INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

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

João Tiago de Jesus Elias Ferreira

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Professor Doutor José Manuel Nunes Salvador Tribolet
Orientador:	Professor Doutor Luís Eduardo Teixeira Rodrigues
Vogais:	Professora Doutora Maria Antónia Lopes

Outubro 2009

Acknowledgements

I wish to thank my family, friends and colleagues.

I wish also to thank my advisor Professor Luis Rodrigues for his commitment in the production of the work described here. I am also grateful to João Leitão for the helpful discussions regarding the contributions of my work, his help when preparing the Autonomics 2009 paper, and his comments on earlier versions of this manuscript.

Finally, I would also like to thank the Apache Felix community for the support and the help provided in the clarification of my questions regarding the OSGi implementation. The fast response and the ability to patch Ponder2 toolkit provided by Kevin Twidle was also very valuable.

This work was partially supported by FCT, through project Pastramy, PTDC/EIA/72405/2006.

Lisboa, Outubro 2009

João Tiago de Jesus Elias Ferreira

Resumo

A especificação OSGi está a ser cada vez mais utilizada na construção de aplicações complexas. Esta fornece o suporte para a construção de aplicações modulares, através da composição de módulos reutilizáveis e auto-contidos, denominados *bundles* ou pacotes. Os pacotes podem ser adicionados e removidos em tempo de execução, sem ser necessário parar toda a aplicação. O suporte para arquitecturas orientadas ao serviço fornecido pela plataforma OSGi permite que estes pacotes estejam fracamente ligados.

Aplicações complexas têm sido construídas sobre a plataforma OSGi sendo compostas por um número elevado de pacotes com múltiplas interdependências entre si. Embora o OSGi facilite a gestão destas aplicações (devido a modularidade introduzida), esta tarefa é ainda complexa e lenta.

Esta dissertação apresenta A-OSGi, uma bancada que assenta nas capacidades nativas da plataforma OSGi para suportar a construção de aplicações autonómicas, isto é, aplicações que apresentam capacidades de auto-gestão, baseadas no OSGi. A bancada A-OSGi oferece um conjunto de mecanismos para este fim: a possibilidade de recolher indicadores de desempenho dos módulos instalados, controlar a forma como os diferentes módulos se ligam a serviços, recolher informação em tempo de execução sobre a ligação entre os pacotes e interpretar uma linguagem que descreve as políticas que definem o comportamento autonómico da aplicação.

Um protótipo da bancada A-OSGi foi desenvolvido de forma a ilustrar as capacidades da arquitectura, tendo para tal sido desenvolvido vários mecanismos com base em tecnologias e ferramentas existentes. O protótipo foi avaliado com um caso de estudo onde a aplicação foi dinamicamente adaptada em função de uma carga variável. Os resultados mostram que a aplicação obteve um melhor desempenho em comparação com a aplicação sem adaptação.

Abstract

The OSGi specification is becoming widely adopted to build complex applications. It offers adequate support to build modular applications, based on the composition of small, reusable and self-contained modules, called *bundles*. *Bundles* can be added and removed in runtime without stopping the entire application. Bundles can be loosely coupled by leveraging on OSGi service-oriented architecture support.

Complex applications have been built on top of OSGi, resulting from the composition of a large number of bundles with many inter-dependencies. Although OSGi facilitates the management of a complex application (because of the modularity introduced), this is still a complex and time-consuming task.

This dissertation presents A-OSGi, a framework that leverages on the native features of the OSGi platform to support the construction of autonomic OSGi-based applications. Autonomic applications present self-management capabilities, that ease the execution of the management tasks. A-OSGi offers a number of complementary mechanisms for this 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 bundles bind to services and to gather this information in runtime, and a policy language that allows to define the autonomic behavior of the OSGi application.

A prototype of A-OSGi has been implemented to illustrate the capabilities of our architecture. To this goal, we have implemented a number of mechanisms that integrate several existing technologies and tools. The prototype was evaluated with a proof-of-concept case study, where the application is adapted in face of a changing workload. Results show that the application exhibits a better performance when compared with the execution of the application without adaptation.

Palavras Chave
Keywords

Palavras Chave

Computação Autonómica

OSGi

Computação Orientada ao Serviço

Keywords

Autonomic Computing

OSGi

Service Oriented Computing

Index

List of Figures

List of Tables

Acronyms

API	Application Programming Interface
A-OSGi EC	A-OSGi Execution Component
A-OSGi KC	A-OSGi Knowledge Component
A-OSGi MAC	A-OSGi Monitoring and Analysis Component
A-OSGi PIE	A-OSGi Policy Intepreter and Enforcer
ECA	Event-Condition-Action
EJB	Enterprise Java Beans
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JAR	Java Archive
JMX	Java Management Extensions
JSR	Java Specification Request
JVM	Java Virtual Machine
JVMTI	Java Virtual Machine Tools Interface
MAPE-K	Monitoring, Analysis, Planning, Execution, Knowledge
OSGi	Open Services Gateway Initiative
POJO	Plain Java Old Object
SOA	Service Oriented Architecture
XML	Extensible Markup Language

1 Introduction

The OSGi specification (OSGi Alliance 2007a) defines a standardized component oriented platform for building service oriented Java^{TM} applications from the composition of reusable and self-contained modules, called *bundles*. OSGi provides the primitives and the runtime support that allows the implementation of these applications. The OSGi platform also provides the support for dynamically changing such compositions (for instance, by starting, stopping and updating individual bundles) without being necessary to restart the entire application. To minimize the level of coupling, OSGi provides a service oriented architecture that enables bundles to register and dynamically discover services for use, establishing relations among the several bundles.

OSGi was initially developed for embedded systems software and later automotive electronics. However, its advantages made the technology an appealing foundation to build more complex applications for other areas such as Desktop Applications, Enterprise Applications and also Web Applications.

1.1 Motivation

OSGi applications present very dynamic properties, since its components (i.e bundles) can be started and stopped at any time, and, therefore, the services they provide may also become available or unavailable at any time. When these applications are composed by a large number of bundles and services, this dynamism may be difficult to manage.

Another key issue associated with the deployment and management of such complex 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 components, makes the configuration of applications a significant challenge (Diao, Gandhi, Hellerstein, Parekh, & Tilbury 2002; van der Mei, Hariharan, & Reeser 2001). The dynamic nature of the OSGi platforms, making difficult to predict at design time the bundles to be deployed in runtime, as well as the interaction patterns between them, turns this challenge even more daunting.

Autonomic computing has emerged as a viable approach to manage complex systems such as the ones described above (IBM 2001). It advocates that a system must own self-management components, able to offer self-configuration, self-optimization, self-healing and self-protection features. 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 applications.

1.2 Contributions

This dissertation proposes A-OSGi, a framework to support the construction and execution 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 bundles 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.

1.3 Results

A prototype of the A-OSGi framework has been implmented. The A-OSGi prototype uses a number of technologies including an OSGi platform and a policy interpreter toolkit, that integrated constitute the A-OSGi framework.

The prototype has been evaluated in order to demonstrate the benefits of A-OSGi architecture and mechanisms. The evaluation was performed using a proof-of-concept case study of an online music store, performing adaptations over the applications in face of changing workloads. The experimental results collected show that our framework is able to support the expression of autonomic behaviors that adapt the application with a bundle level granularity without incurring in an excessive overhead.

1.4 Research History

This dissertation work was performed in the Distributed Systems Group (GSD) at INESC-ID Lisboa. The work was initiated with an Autonomic Computing survey and familiarization with OSGi technologies. The initial objectives were to provide a fine grained monitorization of the OSGi platform and the support for the description of adaptation plans using ECA rules. However we noted that the available actions supported by the OSGi platform were not sufficient to perform adaptations over the OSGi platform and applications. As a result we designed and implemented the prohibitions and obligations mechanism, that allows a better control of the bindings of service oriented OSGi applications.

During my work I enjoyed from the fruitful collaboration with João Leitão and the remaining GSD team members, namely José Mocito, Nuno Carvalho and Liliana Rosa.

The results of this dissertation were published in (Ferreira, Leitão, & Rodrigues 2009).

1.5 Dissertation Structure

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

2 Related Work

In this chapter we provide a brief description of the OSGi platform and its services specification. Then we describe the Autonomic Computing initiative, its origins and objectives, as well a reference model to implement autonomic elements named MAPE-K loop. We describe each of its components, presenting example technologies used to implement them, and some relevant works that have addressed the implementation of the complete MAPE-K loop.

Later we describe some key concepts that explain the performance bounds of a computing system, and how these limitations can be tackled. Finally, we present related works that have explored strategies to enrich the OSGi platform with mechanisms to assist in the creation of autonomic applications or autonomic properties, for instance, by proposing adequate monitoring mechanisms.

2.1 OSGi Platform

The OSGi platform (OSGi Alliance 2007a) is a container supporting the construction and deployment of extensible Java-based applications composed of reusable and self-contained components, usually named *bundles*. The platform allows to install, update, and remove bundles without stopping or restarting the system. Moreover, bundles can be loosely coupled, through the usage of a service oriented architecture, where bundles interact in a publish/find/bind service model. 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. Moreover, OSGi specifications define a number of standard services that different vendors can implement such as a log service, a HTTP service, and a event service (OSGi Alliance 2007b).

