses the Jetty HTTP Server [10]. The interfaces of the KC, EC, and MAC components are exported as JMX Managed Beans [17]. Thus, any existing JMX client (like [19, 20]) can use these components, and subscribe the MAC events, or invoke the KC and EC methods. This allows the services provided by these components to be used by third party components and even other applications. Moreover, the operation of the MAC component required the inclusion of a JVMTI Agent [18] at the JVM level. Finally, the PIE component is based on the Ponder2 policy interpreter for handling our ECA rules [24].

4.2 MAC Implementation

The MAC component monitors different aspects of the OSGi execution. Each of these aspects has its own specific requirements in terms of implementation. Namely:

- The MAC monitors the requests received by the HTTP server and stores information concerning the bundle in charge of processing the request. Therefore, it is able to provide information about the absolute number of requests processed by each bundle and the relative distribution of requests among bundles. It also stores the observed latency in the processing of each request. To implement such functionalities, the HTTP server bundle had to be changed in order to support the interaction between the entry point for requests in our architecture and the MAC.

- The MAC monitors CPU usage and memory consumption per bundle. In order to extract this information, some sort of isolation among bundles needs to be implemented. To implement our prototype, we used a thread based approach to achieve the isolation, by creating a hierarchy of *ThreadGroups* that associates a different *ThreadGroup* to each bundle. To create this hierarchy of threads, we have altered the life cycle layer of OSGi such that, whenever a bundle is started, the starting method is executed in a new thread from the *ThreadGroup* of that bundle. As a result, all threads created by the starting thread belong to the *ThreadGroup* associated with the bundle. Furthermore, clients of a service are provided with a proxy that executes the service methods in a thread associated to the bundle that registered the service.

  We are aware that the thread based approach used in the current prototype has a number of limitations. In first place, it has a non-negligible overhead as requires two context switch in each service invocation. Furthermore, it is unable to isolate interactions that do not use the service interfaces (such as when a bundle invokes directly methods of classes from another bundle). Finally, this approach may cause deadlocks in services with synchronized methods. Therefore, the approach requires a careful configuration of which services need to be isolated. Still, it its able to provide enough feedback to support the required information to implement many relevant autonomic behaviors.

Event Name	Event Attributes
CPUUsage	BundleID, value, oldvalue
MemoryUsage	BundleID, value, oldvalue
RequestsPerSec	BundleID, value, oldvalue
Latency	BundleID, value, oldvalue
BundleStarted	BundleID
BundleStopped	BundleID
ServiceRegistered	BundleID, ServiceID
ServiceUnregistered	BundleID, ServiceID
ClientRegistered	ClientBundleID, ServiceID
ClientUnregistered	ClientBundleID, ServiceID

Table 1. A-OSGi Context Monitor Event

Given that the problem of providing isolation among OSGi bundles is a challenging research topic on its own, we expect to incorporate in the future results from complementary on-going research[5].

With thread isolation, CPU usage can be calculated iterating over the threads associated to a bundle ThreadGroup and sum all the threads CPU time. The same approach can be extended to memory since its possible to detect the allocation of objects and assign allocations to the thread that is performing that operation.

- The MAC monitors notifications provided by the OSGi platform concerning the service registration and bundle life cycle. The binding between a client bundle and a service is monitored by leveraging on the iPOJO functionalities.

The complete list of events currently provided by the A-OSGi MAC is listed in Table 1.

4.3 EC Implementation

The EC component not only provides an interface to start and stop bundles (something that is directly supported by the standard OSGi implementation) but, more importantly, provides interfaces to control how bundles bind to each other and, as a result, to control which of multiple alternative implementations of a given service can, or should, be used. For that purpose, the EC offers the following mechanims:

bindings obligation: a binding obligation specifies that a bundle which operation requires a given service will be obliged to use a specific service implementation. The purpose of this mechanism is to force the use of a service implementation by a bundle.

binding prohibitions: a binding prohibition specifies that a bundle which operation requires a given service cannot use a specific service implementation. The purpose of this mechanism is to limit the use of service implementations by bundles.

service property configuration: the EC also provides support to change the value of a property associated to a service implementation. This functionality can be used to alter properties that the develloper of the bundle exposed as a service property.