The OSGi platform was initially developed to support applications for embedded systems and network devices, providing a thin kernel for the remote deployment of applications. However

OSGi met its greatest expansion when the Eclipse IDE [1] adopted it as the base for its plugin architecture (Gruber, Hargrave, McAffer, Rapicault, & Watson 2005). OSGi is now widely used for both desktop and server applications (Eclipse and a number of different application servers such as Jonas (OW2 Consortium 2008) and Glassfish (Sun Microsystems 2008)), and also for developing web applications (Spring Source 2009; OSGi Alliance 2007b) as evidenced by the creation of expert groups that address these areas to contribute to the future of the OSGi specification.

2.1.1 OSGi Architecture

The OSGi platform presents a layered architecture (where a layer can use all the layers bellow it) as depicted in Figure 2.1.

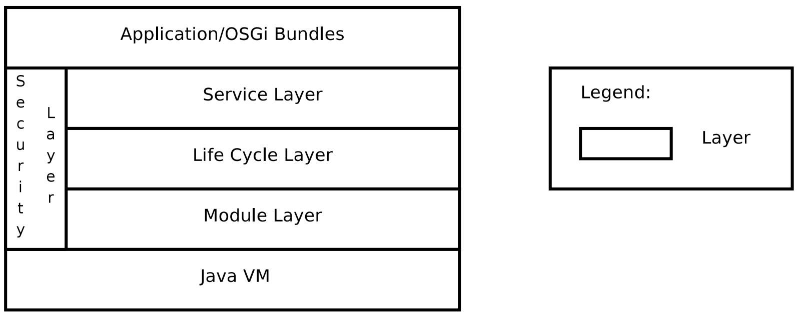

Figure 2.1: OSGi Architecture

The architecture is divided in the following layers:

- The OSGi platform executes in a *Java Virtual Machine*. The current specification require that the platform implementations are Java 1.4 compatible, although a prototype exists that incorporates Java 5 features in the platform such as generics and annotations (Hargrave & Kriens).
- The *Module Layer* defines the modularization model employed by the platform, including the Java packages visibility and sharing rules among bundles. Unlike other solutions that do not allow the sharing of classes between bundles (packages are all bundle private like in EJB (Enterprise Java Beans) (Sun Microsystems 2006a)), in OSGi classes can be shared between bundles allowing bundles to contain libraries for other bundles and

[1] `www.eclipse.org`

reducing the memory consumption. However sharing classes raises a number of problems such as dependencies between bundles and different bundles providing the same class.

- The *Life Cycle Layer* provides an API to support the management of bundles. This API includes the runtime support to install, start, stop, update, and uninstall bundles. A bundle is installed from a JAR (Sun Microsystems 2003). The JAR's manifest headers describe a bundle properties such as bundle name, exported and imported packages and others as defined in the OSGi specification. Before a bundle can be started it must be *resolved*, i.e, its package dependencies must be analyzed and binded in order for the bundle to start. This process may trigger the resolving of other bundles. After the bundle is resolved its start method is called.

  During the life cycle of a bundle a lot of events can happen in the platform such as other bundles starting, stopping, being installed or removed. Such events can break the dependencies of the bundle. For example the exporter of a package is uninstalled. When this event happens, the uninstalled bundle exported packages remains available as long as there are other bundles depending on them. However a *refresh* action can be called, that stops all bundles depending on those packages, removes the dependencies, resolves and restarts the bundles. This mechanisms frees the bundle from being concerned with the change of its imported and exported packages as it is always stopped before the dependencies are changed and is started to use the new version of the dependencies.

- The *Service Layer* has the responsibility of providing mechanisms to support a service-oriented architecture (SOA) on top of the OSGi platform. This SOA support allows programmers to develop loosely coupled bundles that can adapt to the changing environment in runtime, without restarting bundles as opposed to the import/export packages mechanism. The SOA becomes even more essential in OSGi due to the OSGi platform dynamic nature, as bundles can suddenly become active, providing a new functionality to other bundles of the system. This layer allows bundles to: i) register service objects with the Service Registry; ii) search the Service Registry for matching services; iii) receive notifications when services become registered and unregistered.

  The objects registered with the Service Registry are called services, since they are registered with an interface name that represents the methods the service provides. Besides the interface, services are registered with a set of properties. Searching the available services is done with a filter language that allows to filter all the available services by the interface name and service properties. Figure 2.2 represents a bundle A that registered a Service S and a bundle B that uses the service. This representation will be used throughout this dissertation.

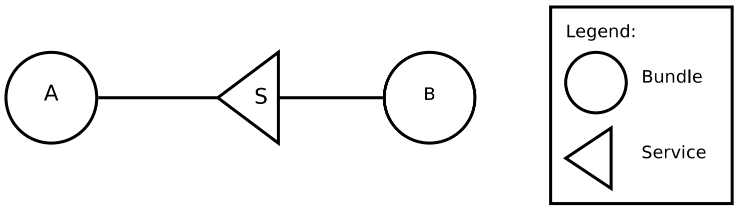

Figure 2.2: OSGi Service

- The *Security Layer* extends the Java 2 security architecture, specifically the permission model to adapt it to the typical use cases of OSGi deployments.

  Since the OSGi platform can execute different bundles from different sources it requires a more strict security model that is based on permissions. The permissions implemented by the OSGi security layer are bundle-based instead of application-based. The layer also offers support to protect several type of resources: files, packages, and services.

  Another security extension is the visibility between packages. The standard Java access modifiers (private, package private, protected and public) are extended with an extra level of module privacy by making packages only visible within the bundle.

2.1.2 OSGi Service Oriented Components

As previously described, the OSGi platform provides the support for applications built using a service-oriented architecture (SOA). This approach allows the construction of applications that can react to changes in the environment, such as a providing a service when a external event occurs.

Although OSGi provides some mechanisms that ease the construction of such applications (the Service Tracker class) it is still a repetitive and error-prone task to manage dependencies directly. A number of solutions exist to ease the management of services dynamism such as Declarative Services (DS), Spring Dynamic Modules, and iPOJO. These tools automate the service registration and service dependency management. Besides of easing the management of the service dynamism, these solutions provide bundle reduced startup time and memory footprint because the services are loaded only when they are needed instead of when bundles start. These solutions provide a so called service oriented component model.

All these solutions use separation of concerns, i.e the business logic is separated from the service dependencies management. Hence the business logic is implemented by POJOs (Plain Old Java Objects) and the service dependencies are specified in metadata that describes the service oriented components details.

2.1.2.1 Declarative Services

Declarative Services (DS) is part of OSGi Specification. It relies on a declarative model for publishing, finding and binding to OSGi services.

Bundle developers that want to design DS bundles must implement a component class implementation that is a simple POJO and specify metadata to describe the component class information. The description is made using a XML document (the component description) that describes:

- The implementation class of the component.
- The provided service interfaces.
- The required services interfaces, as well the cardinality, and the bind and unbind methods that will be called when the services are available.

The entity responsible for registering the provided services and retrieving the required services is the Service Component Runtime (SCR) which is a bundle that monitors bundle start and stop events. When a bundle is started, the SCR searches for the component description and, if all the required services dependencies (cardinality) are fulfilled, the provided service is registered.

2.1.2.2 Spring Dynamic Modules

Spring Dynamic Modules existence its prior to OSGi. Spring already existed as a framework to build applications from the composition of modules called *Spring Beans*. Spring Dynamic

Modules allows Spring Applications to be deployed over the OSGi platform without writing any code to couple the application with the OSGi platform, but instead by simply leveraging in the OSGi benefits (modularity, versioning support, and module life cycle operations).

Like in DS, to register a service or retrieve a service dependency the Spring Beans must include a XML description of the these attributes.However the size of the Spring Dynamic Modules framework is a disadvantage requiring a very large number of bundles to be deployed in the OSGi platform.

As a side note, notice the existence of the Spring bundle repository that hosts a very large collection of "OSGified" enterprise libraries JAR's, which is a contribution to a widely OSGi adoption.

2.1.2.3 iPOJO

iPOJO is a flexible and extensible service oriented component model that offers more features than the previous described tools. The component metadata is also described in a XML document, however it can also be described using Java annotations.

The development process of iPOJO bundles is a bit different than the process described above for the other tools. It is required to process the bundle after it is packaged (as a bundle) to perform bytecode instrumentation. This step is necessary to perform field injection of dependencies and intercept accesses to those fields. Field injection replaces the necessity of implementation of bind and unbind methods for when each service is available, and instead injects the service object in the component field. This processing step also adds some metadata to the bundle manifest to improve the startup time and overall performance.

iPOJO provides features of a service oriented component model such as service registration and service dependencies management. However, its extensible architecture allows the development of handlers to address other additional non functional requirements.

2.1.3 OSGi and Java Specifications

In this section we dwelve in the future of OSGi and the Java language specification and how they will coexist.

The elaboration of the next Java specification (Java 7) is being discussed in the JCP (Java Comunity Process) in different JSR (Java Specification Requests). Java 7 is addressing the language support for modularity however, a number of JSR's exist that present diferent requirements and existing solutions such as OSGi, namely: JSR 277, JSR 291, and JSR 294.

We now provide additional details on the JSRs:

- JSR 277 (Java^{TM} Module System) was the first JSR specification for a Java Module System, proposed in 2005. It proposes similar features to the OSGi platform, although they are a subset of OSGi features. For instance JSR 277 does not addresses the definition of a distribution format. Today this JSR is already inactive.
- JSR 281 (Dynamic Component Support for Java^{TM} SE) is the IBM proposal that brought OSGi to the JCP and is currently in its final state.
- JSR 294 (Improved Modularity Support in the Java^{TM} Programming Language) is where the language and virtual machine support for modularity in the Java platform is being discussed. JSR 294 does not define a module system, but simply the language changes and required support for the JVM. In fact, JSR 294 only provides a description of mechanisms upon which a module system can be built.

At the moment two module systems for Java already exist: OSGi and Project Jigsaw. OSGi has ten years and is currently widely adopted in a number of applications and platforms.

Project Jigsaw is an Open JDK project and consequently is considered in the Reference Implementation(RI) of JSR 294 as Open JDK is the RI for JSR 294. Although Jigsaw has its merits and similarities with OSGi, it also has its incompatibilities. Hence the question: "Why not build the Java module system on top of the proven and de facto standard OSGi dynamic module system?". Unfortunately, it is suspected that the answer to these questions has little to do with technology, and more to do with commercial constrains.

2.2 Autonomic Computing

The growing complexity of computing systems, the effort associated to its maintenance, and the expected lack of IT personal to support it, are some of the fundamental problems

Self-Configuration	An autonomic computing system configures itself according to high-level goals. However thee goals might not necessarily specify how to accomplish them.
Self-Optimization	An autonomic computing system optimizes its use of resources in order to improve performance or quality of service.
Self-Healing	An autonomic computing system detects problems and attempts to fix or mitigate their impact.
Self-Protection	An autonomic computing system protects itself from malicious attacks and tunes itself to achieve security, privacy, and data protection.

Table 2.1: Autonomic Self-* Properties

that Autonomic Computing (AC) proposes to solve. In 2001 IBM released a manifesto (IBM 2001) identifying these problems and defining as the only solution as: Autonomic Computing. To build computing systems that could manage themselves given high-level objectives from administrators.

The term “Autonomic Computing” has its roots in biology, namely in the Autonomic Nervous System that governs our heart rate and body temperature, thus freeing our conscious brain from the burden of dealing with these and many other low-level, yet vital, functions.

Like biological systems, computing systems must own a set of properties, that enable computing systems to maintain and adjust their operation in the face of changing conditions (e.g components, workloads, demands, and other external conditions) and in the face of hardware or software failures. These properties are self-configuration, self-optimization, self-healing, and self-protection, also known as self-* properties. These properties are summarized in Table 2.1.

The Autonomic Computing initiative proposed a vision (Kephart & Chess 2003) where computing systems must own these self-* properties. The vision proposes the progressive addition of these properties (or a sub-set) to existing systems or subsystems (for example an autonomic database or autonomic middleware) creating autonomic elements, and further increasing the autonomicity of systems by integrating the different autonomic elements in common platforms.

Autonomic Computing as a research topic is not new, embracing existing computer science research areas such as software architecture, artificial intelligence, and control theory, as well as other sciences as biological systems and economic mechanisms (Kephart 2005). Concepts like self-adaptation and context-awareness already existed before the Autonomic Computing initiative. Emergent behaviors are also being used to introduce autonomic capabilities to system, inspired by biological systems like ant colonies or insect swarms.

Some research challenges that were created by Autonomic Computing can be described as follows:

- research on autonomic elements: research that focus on technologies that can be generally applied to autonomic elements (i.e. planning, modeling, optimization) and research that focus on improving autonomic capabilities of specific elements (autonomic servers, storage systems).
- research on autonomic systems: research on technologies that entail interactions between autonomic elements to achieve global system level goals (i.e. problem determination, automated workload management and configuration) and architectures to support such interaction.
- research on human-computer interactions: research on the representation of high level policies created by specialist, and more often system administrators.

In the next sections we provide a historical perspective on works that affected Autonomic Computing research, followed by the description of a reference architecture for the implementation of autonomic elements: the MAPE-K Loop. We describe each of the MAPE-K loop components and that illustrate some existing research works that have focused on those components. Finally, we present some systems that exemplify implementations of this reference architecture. This portion of related work survey was greatly influenced by Huebscher & McCann (2008) and Kephart (2005).

2.2.1 Historical Perspective

Before the release of IBM Autonomic Computing manifesto (IBM 2001) some previous works already tried to address some Autonomic Computing aspects and concepts such as self-adaptation or self-protection. In this section we describe some of these works in a chronological order to provide to a brief overview on the evolution of this research area.

Situational Awareness Systems: SAS was a DARPA project initiated in 1997. Its aim was to create personnel communication and location devices for soldiers on the battlefield.

Soldiers could enter information on their personal devices that autonomously spread to all other soldiers. These personal devices had to be able to communicate with each other in aggressive environmental conditions. This is a form of decentralized peer-to-peer mobile adaptative routing, which has proven to be a challenging self-management problem. Devices adapt routing and communication frequency according to the environment and topology conditions.

Dynamic Assembly for Systems Adaptability, Dependability, and Assurance: Other DARPA project, that started in 2000, is the DASADA. Its main objective was to research and develop technology that could enable mission critical systems to meet high assurance, dependability, and adaptability requirements. The project also pioneered the architecture-driven approach and the notion of using probes and gauges for monitorization.

Self-Regenerative Systems: Another DARPA program was the Self-Regenerative Systems that started in 2004 (after IBM Autonomic Computing manifesto). Its aim was to "develop technology for building military computing systems that could provide critical functionality at all times, in spite of damage caused by unintentional errors or attacks". One key aspect of this project is its resistance to errors by generating a large number of software versions that have similar functionality, but different implementations.

Autonomous NanoTechonology Swarm: A project that started in 2005 at NASA was Autonomous NanoTechonology Swarm (ANTS). The plan was to launch into an asteroid belt a swarm of 1000 small spacecraft in order to explore the asteroid belt. Since 60%-70% of the swarm is expected to be lost as they enter the asteroid, the surviving crafts must work together. Small groups with a coordinator should automatically be formed, that coordinate in order to explore the asteroid belt. The coordinator uses gathered data to issue orders to the explorer crafts in its group.

2.2.2 MAPE-K Autonomic Loop

IBM proposed a reference architecture to design autonomic applications (IBM 2006). Although it is a well known reference model, not all research conducted can be fitted in it. This is however, the model more used to communicate and describe architectural aspects of autonomic systems.

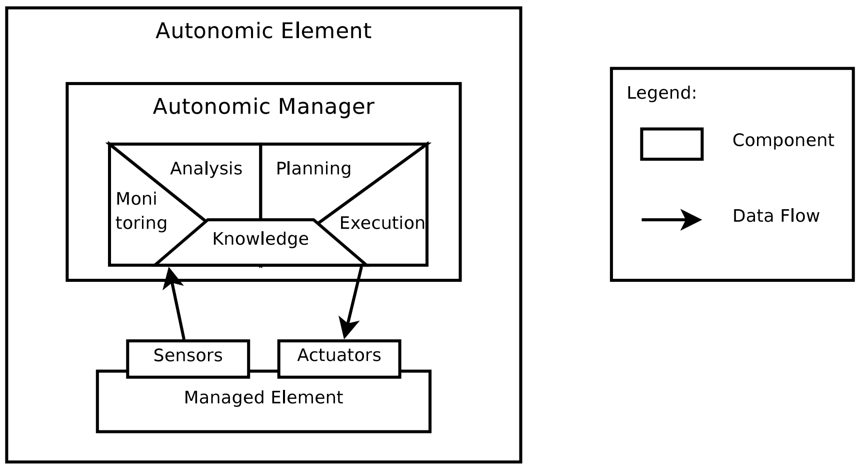

Figure 2.3: MAPE-K Control Loop Architecture

The MAPE-K autonomic management control loop consists on an Autonomic Manager that is responsible for the autonomic behavior of the managed element. The Autonomic Manager loop is made of the following components: monitoring (M), analysis (A), planning (P), and execution (E). The K stands for a shared knowledge base that supports these operations. This model is depicted in Figure 2.3.

Through the usage of sensors, the managed element is continuously monitored; this information is analyzed and an adaptation plan is generated; this adaptation plan is executed through actuators in the managed element. This autonomic loop only requires human intervention for the definition of the high level goals. In the next sections a description of each MAPE-K component is provided as well as some technologies that can be used implement them.

2.2.2.1 Monitoring

The monitoring component is responsible for managing the different sensors that provide information regarding the managed element, such as performance metrics and external events. This component provides self-awareness (aware of its internal state) and environment-awareness (aware of external operation conditions) properties to the autonomic element.

In general, the monitoring mechanisms can be grouped in two categories:

- Passive Monitoring: The managed element provides the mechanisms to support the monitoring of the interest metrics. In operating systems context, this is the general mechanism since operating systems provide interfaces for internal metrics such as CPU or memory consumption by a specific process.

- Active Monitoring: When the managed element does not offer the mechanism to monitor its internal state and metrics, it is necessary to modify, in some fashion, the managed element to allow the obtaining of relevant metrics.

Some relevant metrics that sensors can capture are the (current) consumption of critical resources (such as CPU and memory), performance metrics (such as the number of processed requests per second and the request process latency), and application domain specific metrics. Sensors can also raise notifications when important events happen, such as an external condition happens.

Some existing technologies that provide monitoring capabilities to the JVM and Java applications are JMX and JVMTI. Both will be described in section 2.3.

2.2.2.2 Analysis

The analysis component is responsible for processing the information provided by the monitoring component and to generate high level events that capture root causes of a problem or high level conditions from the aggregation of different monitoring metrics. These events should then be processed by the planning component.

Rule engines and correlation engines could be used to analyze monitored data to extract trends or situations that trigger actions to be executed. However the description of these rules or correlation expressions can be a challenging task, that human experts are likely to need assistance for authoring a large set of rules. Ideally this rule authoring should be based on some form of learning derived from system goals.

The Event Distiller (a component of Kinesthetics extreme) described further ahead in the text is an example of a rule engine to detected correlation between events. Other commercial products for event detection exist (Zhang, Cohen, Goldszmidt, Symons, & Fox 2005).