The complete list of action supported by the EC component is listed in Table 2. In order to implement the EC component we have augmented the OSGi service layer. In A-OSGi, this layer maintains, for each bundle, the associated obligation and prohibitions. This information is used in run-time to ensure that bindings among bundles and services satisfy the constraints defined in each moment. We resort to iPOJO functionality to ensure the correctness of bindings.

Action Name	Parameters
StartBundle	BundleID
StopBundle	BundleID
SetClientProhibition	BundleID, ServiceID
RemoveClientProhibition	BundleID, ServiceID
RemoveClientProhibitionForServiceName	BundleID, ServiceName
SetClientObligation	BundleID, ServiceID
ChangeServiceProperty	ServiceID, Property, Value

Table 2. A-OSGi Actions

A-OSGi Bundle related functions		
Function	Parameters	Returns
getAllBundles		BundleID[ ]
getWebBundles		BundleID[ ]
getBundleName	BundleID	BundleName
getBundleID	BundleName	BundleID
getUsedServiceNames	BundleID	ServiceName[ ]
getUsedServiceIDs	BundleID	ServiceID[ ]
getUsedServiceIDsbyName	BundleID, ServiceName	ServiceID[ ]
getAllUsedServicesIDs	BundleID	ServiceID[ ]
getProvidedServiceIDs	BundleID	ServiceID[ ]
getProvidedServiceNames	BundleID	ServiceName[ ]
getUsingBundles	BundleID	BundleID[ ]
getAllUsingBundles	BundleID	BundleID[ ]
A-OSGi Service related functions		
Function	Parameters	Returns
getAllServices		ServiceID[ ]
getServiceName	ServiceID	ServiceName
getServiceNames	ServiceID	ServiceName[ ]
getServiceBundle	ServiceID	BundleID[ ]
getServiceImplementations	ServiceName	ServiceID[ ]
getUsingBundles	ServiceID	BundleID[ ]
getAllUsingBundles	ServiceID	BundleID[ ]
getAllUsingWebBundles	ServiceID	BundleID[ ]
getClientProhibitions	BundleID	ServiceID[ ]
getServiceProperty	ServiceID, Property	Value

Table 3. KC interface

4.4 KC Implementation

The KC provides a set of methods that allow to consult runtime information about the installed bundles and the registered services, as well the dependencies between the client bundles and services. To implement these functions, we use the module layer to extract information about services that a bundle is using and the service layer to extract information about the bundles being used by a service. The KC also provides methods to consult the current set of service obligation or prohibitions. The full interface of the KC component is listed in Table 3.

4.5 PEI Implementation

For implementing the PEI component we have used the Ponder2 policy interpreter [24]. Ponder allows to create Managed Objects that we use as adaptors to interact with the MAC, KC and EC components (using the corresponding JMX MBeans). To describe ECA rules, Ponder provides a language called PonderTalk. An example of a ECA rule described with PonderTalk, that prints a message whenever a bundle starts, is presented in Listing 1. The use of Ponder2 allows the dynamic definition of the policies, a property very usefull in a OSGi system due to the dynamic properties of the platform.

Listing 1. Pondertalk ECA rule

```
newpolicy := root/factory/ecapolicy create.
newpolice event: root/event/bundleStart;
  condition: [:bundleID | true];
  action: [:bundleID |
    root print: "Bundle id " + bundleID + " started".
    root print: "Bundle name is " + (bundles getBundleName: bundleID).
  ];
  active: true.
```

5 Evaluation

We now illustrate and evaluate the potential of A-OSGi to build autonomic OSGi-based applications. Our case study uses a Web Application that has been implemented using the architecture described in the previous Section, and that allows to demonstrate some of the main features of A-OSGi.