2.2.2.3 Planning

The planning component is responsible for selecting the actions that need to be applied to the system in order to correct some deviation from the desired system state, or to select the best possible state, according to the high level goals defined. Several technologies exist to express these goals:

- ECA policies: Event-Condition-Action (ECA) policies describe for a specific event that occurs and if a condition hold, the set of actions to execute. A disadvantage associated with the use of ECA rules is that conflicts between different rules might happen and, when a great number of rules are specified, these conflicts become even more hard to detect. Although some conflict resolution research exist (Ananthanarayanan, Mohania, & Gupta 2005), the conflicts may become apparent only at runtime. PONDER is an ECA policy interpreter and rules engine that is widely used (Twidle, Lupu, Dulay, & Sloman 2008)

- Goal policies: Goal policies are more high level than ECA policies since they only specify the desired state, but don't specify individual actions or how to achieve that state. It is up to the autonomic manager to select the actions to execute, using the knowledge about the managed element. Sometimes it can be impossible to achieve the desired state and the planning component may be unable to decide on another target state to which adapt to.

- Utility function policies: Utility functions solve the above problem by quantitatively defining a state desirability. The main problem with utility functions is they are hard to define as every aspect that affects the function must be quantified.

  To limit the state space for testing by the utility function, knowledge can be used. Knowledge about the managed element is used to choose the actions to execute over it, that are considered the best actions to maximize the utility function.

To select the actions to execute over the managed element several approaches have been used. The simplest approach is the specification of the concrete actions, for example the specification of actions when using ECA rules. However ECA rules present the known limitations already described above.

Other approach that improves the managed element state representation and the selection of adaptation tasks is the use of architectural models. An architectural model is composed by a network of components and connectors. Constrains can be specified over the components to detect undesirable states when the managed element violates those constrains. Other advantage is the possibility to apply the adaptation plan over the model in order to verify that the system integrity is maintained. The use of architectural models does not necessarily eliminate the need for ECA rules, since they can be used to express repair strategies. Examples of architectural models used in autonomic computing research can be found in Garlan & Schmerl (2002).

Another existing approach is the use of process coordination and orchestrations to define the adaptation plans. Instead of defining an architectural model of the managed element, a modeling of the adaptation tasks a managed element performs is used. This description defines the tasks and subtasks as well the various components available to perform each subtask. Hence, it is possible to perform adaptation, by switching the components that perform each specific subtask.

2.2.2.4 Execution

The execution component applies the actions selected by the planning component over the managed element. These actions can be applied over specific components (changing a configuration parameter) or the environment (adding more resources for a server cluster), although these actions are usually very specific to the managed element domain. The actions are applied through the usage of actuators.

Like the monitoring component the actions available greatly depend on the managed element interfaces, as well the available technologies to implement actuators.

2.2.2.5 Knowledge

The knowledge base component maintains information about the managed element to support the remaining components. Knowledge can be specified from a number of different sources:

- Human expertise: this is applied when using ECA policies or even utility functions and the system administrator translates his knowledge into policies.

- Reinforcement learning: through the usage of reinforcement learning it is possible to extract knowledge about the managed element. As its most basic it learns policies by trying actions in different states and reviewing the consequences of each action. The main advantage of reinforcement learning is that it does not requires a model of the managed element. However, the large state space impacts on the time to train. Some techniques exist that aim at reducing the state space to test by introducing domain specific knowledge.

2.2.3 MAPE-K Loop Implementation Examples

In this section we present some works that can be considered example implementations of the MAPE-K loop.

2.2.3.1 ABLE Toolkit

ABLE (Bigus, Schlosnagle, Pilgrim, & Mills III 2002) stands for Agent Building and Learning Environment and it is a toolkit for building multiagent autonomic systems. ABLE proposes an architecture that combines different artificial intelligence approaches to build robust autonomic systems. Moreover ABLE provides a component library that eases the development of such agents.

The authors of ABLE proposed a series of example agents to demonstrate the capabilities of their toolkit. One of those examples is an agent that provides a closed-loop controller, called Autotune agent that is used to manage the performance of a Web server. This Autotune agent architecture has great similarity with the MAPE-K loop presented before.

Another agent presented is the Autonomic agent, that is proposed as an “agent capable of playing a role in a future autonomic computing infrastructure”. This agent is composed of different agents that cooperate and compete to take control of the system. The autonomic agent has a number of sensors and effectors, maintains a model of the external environment and internal state of the managed element. This agent approach is greatly influenced by artificial intelligence concepts, particularly those found in Minsky (1988).

The entities that ABLE define are:

- AbleBeans: AbleBeans are standard JavaBeans (a component-model specification for Java

(Sun Microsystems 2006a)) that define a set of attributes (name, state, etc) and behavior (processing methods for agents such as init(), process(), quit()). AbleBeans can be connected to form AbleAgents.

- Connection mechanisms: AbleBeans can be connected using three methods: Data flow (each AbleBean has its own input buffer from where it reads data, that is processed and placed in its output buffers, for other AbleBeans to consume), Events (which allow AbleBeans to register listener objects in other AbleBeans that listen to events) and, properties (used to synchronize two different properties residing in two different AbleBeans).
- AbleAgent: AbleAgents are AbleBeans that can interact with their environment using AbleUserDefinedFunction objects that can wrap external Java objects. They contain other AbleBeans (that can be AbleAgents), providing a way to package different AbleBeans that perform a specific function.

ABLE also provides a comprehensive component library of AbleBeans which includes:

- Data beans: AbleBeans that provide data access and transformation.
- Learning beans: AbleBeans that implement several different learning algorithms that can be combined with data beans to develop data mining capabilities.
- Rule beans: AbleBeans that together with the Able Rule Language (ARL) provide a rule-based knowledge representation formats. ARL support the description of rules that describe the actions to execute.

The Autotune agent implements a basic closed-loop control, similar to the MAPE-K loop, where a system state is monitored, analyzed and adapted to achieve a desired goal state. The Autotune agent contains AutotuneControllers that provide control algorithms (similar to the MAPE-K's analysis and planning components) and AutotuneAdaptors that provide the interface with the managed system (similar to the MAPE-K's monitorization and execution components). To represent the managed system state AutotuneMetric classes are defined that maintain information concerning metrics about configuration and workload indicators. These metrics are maintained by the MetricManager (similar to the knowledge part of the MAPE-K loop). The architecture of the Autotune agent is represented in Figure 2.4.

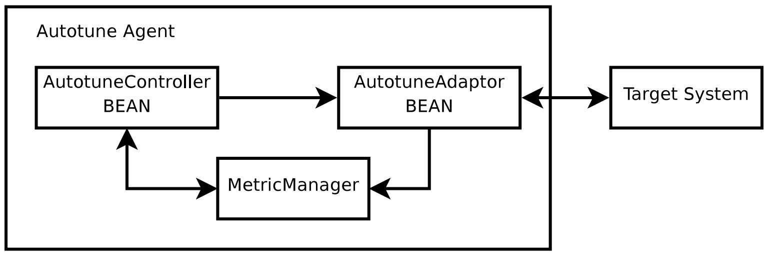

Figure 2.4: ABLE Autotune component architecture

The Autotune agent was applied in the context of performance management for the Apache Web Server[2] (Diao, Hellerstein, Parekh, & Bigus 2003). ABLE is not only used to implement a control loop agent that monitors and adapts the Web Server, but also to automate the task of designing the control algorithm. Hence three different Autotune agents were developed: *a*) a modeling agent; *b*) a control-design agent; *c*) a run-time control agent. At design time, the modeling agent is responsible for generating workloads to understand the Web server behavior (CPU and memory consumption) under these different workloads. The control design agent receives the modeling agent results and generates the controller parameters (according to the criteria specified by the system designer) that will be used by the run-time control agent. At runtime, the run-time control agent monitors the Web server and performs adaptation over the Web server modifying its configuration parameters (MaxClients and KeepAlive) in order to achieve the desired goal, defined by the system administrator. The architecture of these Autotune agents is depicted in Figure 2.5.

2.2.3.2 Kinesthetics eXtreme

Kinesthetics eXtreme (KX) was developed for the DASADA project (DARPA Dynamic Assembly for Systems Adaptability, Dependability, and Assurance) a precursor of the Autonomic Computing initiative. Its principal contribution was an architecture to retrofit autonomic capabilities onto legacy systems, considering that "real world organizations continue to use legacy systems and/or build systems of systems". Hence the work focus more on the collection and processing of monitoring data from legacy systems and execution of adaptation plans rather than algorithms and adaptation planning.