The set of OSGi bundles used by our application is depicted in Figure 3. We consider two web bundles that implement the presentation layer for an on-line store that sells CDs and DVDs. These web bundles are implemented as individual bundles that register with our altered version of the Jetty web server. Both web bundles allow remote clients to: i) list a sub set of products, available in the store and currently in stock, and ii) to get details for a specific product. Information about available items in stock is provided by a *stock service* that consults a local database. There are two (independent) bundles that offer this service with distinct tradeoffs between quality of service and resource consumption. In more detail, the first implementation of the stock service, simply named *Basic*, only resorts to the internal database to provide information about products.The second implementation of this service, named *Premium*, additionally relies on on a costumer preferences service, to order the product list according to the client preferences. Also, the premium service can offer suggestions about other products that may be of interest to the user and, therefore, returns additional items when the client searches for CDs or DVDs.

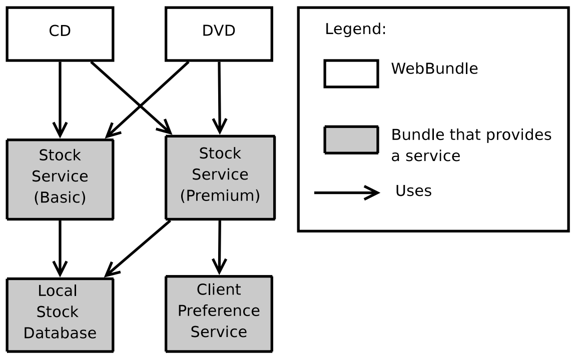


Figure 3. Case Study Components

The functionality provided by the Premium implementation, offering personalized content, can improve the costumer satisfaction and also generate more revenue to the store. Unfortunately, this additional quality of service comes at the expense of increased resource consumption. In situations where the server becomes overloaded with requests, it may be preferable to satisfy more requests, using the Basic implementation, than to provide the Premium service to a subset of clients and drop the remaining requests. Naturally, when the load allows, one would like to serve all requests using the Premium service. Furthermore, we would like to have the possibility of making these adaptations for each service independently of each other. For instance, if only the CD bundle is overloaded with requests, it may be possible to adapt only the stock implementation used by this service, and continue to use the Premium implementation for DVD buyers. As we will show, the A-OSGi architecture provides support to specify and implement this sort of policies.

5.1 Using A-OSGi

We now describe how A-OSGi can be used to implement the policy described above for our case study. The policy can be described by only two rules, depicted in Listing 2. The first rule simply prohibits any web bundle that is consuming more than 35% of cpu from using the Premium implementation of the stock service. The second rule removes this prohibition when a web bundle uses less than 5% cpu. The adequate thresholds for the cpu usage were determined experimentally. This policy ensures that the most expensive implementation is used, if and only if, the resources are enough to sustain the current load.

Listing 2. Policy

```
newpolicy := root/factory/ecapolicy create.
newpolicy     event: root/event/bundleCPU;
  condition: [:value :bundleID |
    usedstockservice := ((bundles getUsedServiceIDsbyName: \
      bundleID name: "pt.jjef.mediaportal.stock.service.StockService") at: 0).
    usedstockbundle := (services getServiceBundle: usedstockservice).
    stock1bundle := (bundles getBundleID: "pt.mediaportal.stock.Premium").
    (value > 35) & (usedstockbundle == stock1bundle) ];
  action:     [:value :bundleID |
    usedstockservice := ((bundles getUsedServiceIDsbyName: \
      bundleID name: "pt.mediaportal.stock.service.StockService") at: 0).
    services setClientProhibition: bundleID serviceID: usedstockservice.
    bundles stopBundle: bundleID.
    bundles startBundle: bundleID.
    ];
  active: true.

newpolicy := root/factory/ecapolicy create.
newpolicy     event: root/event/bundleCPU;
  condition: [:value :bundleID |
    usedstockservice := ((bundles getUsedServiceIDsbyName: \
      bundleID name: "pt.jjef.mediaportal.stock.service.StockService") at: 0).
    usedstockbundle := (services getServiceBundle: usedstockservice).
    stock2bundle := (bundles getBundleID: "pt.mediaportal.stock.Basic").
    (value < 5) & (usedstockbundle == stock2bundle) ];
  action:     [:value :bundleID |
    usedstockservice := ((bundles getUsedServiceIDsbyName: \
      bundleID name: "pt.mediaportal.stock.service.StockService") at: 0).
    services removeClientProhibition: bundleID serviceID: usedstockservice.
    bundles stopBundle: bundleID.
    bundles startBundle: bundleID.
    ];
  active: true.
```