The authors define a reference model depicted in Figure 2.6, that contains four components:

[2]`http://httpd.apache.org/`

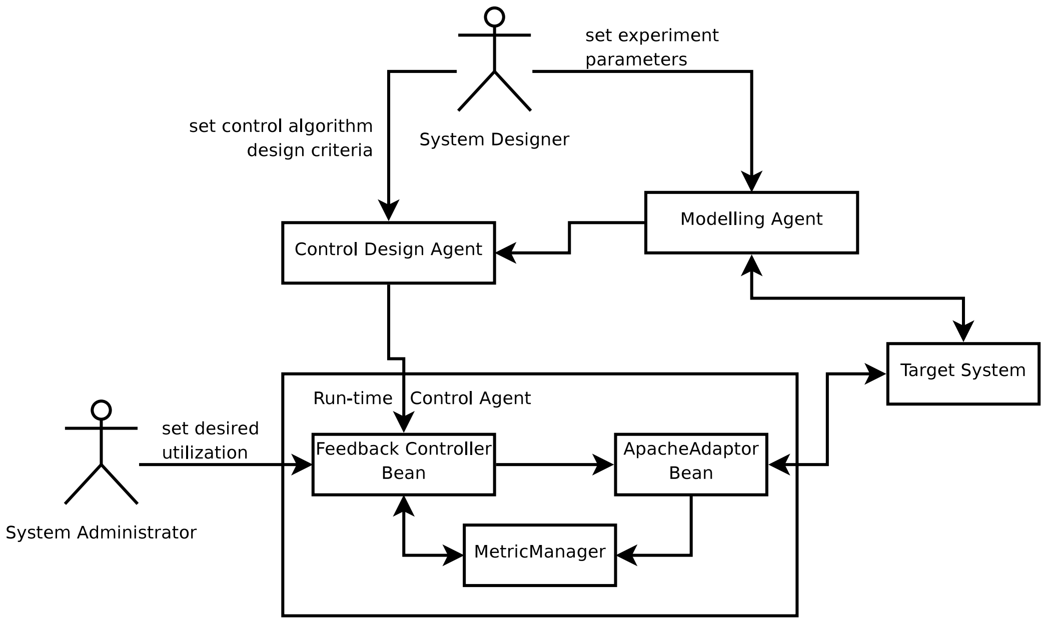

Figure 2.5: ABLE Autotune development scenario for Apache Web Server

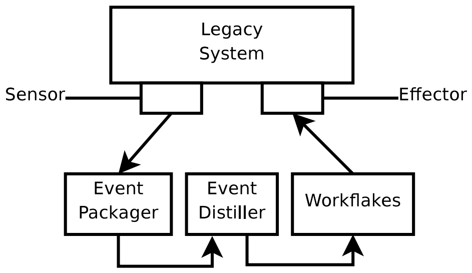

Figure 2.6: Kinesthetics eXtreme reference model

sensors, gauges, controllers, and effectors. These can be easily mapped on the MAPE-K architecture. Sensors are responsible for monitoring the target system to collect primitive data, while gauges aggregate, filter, and interpret the sensor data according to system models. Controllers make decisions on what adaptations to be performed, that trigger one or more effectors to interact with the target system to perform the low-level actions as directed by the adaptation plan. Behavioral models support the execution of gauges and controllers, providing information about the target system and its environment: its architecture, communication topology, operation rules, etc. The similarities between this reference model and the MAPE-K loop are clear.

KX is an implementation of the reference model described above and its architecture is depicted in Figure 2.7. The different components communicate through a publish/subscribe model, in order to improve the reusability of the components, with multiples KX gauges and controllers instances.

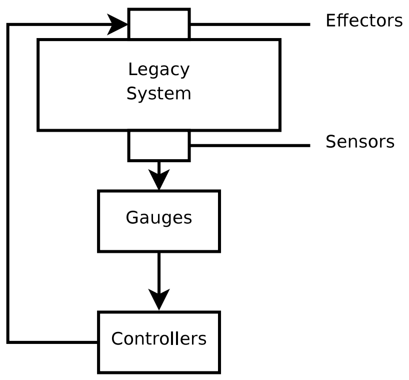

Figure 2.7: Kinesthetics eXtreme architecture overview

KX does not relies on any particular sensor or effector technology since the best technologies to use depend on the target system implementation details. However the Event Packager (Kaiser, Parekh, Gross, & Valetto 2003) component is responsible for preprocessing the different output formats of sensors to a common format. This component maps to the MAPE-K monitorization component.

The Event Distiller (Kaiser, Parekh, Gross, & Valetto 2003) performs sophisticated, cross-temporal event pattern analysis and correlation from different sensors data streams, in order to monitor desirable and undesirable behaviors. The Event Distiller is configured with rules that define complex event patterns. This component maps to the MAPE-K analysis component.

Decision and coordination capabilities are responsibility of Workflakes (Valetto & Kaiser 2003), a component that employs the workflow paradigm to execute an adaptation plan. This is the approach described in the planning component of the MAPE-K loop as process coordination.

The Behavior Models are not presented in the figure, however they support the execution of Event Distiller and Workflakes with an architectural model. The authors integrated Workflakes with CMU's Acme ADL constrains (Garlan & Schmerl 2002) so constrains can be defined to support the sensors monitoring and gauges decision.

2.2.4 Autonomic Computing in Virtualized Server Environments

In this Section we describe a work that addresses the design of an autonomic element for virtualized consolidated server environments, that host multiple multi-tier applications (Jung, Joshi, Hiltunen, Schlichting, & Pu 2008).

Virtualized server environment are becoming widely used to reduce consumption of space, hardware and energy since applications are deployed over virtual machines that allow a single machine resources to be shared between applications, but at the same time providing isolation between applications.

Other advantage of virtualization is that these resources can be dynamic allocated to the different applications. Hence it is not necessary to predict the maximum workload an application will be subjected to, in order to statically define the resources the application will use, considering the worst case scenario. Autonomic computing techniques in this context can be used to dynamically redistribute the amount of resources allocated to each application, according to the dynamic evolution of the environment.

The authors identify the following different alternatives for the generation of adaptation rules:

- Manual specification of adaptation rules based on the experience of a system administrator. This approach has the main disadvantage of the complexity associated to the generation and maintenance of such adaptation rules.
- Use of learning to train a controller that generates adaptation rules at execution time. This online learning may give rise to undesirable emergent properties that the system administrator may not detect, reducing the predictability of the system.

The proposed solution is an hybrid approach for the creation of adaptation policies which is based on static rule policies, that are automatically generated offline. The authors argue that this approach combines the strengths of both basic approaches depicted above: static rules that can be analyzed by system administrators and automatic generation of the rules.

The architecture of the system is illustrated in Figure 2.8 and it consists on the following components:

- The *Rule Constructor* is responsible for generating series of worload/configuration tuples that represent for each workload the configuration of the system that maximizes the performance of the system (this performance is measured with an utility function as described below). The Rule Constructor uses the Optimizer component to determine the suitable configuration for a specific workload.

- The *Optimizer* is responsible for choosing the configuration that maximizes the performance of the system for a specific workload. This is not a trivial task as testing all possible configurations is computationally impossible. Hence, other contribution of the authors work is a method to reduce the search state of possible configurations that will effectively be tested. The Optimizer component generates a number of candidate configurations that sends to the Model Solver component that will compute a performance measurement for the configuration.

- The *Model Solver* generates performance values for a given system configuration. These performance values are the result of an utility function that measures among other things the expected response time of the system. An architectural model is also used to represent the managed element, i.e a multi-tier application virtualized server.

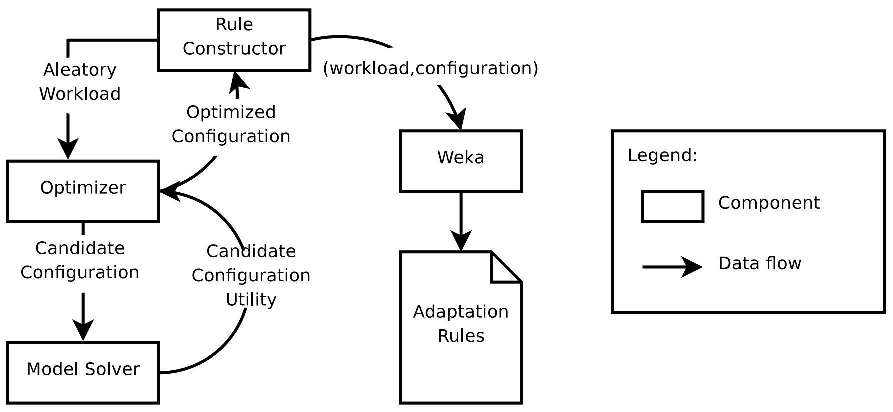

Figure 2.8: Architecture for generating adaptation policies for virtualized environments

After the generation of the workload/configuration tuples by the *Rule Constructor*, these tuples are parsed by the *Weka* component (a tool for the construction of decision trees)[3] that generates a static rule representation of the adaptations created by the rule constructor. These rules can then be analyzes by the system administrator before being introduced in the system.

[3]http://www.cs.waikato.ac.nz/ml/weka/

2.2.5 Degrees of Autonomicity

IBM also proposed a set of Autonomic Computing Adoption Model Levels that caractherize how much autonomic a system is:

- Level 1 Basic: systems that are managed by highly skilled professionals who use monitoring tools and make the required changes manually.
- Level 2 Managed: The system's monitoring tools collect information in a intelligent way to lower the systems administration burden .
- Level 3 Predictive: More intelligent monitoring is done to recognize systems behavior patterns and suggest adaptation actions to be executed by the IT personnel.
- Level 4 Adaptative: The system uses the same monitoring tools employed in Predictive level, but it is further able to take actions by itself, minimizing human intervention.
- Level 5 Autonomic: The system and its components are dynamically managed by business rules and policies, freeing IT staff to focus on maintaining other business needs.

2.3 Java Monitoring Tools

This section presents some technologies that can be useful in the monitoring component implementation of an autonomic manager for OSGi applications. These tools are JVM Tools Interface (JVM TI) and Java Management Tools (JMX).

2.3.1 JVM Tools Interface

JVM Tools Interface is a native programming interface that allows tools to inspect the state and control the execution of applications running in the JVM. The clients of this interface are called JVMTI agents. These JVMTI agents can invoke a number of functions and be notified of interesting occurrences through events. The available functions of a JVMTI agents include:

- Memory Management functions: permit the allocation and deallocation of memory.
- Thread and ThreadGroup functions: permit to get Threads and ThreadGroup information.

- Stack Frame, Heap, Local Variable functions: permit the inspection of stack frames, heap objects and local variables.
- Class, Object, Field, Method functions: permit the retrieving of information about these entities.
- Event Management functions: permits the agent to subscribe JVMTI events. Available events are described shortly.
- Capability functions: permits the agent to set the capabilities needed. Capabilities must be set at VM start event.
- Timers functions: permit to get the current time and consumed cpu time from threads.

The events an agent can subscribe include: *a*) VM start, initialization and shutdown, *b*) thread start and stop, *c*) class file load, class load and class prepare (these events allow the bytecode instrumentation), *d*) field access and modification, *e*) method entry and exit, *f*) VM object allocation and deallocation, *g*) garbage collection start and finish.

The method entry and method exit defined above significantly degrade performance since they require switching from bytecode execution to native execution. Instead JVMTI provides support for bytecode instrumentation, the ability to alter the Java virtual machine bytecode instructions which comprise the target program. Because the inserted code is standard bytecode, the JVM can run at full speed and optimize both the target program and the inserted code.

One of the advantages of the JVMTI agent is its loading time: the agent its loaded by the JVM in its early stages, even before any classes are loaded, allowing the interception of important events of the JVM, such as thread creation, class loading and other events.

2.3.2 Java Management Extensions

The Java Management Extensions (JMX) specification defines an architecture, the design patterns, and API for management and monitoring of JVM and Java Applications. The architecture defined by the JMX Specification (JSR 3) is depicted in Figure 2.9 and is comprised by three layers:

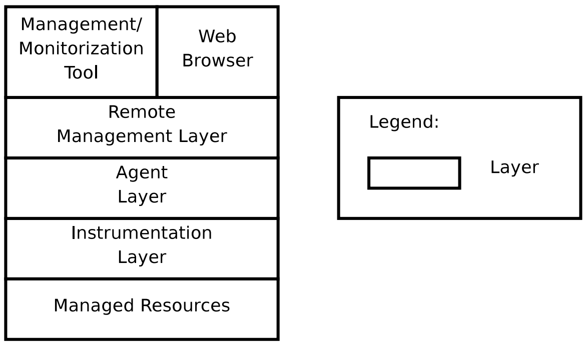

Figure 2.9: JMX Architecture

1. Instrumentation Layer: resources to be managed and monitored are wrapped by Managed Beans (MBeans) that expose their attributes and operations. Each Mbean must be registered in the Agent Layer.

2. Agent Layer: defines the MBean Server, the entity where MBeans must be registered. Agents control the MBeans and make them available to remote management.

3. Remote Management Layer: defines protocol adaptors (HTTP adaptors and WebService adaptors) as well as standard connectors that enable remote management.

A number of monitoring tools exist that implement the JMX Specification. These tools connect to a JMX agent and allows the user to browse the existing MBeans. Jconsole and VisualVM are tools that are bundled with Sun JVM that implement the JMX specification.

2.4 Performance Metrics Overview

Monitoring the CPU and memory consumption allows to partially measure the load of a system, however they are not metrics that capture the performance of a system from the user point of view. For instance, a system can maintain the CPU consumption in constant values, but not being able to process all the incoming requests, while other can present a high variable CPU consumption, and still be able to process all incoming requests. In this Section we introduce some metrics that allow to measure a performance of a system. These metrics were first presented in Brewer (2001).

2.4.1 Uptime

Uptime represents the time fraction a system is available. It can be calculated using two other metrics: the Mean Time Between Failures (MTBF) and the Mean Time to Repair (MTTR), accordingly to the following formula:

$$uptime = \frac{MTBF - MTTR}{MTBF}$$

The uptime can be improved reducing the failures frequency or reducing the time required to repair the system after failures.

2.4.2 Yield

Yield represents the fraction of queries that are completed:

$$\text{yield} = \frac{\text{queries completed}}{\text{queries offered}}$$

This metric has a numerical value which is associated to the uptime however, it better reflects the uptime as perceived by the user. For instance, a second of unavailability when there are no queries does not affect users or yield, but reduces uptime. However, the same period of unavailability in a peak period reduces both yield and uptime, so yield is a more precise metric than uptime since it reflects the user perceived system performance.

2.4.3 Harvest

Harvest reflects the fraction of data that is available at a specific moment, that reflects the completeness of a response provided by the system.

$$\text{Harvest} = \frac{\text{data available}}{\text{complete data}}$$

This metric reflects the fact that it is possible to lower the load that each request imposes to the system, by reducing the information returned to the user, for example searching only a fraction of a database (for a search query) or by reducing the quality of contents (in a multimedia portal).

2.4.4 DQ Principle

The DQ principle, as proposed by Brewer, indicates that the maximum capacity of a system can be capture by the product of yield and harvest, and that this capacity is constant:

$$\text{yield} \times \text{harvest} \approx \textit{constant}$$

This principle capture the fact that all systems have a physical limit when processing requests, for instance the I/O bandwidth, that none performance manager can improve. The best a performance manager can do is to adjust the trade-off between the yield and harvest.

These metrics and principle can be applied to the OSGi platform in the following way:

1. In load situations it is possible to reduce the harvest of the system by changing bundles for other versions of the same, that offer a a subset of the available data (reducing harvest) in order to process more requests (increasing yield).
2. Alternatively, it is possible to stop some bundles (reducing the harvest and the yield of a partition of the system) in order to increase the yield of other components of the OSGi application.

2.5 Autonomic computing support for OSGi

OSGi based applications have increased in complexity over the years (as the examples provided in Section 2 show), however the OSGi platform still lacks support for developing autonomic applications.

The platform does not provide mechanisms to monitor the operation of individual bundles and to take advantage of distinct service implementations that potentially present different trade-offs between quality of service provided to the clients and required resource consumption to provide that service. In this Section we will overview research works that address the design and implementation of some building blocks components to support an OSGi autonomic platform.

2.5.1 OSGi Monitoring

Several previous works have addressed the topic of monitoring OSGi applications (Miettinen 2008; Geoffray, Thomas, Clément, & Folliot 2008). Most of these solutions have focused on providing an adequate per-bundle CPU consumption isolation.

The work presented in Miettinen (2008) addressed the topic of monitorization and visualization of resource consumption in OSGi-based applications. To achieve a bundle-level isolation and consequent monitorization, they employ a thread-based approach, by creating *threads* and *ThreadGroups* that are associated with an individual bundle. The OSGi application execution is separated (not parallelized) in the different bundle-associated *threads*.

Once this isolation is achieved, the CPU consumption is measured by summing the CPU time of all the threads associated with a bundle.

This solution induces some overhead due to the number of *threads* created and the corresponding *thread* context switches required. Hence the authors opted for performing isolation and monitorization of selected bundles. However this solution is also not perfect since it raises resource consumption accountability issues as it will be described. Nevertheless, for practical reasons, this was the basic approach followed in our work.

Another approach can be found in Geoffray, Thomas, Clément & Folliot (2008), where the authors employ specific JVM alterations called *isolates* (or Java Processes) to implement component isolation in OSGi. This isolation can be used to monitor the resource consumption such as CPU, but also isolate OSGi bundles from each other. However OSGi applications require inter-bundle communication, hence communication between isolates is necessary. The authors addressed this problem by implementing a lightweight mechanism that simply changes a thread isolate reference whenever a inter-bundle call occurs.

The authors did not implemented the CPU consumption monitoring, but propose a solution where the JVM regularly samples the CPU time of the isolate associated thread. Unfortunately, this solution only works in modified, JVMs, which is an inconvenient for its widespread usage.

Other more generic tools could also be applied to monitor the resources consumption in a JVM, such as using bytecode instrumentation for CPU accounting and consumption monitoring (Hulaas & Binder 2008). This approach considers the addition of bytecodes after each instruction to calculate an estimate CPU usage. As one might predict this original and portable approach incurs a certain overhead, however the authors contribute an mechanism to reduce this overhead by reducing the number of bytecode modifications performed using a static path prediction scheme. Besides requiring the bytecode instrumentation of the application increasing the code size, the evaluation results presented by the authors are not very encouraging.

2.5.2 Service Oriented Components Models

As previously described, Service Oriented Component models provide the mechanisms for managing services dependencies in OSGi applications. These mechanisms introduce substitutability for a bundle service dependencies, i.e. the services a bundle uses can be substituted by other services that implement the same interface. This property can be a building block of self-healing properties to OSGi applications, as when a service becomes unavailable it automatically finds another service to replace it. For example the authors of iPOJO, performed some research in providing autonomic capabilities to iPOJO based applications (Diaconescu, Bourcier, & Escoffier 2008).

Summary

In this chapter we have presented an overview of the main background this work, namely the OSGi platform and the Autonomic Computing initiative. For this purpose, we have provided a description of OSGi platform, its specification and architecture, as well as some tools that ease the construction of service oriented components, such as iPOJO.

Then we presented the Autonomic Computing initiative, including its roots and objectives. We then described a reference model for the implementation of autonomic elements, the MAPE-K Loop, and presented some example implementations.

The chapter has also made an overview of some tools that can be used to implement monitoring mechanisms over Java applications and the OSGi platform in particular. Finally, some existing research works that have addressed the design of components to support an autonomic OSGi platform have been surveyed.

3 A-OSGi Architecture

The A-OSGi framework offers a number of extensions to the OSGi platform that support the development of autonomic OSGi applications. In this chapter, we provide an overall overview of the A-OSGi architecture. First we state the requirements of our framework, then we present the architecture followed by a detailed description of each of its components.