Adaptation is performed with bundle-level granularity. The way the rules are specified does not require the CD or DVD web bundles to be named explicitly. Therefore, in run-time, depending on the system load, they may be applied to just the CD service, to just the DVD service, or both. This is possible because the KC component maintains updated information about each bundle, specifically on their bindings. Also, since A-OSGi offers the flexibility to choose which services should be monitored, in this case, it is possible to configure the platform such that only the CD and DVD services are monitored, reducing the monitoring overhead to a minimum. Run-time adaptation is performed by restarting the target of the rule. This forces iPOJO to reevaluate the bindings of the target bundle, taking into consideration the new set of rules in the system.

5.2 Performance

To evaluate experimentally A-OSGi we used a workbench composed of two Intel core-2 duo at 2.20 Ghz with 2Gb of memory. Both machines run Linux (Ubuntu 8.10 Desktop Edition) and the Sun Java Virtual Machine 1.6. Both nodes are connected by a 100 Mbit switch. We deployed A-OSGi in one of these machines, and loaded the policy depicted in Listing 2. The other machine is used to generate the workload using Apache JMeter 2.3.2 to emulate clients executing requests to the server. Clients operate by requesting a list of either DVDs or CDs from the server, and subsequently requesting details on one of the returned items.

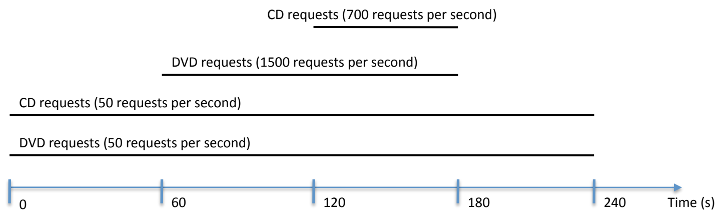


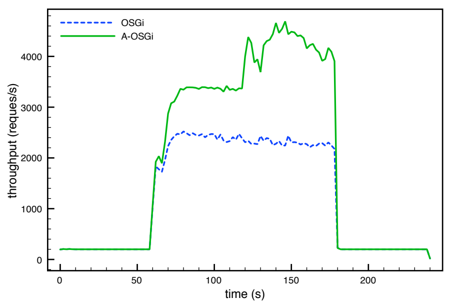
Figure 4. Workload Description

During the experiments the web application is subject to 3 different workloads that we have named, CD/DVD, CD/DVD+, and CD+/DVD+. The CD/DVD workload imposes 50 requests per second to the CD service and another 50 requests per second to the DVD service. This load is low enough such that the Premium implementation of the stock service can be used to answer all requests without overloading the system. The CD/DVD+workload, in addition to the previous requests, imposes a load of more 1.500 requests per second to the DVD service. To sustain this load, one is required to adapt the implementation of the stock bundle used by the DVD requests (CD requests do not need to be affected by the adaptation at this point). Finally, the CD+/DVD+ workload includes an excess of 700 requests per second to the CD service. At this point, both the DVD and CD requests are required to use the Basic implementation of the stock service to sustain the heavy load.