3.1 Overview

The A-OSGi architecture follows the general MAPE-K model (introduced previously in Section 2.2.2), to support the autonomic management of OSGi applications. More specifically, we have augmented the OSGi platform with functionalities that support monitoring, analysis, planning, execution, and knowledge aspects of that model, as well as sensors and actuators to interact with the OSGi platform and applications.

3.2 A-OSGi design requirements

As stated before the A-OSGi framework is inspired in the MAPE-K control loop. The different MAPE-K components have different requirements and some are very dependent on the managed element (i.e the OSGi platform and application). This is the case for monitoring, execution and knowledge components. Further ahead in this Section we state these requirements for each MAPE-K component.

Another requirement of our architecture is the support for “legacy” OSGi bundles, i.e. bundles that were implement for the regular OSGi platform, that can be installed over the A-OSGi platform. The OSGi platform facilitates the achieving of this requirement since our extensions can be implemented without explicit support of the bundles, eliminating the need for modifying legacy bundles.

3.2.1 Monitoring

The monitoring component provides information regarding the managed element (i.e. the OSGi platform and the deployed bundles that constitute the application). The relevant metrics we considered in this context were resource consumption metrics, such as CPU and memory consumption, and performance metrics such as throughput or latency to process requests.

Besides the metrics being monitored, there are a number of OSGi platform related events that must be registered. These events are related to bundles and services life-cycle such as bundles starting and services registering.

Since an OSGi application is created from the composition of different bundles, an important issue to be addressed is the granularity of the monitorization. Although the performance and resource consumption of the JVM as a whole is of interest (for example the JVM memory consumption), the possibility to understand the contribution of each OSGi bundle to that metric is a valuable information that the monitoring component should provide. Therefore, it is a requirement of our framework the fine-grained monitoring of the OSGi application (i.e at the bundle level).

Another issue related to the used granularity and the sharing of classes between bundles is the resource consumption accounting. This problem can be described as follows: a bundle A provides a service that bundle B uses. As bundle B invokes the service methods to which bundle should the resources be accounted? The alternatives are:

1. Direct accounting: The resources consumed by the service are accounted to bundle A, the service provider bundle.
2. Indirect accounting: The resources consumed by the service are accounted to bundle B, the caller bundle.

We will discuss our approach in the next Section: A-OSGi architecture section.

3.2.2 Analysis

The analysis component aggregates data from the different metrics gathered by the monitoring component in order to capture macroscopic scenarios related to the state of the managed element. In the OSGi context, the analysis component could use the different metrics monitored (high usage of resources in different bundles) in order to recognize undesirable states or known patterns that could be used by the planning component to avoid undesirable situations before they happen (e.g. foresee an incoming overload period).

3.2.3 Planning

The planning component chooses the set of actions that are executed over the managed element according to the monitored information. Both the actions and monitored data are provided by the monitoring and execution components, respectively. The OSGi context does not imposes any requirement on the planning component. However, we must select a technology to address this issue. In the A-OSGi architecture Section we will describe how this component is designed.

3.2.4 Execution

The execution component applies the actions selected by the planning component. By itself, the OSGi platform already supports some actions, as the ability to start and stop bundles, with the consequence of exposing or disabling some functionality. We think these actions are not sufficient in order to support the constructions of OSGi autonomic applications. It would be of interest to support the dynamic change of bindings between bundles and services, i.e. changing a service implementation for a different one that, although providing the same interface or functionality does it with a different quality of service and resource consumption trade-off. Later in Section 3.3 we will detail the supported actions.

3.2.5 Knowledge

The knowledge component supports the operation of the other MAPE-K components. Since the OSGi platform presents such a dynamic nature, this component is of great importance to the operation of the other components since the state and relations of the managed element are hard to resolve (or even impossible) at design time. Hence, the knowledge component provides information about the deployed bundles, registered services, and the relation between services and bundles. For example one would like to know which services a bundle depends on, which services a bundle is currently using, and so on.

3.3 A-OSGi Architecture

The A-OSGi architecture follows the general MAPE-K model as stated before. The MAPE-K functionalities are provided by four 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). We will describe each of these components in the following sections.

A-OSGi is implemented on top of the original OSGi platform, however some extensions and modifications to the base OSGi platform were required. The A-OSGi architecture is depicted in Figure 3.1.

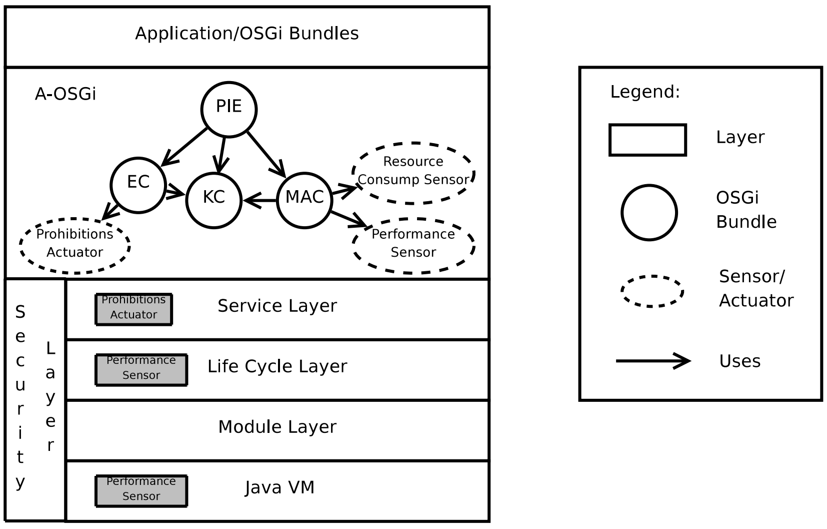

Figure 3.1: A-OSGi Architecture

3.3.1 A-OSGi Monitoring and Analysis Component (MAC)

The MAC component provides the monitoring and analysis functionalities of the MAPE-K loop and is responsible for retrieving information from sensors. 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 the provided events. However, by exposing a publish-subscribe interface, we facilitate the usage of this component by other applications, for instance a monitoring application.

The metrics that MAC component monitors are resource consumption and performance metrics. The resource consumption metrics include CPU and memory consumption, while the performance metrics are throughput and latency of requests.

The sensors do the monitoring of these metrics per bundle, i.e. the granularity of the monitorization is the bundle. This allows a more fine grained monitorization in order to better perform adaptation over the OSGi platform and application.

The accounting of the resources (discussed previously) was first approached with a direct approach, i.e. the resources consumed are accounted to the service provider bundle. The direct accounting requires an isolation mechanism to separate the resources consumed by the caller and the callee. Due to performance overheads and implementation issues we did not isolate every bundle, allowing the configuration of what bundles to isolate. This solution allows the usage of both direct and indirect accountancy strategies for different bundles.

Consider the following example where a bundle A uses a service S provided by bundle B and bundle B uses a service T provided by C as figure 3.2 exemplifies. When service S is invoked it will invoke service T as the sequence diagram in figure 3.3 shows.

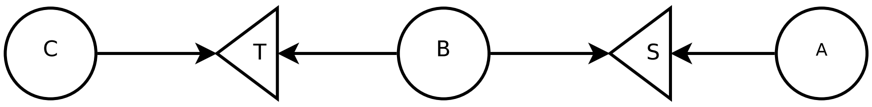

Figure 3.2: Isolation Example

The system administrator is able to specify which services should be isolated. The following combinations might occur:

1. Both services isolated: three resource consumption values will exist, one for each bundle.
2. Service S isolated: two resource consumption values will exist, one for bundle A and other for bundle B plus C.

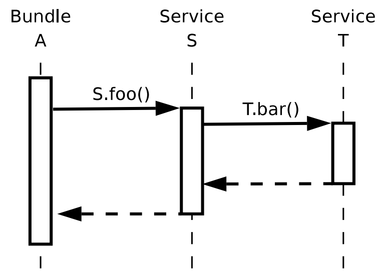

Figure 3.3: Isolation Example Sequence Diagram

3. Service T isolated: this scenario is similar to the previous one, however the two resource consumption values are different, one for bundle A plus B, and another for bundle C.

4. Only one resource consumption value exist, for the three bundles.

This number of alternatives allows the system administrator to deploy a number of different monitoring solutions depending on adaptation requirements.

The MAC component is also responsible for generating new events from the composition of other events. In the current prototype (described in Chapter 4), there is no explicit support to specify these using some form of domain specific language constructs: analysis events 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. Hence the "analysis" part of the MAC component is to filter some events to the other components, for example only triggering events when there are changes in the resource consumption levels.

3.3.2 A-OSGi Execution Component (EC)

The EC provides the execution component of the MAPE-K loop and is responsible for executing actions over bundles, services, and the base OSGi platform. It provides an API that other components can use to actuate over the managed element. In the current version of the architecture, only the PIE component uses the EC component.

As noted before the actions provided by the OSGi platform (start and stop bundles) are not sufficient to implement some autonomic behaviors. We designed an extension that allows to have a more fine grained control on how bundles bind to services through a set of prohibitions and obligation expressions.

The obligations and prohibitions mechanism is an extension to the SOA provided by the OSGi platform, that filters the services a bundle can find. With this mechanism it is possible to control to which services a bundle binds, imposing bind obligations or prohibitions. This is very useful in the scenario where several implementations of the same service exist, but the several implementations offer different trade-offs between resource consumption and offered quality of service.

In summary, this interface allows 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.

3.3.3 A-OSGi Knowledge Component (KC)

The KC component provides the knowledge component of the MAPE-K loop and allows other components to consult information regarding the state of the managed element. In our current architecture the information maintained by the KC is accessed by the PIE component, which uses it to compute adaptation plans, but is also used by the MAC and EC components in order to support their operation.