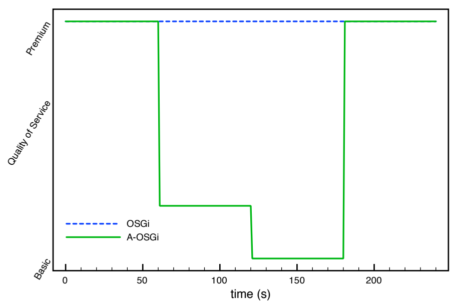
The system is initiated with the CD/DVD workload. At time 60 the workload is changed to the CD/DVD+ workload. Subsequently, at time 120 the workload is increased again to CD+/DVD+. Finally, at time 180 the workload returns to the baseline CD/DVD workload. Each individual workload was generated by a group of 10 client threads. These workloads are illustrated in Figure 4 (time is measured in seconds).

The results are depicted in Figure 5. The first plot compares the performance of a static configuration (providing the premium service) against the autonomic configuration. The adaptations that result from execution the policy can be inferred by the quality of service provided to the user in plot 5(b). Clearly, the autonomic configuration is able to ensure a much better throughput than the static configuration, by dynamically changing to the less expensive implementation of the stock bundle. Plot 5(c) depicts the total number of requests processed by both configurations. This last plot makes clear that the autonomic version responds better to the increase in the workload.

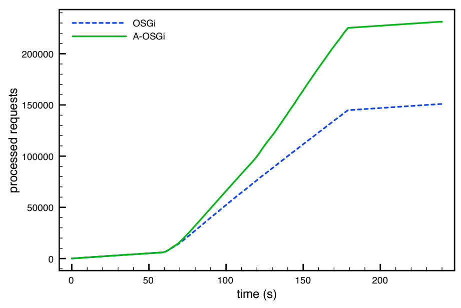
Finally, plot 5(d) compares the average request latency of the application running in the A-OSGi framework against the same application, under the same medium workload, running in a plain OSGi



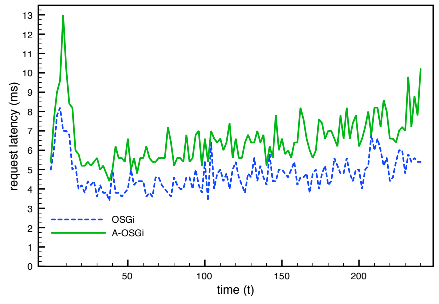
(a) Throughput



(b) Quality of Service



(c) Processed Requests



(d) Overhead

Figure 5. Performance with and without adaptation

framework. This allows to assess the overhead induced by the current implementation of the A-OSGi mechanisms. The difference is in the order of 25%, which is not surprising, given that many of the A-OSGi components are not yet fully optimized (in particular the isolation mechanisms required for detailed monitoring).

5.3 Other Policies

Due to lack of space, we have only discussed and evaluated one of the several policies that could be applied to the case study. However, we would like to point out some other alternatives that would also be supported by the A-OSGi framework. Alternatively or in addition to commuting between the Basic and Premium implementation, the policy could also configure the operation of each of these implementations (for instance, by changing the number of recommendations returned to the client by the Premium service). This would require to write rules specific for each bundle implementation, a feature that our simple case-study does not illustrates. Also, instead of setting individual binding constraints, the global behavior of the system could be controlled by simply installing or uninstalling bundles on the fly.

6 Conclusions

In this paper we have proposed A-OSGi, a framework that augments the OSGi platform to support the implementation of autonomic OSGi-based applications. A-OSGi offers a number of complementary mechanisms to this end, including the ability to extract performance indicators about the execution of deployed bundles, mechanisms that allow to have a fine grain control of how services bind to each other, and support to describe the the autonomic behavior of the OSGi application using a policy language.

The architecture has been implemented. Experimental results have illustrated the benefits of the approach: we were able to selectively adapt the implementation of a bundle used by different services, in order to augment the system performance in face of dynamic workloads. As future work, we plan to study ways to optimize the performance of some of the A-OSGi components, such as the MAC (by using more efficient isolation techniques), to reduce the overhead imposed by the autonomic mechanisms.

References

[1] Yixin Diao, N. Gandhi, J.L. Hellerstein, S. Parekh, and D.M. Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pages 219–234, 2002.

[2] Eclipse Equinox. Homepage. `http://www.eclipse.org/equinox/`.

[3] C. Escoffier, R.S. Hall, and P. Lalanda. ipojo: an extensible service-oriented component framework. pages 474–481, July 2007.

[4] Felix Apache. Homepage. `http://felix.apache.org/`.

[5] N. Geoffray, G. Thomas, C. Clément, and B. Folliot. Towards a new Isolation Abstraction for OSGi. In *Proceedings of the First Workshop on Isolation and Integration in Embedded Systems (IIES 2008)*, pages 41–45, Glasgow, Scotland, UK, April 2008.

[6] O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The eclipse 3.0 platform: Adopting osgi technology. *IBM Systems Journal*, 2005.

[7] Jarle Hulaas and Walter Binder. Program transformations for light-weight cpu accounting and control in the java virtual machine. *Higher Order Symbol. Comput.*, 21(1-2):119–146, 2008.

[8] IBM. Autonomic computing: Ibm's perspective on the state of information technology. *IBM Journal*, 2001.

[9] IBM. An architectural blueprint for autonomic computing, fourth edition. Technical report, IBM, 2006.

[10] Jetty HTTP Server. Homepage. `http://www.mortbay.org/jetty/`.

[11] G. Kaiser, J. Parekh, P. Gross, and G. Valetto. Kinesthetics extreme: an external infrastructure for monitoring distributed legacy systems. *Autonomic Computing Workshop, 2003*, pages 22–30, June 2003.

[12] Knopflerfish. Homepage. `http://www.knopflerfish.org/`.

[13] T Miettinen. *Resource monitoring and visualization of OSGi-based software components.* PhD thesis, VTT Technical Research Centre of Finland, 2008.

[14] OW2 Consortium. Jonas - White Paper v1.2. `http://wiki.jonas.objectweb.org/xwiki/bin/download/Main/Documentation/JOnAS5_WP.pdf`, 2008.

[15] Pax Web. Homepage. `http://wiki.ops4j.org/display/paxwev/Pax+Web/`.

[16] Spring Source. Spring Dynamic Modules for OSGi. `http://www.springsource.org/osgi`, 2009.

[17] Sun Microsystems. Java Management Extensions. `http://java.sun.com/javase/6/docs/technotes/guides/jmx/index.html`.

[18] Sun Microsystems. Java Virtual Machine Tools Interface. `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`.

[19] Sun Microsystems. Jconsole. `http://java.sun.com/javase/6/docs/technotes/guides/management/index.html`.

[20] Sun Microsystems. Visualvm. `http://java.sun.com/javase/6/docs/technotes/guides/visualvm/index.html`.

[21] Sun Microsystems. Sun GlassFish Enterprise Server v3 Prelude Release Notes. `http://docs.sun.com/app/docs/coll/1343.7`, 2008.

[22] The OSGi Alliance. OSGi Service Platform Core Specification, Release 4, Version 4.1. `http://www.osgi.org/Download/Release4V41`, 2007.

[23] The OSGi Alliance. OSGi Service Platform Service Compendium, Release 4, Version 4.1. `http://www.osgi.org/Download/Release4V41`, 2007.

[24] K. Twidle, E. Lupu, N. Dulay, and M. Sloman. Ponder2 - a policy environment for autonomous pervasive systems. pages 245–246, June 2008.

[25] R.D. van der Mei, R. Hariharan, and P.K. Reeser. Web server performance modeling. *Telecommunication Systems*, 2001.