The KC component provides an API to discover the set of deployed bundles and registered services as well as to gather properties about bundles (such as bundle name and bundle identifier) and services (such as service interface and service properties). Furthermore, it provides functions that return a bundle currently used services, provided services, and the bundles that are using a specific service. This component also provides information about the prohibitions/obligations mechanism.

3.3.4 A-OSGi Policy Interpreter and Enforcer (PIE)

The PIE component provides the planning component of the MAPE-K loop and is responsible for performing adaptation plans enforcing the system policies. The system policy is described by a set of ECA rules.

Each ECA rule specifies an event (from the available MAC events), a condition (possibly using KC functions to compute a predicate, evaluated at run time), and actions to be executed if the condition holds true.

Following the dynamic nature of the OSGi platform these system policies can be deployed in run-time to support a major alteration of the system policy without restarting the system, for example when more bundles (and additional functionality) are installed that need specific (or additional) adaptation plans.

Summary

In this chapter we have presented the A-OSGi architecture, inspired by the MAPE-K autonomic control loop. We have first identified the design requirements for each MAPE-K component (such as monitoring or execution) and then presented a description of each component of our architecture, namely: MAC, EC, KC, and PIE.

4 Prototype Implementation

In this chapter we describe in some detail the implementation of A-OSGi architecture. The components of the A-OSGi architecture are implemented, themselves, as OSGi bundles and leverage on some modifications to the OSGi platform. Naturally, these bundles need to be installed and started to support the autonomic behavior of OSGi applications. As one would expect the bundles have dependencies on each other (for example the PIE component requires the MAC component).

Some of the functionality required to implement these bundles required small changes to the standard OSGi platform. More precisely, to implement the monitoring component we had to modify the *Life Cycle Layer* and the *Service Layer* of the basic OSGi framework; and to implement the binding obligations/prohibitions mechanism we had to modify the *Service Layer* of the original OSGi framework.

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 (Eclipse Equinox), Apache Felix (Felix Apache) and Knopflerfish (Knopflerfish). For the work presented in this paper we have selected the Apache Felix 1.6.0 implementation. We chose this implementation for its good documentation and great community support. The Apache Felix project also hosts a number of sub-projects not directly related to the OSGi specification, for example a command-shell bundle, a webconsole bundle and iPOJO. Some of these projects influenced the 4.2 OSGi specification draft (OSGi Alliance 2009).

As mentioned before small changes in the OSGi platform were necessary, however we believe that the changes performed can easily be ported to other existing implementations.

We chose to export the interfaces of the KC, EC, and MAC components as JMX Managed Beans (Sun Microsystems 2006b). Thus, any existing JMX client 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 remote applications.

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 (Pax Web) implementation of the OSGi HTTP service specification (OSGi Alliance 2007b), that uses the Jetty HTTP Server (Jetty HTTP Server).

4.2 Using A-OSGi

A requirement of our architecture is the support for "legacy" OSGi bundles, i.e. bundles that were implement for the OSGi platform.

This was achieved by implementing our mechanisms as modifications and extensions to the OSGi platform and clearly separating the functional requirements of applications from the non functional requirements of the autonomic behavior. The autonomic behavior description is specified by the PIE component that is decoupled from the business code from bundles.

However we detected some issues in the reasoning about the state of a SOA application, namely the detection of bindings between bundles and services, i.e detecting exactly when a bundle binds to a service.

The *Service Layer* provides an API for bundles to publish and find services. To publish a service a bundle registers with the OSGi platform an object that implements the specified interface (as Listing 4.1 illustrates); a bundle that needs a service must retrieve a ServiceReference that identifies the needed service and get the service from the ServiceReference (as Listing 4.2 illustrates). The OSGi platform keeps track of which bundles have ServiceReferences to which services in order to notify the bundles when services disappear.

Although the OSGi platform provides some classes (such as serviceTracker) to ease the retrieving of services, i.e the code responsible for managing a bundle service dependencies, delegating this management of dependencies to the developer of the bundle, makes it very

```
HelloService hello = new HelloService();
Properties props = new Properties();
props.put("Language", "English");
context.registerService(HelloService.class.getName(),hello, props);
```

Listing 4.1: Registering an OSGi Service

```
ServiceReference sr  =  bc.getServiceReference(HelloService.class.
   getName());
HelloService hello = (HelloService)bc.getService(sr);
hello.foo();
```

Listing 4.2: Getting an OSGi service

difficult to understand bundles dependencies, since they are not directly described in some form of metadata (the dependencies are described among the bundle code). Although the platform keeps track of the ServiceReferences a bundle has, this is a very limited approach to determine a bundle service dependencies.

Other problem is related to the weak implementation (or sometimes simplification) of a bundle services dependencies code as the Listing 4.2 is an example (notice that between getting the ServiceReference and getting the service object the service can disappear). Also, related to implementation issues, the code of a bundle that is responsible for the management of service dependencies, may not be prepared to manage the existence of different implementations of a given service.

Consider the following scenario: a bundle requires a service but several implementations of that service are registered in the OSGi platform. The client bundle can listen for events when services become available or unavailable. Several approaches exist to search, bind, and use that service:

1. The bundle maintains a list of ServiceReferences for the available service implementations and manages this list according to the received events. The bundle gets the service from one ServiceReference and uses the service object to call its methods.

2. The bundle has a list of service objects and manages this list according to the events received. Whenever the bundle wants to use the service it uses a service object from that list.

Although the differences are minimal, from the OSGi platform point of view in the first case the bundle is using one service but in the second case is using all the available services that implement the required interface.

Instead of trying to understand the services dependencies of a bundle accordingly to its service usage, we require the bundle developer to use a service-oriented component model tool to describe the services dependencies and to manage the retrieving of the services. With this approach the model tool uses a unique method for managing ServiceReferences and services, as well releases the bundle developer from the task of managing the bundles dependencies directly. We leverage on the model tool to understand the services dependencies.

Although this seems a hard requirement for the usage of A-OSGi, we think its a better approach than returning wrong information concerning a bundle dependencies.

4.3 MAC Implementation

The MAC component monitors different aspects of the OSGi execution as described in the previous Chapter, such as resource consumption and performance with a bundle grain level. A number of sensors were implemented, each with its own specific requirements, as will be described in the following Sections.

4.3.1 Performance Sensor

A sensor that monitors the requests received by the HTTP server and stores information concerning the bundle in charge of processing the request. This sensor is able to provide information about the number of requests processed by each bundle per second. 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 monitor the received requests.

Since the registering of servlets and resources is bundle based, i.e each bundle registers its resources, these modifications only required some study of the Pax Web source code and minimal modifications in order to measure the CPU and time consumed in the processing of each request and store this information associated with the bundle responsible for the request.

Since this sensor only monitors some bundles, i.e the bundles that register resources in the HTTP server, to differentiate such bundles we call them: “WebBundles”.

4.3.2 Resource Consumption Sensor

This is a sensor that monitors CPU usage and memory consumption per bundle. As described earlier, to support the bundle granularity of the monitorization some sort of isolation among bundles is necessary. To implement our prototype, we used a thread based approach to achieve the isolation, by creating a bundle specific *ThreadGroup* that aggregates all the threads of a 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 with the parent ThreadGroup as the *ThreadGroup* of that bundle. As a result, all threads created by the starting method belong to the *ThreadGroup* associated with the bundle. Figure 4.1 presents an example of the *ThreadGroups* hierarchy before and after the modification.

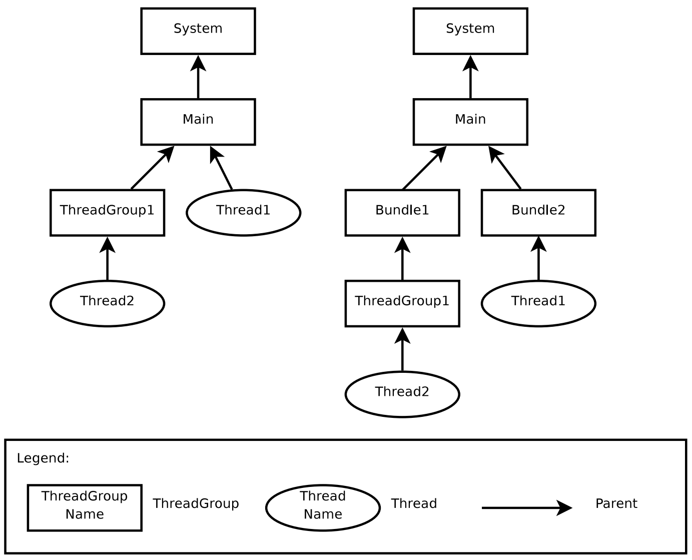

Figure 4.1: Comparison of ThreadGroups hierarchy

However, not all start methods create threads to execute the bundle functionality, as they may provide library classes or publish services for other bundles to use, making this mechanism useless. We decided to only isolate service interactions, since isolating class interaction would be a very difficult task out of the scope of this work, being, however, a limitation of our framework.

Therefore, clients of a service are provided with a proxy that executes the service methods in a thread associated to the bundle that registered that service, providing a direct accounting. This behavior was achieved by modifying the iPOJO service registration mechanism.

When a bundle is configured to execute isolated, instead of registering a service implementation in the OSGi registry, a proxy is registered that executes the service methods in a separate Thread associated to the registering bundle *Thread Group*.

This was achieved using the *java.lang.reflect.Proxy* class that permits to create dynamic proxy classes. Hence, when iPOJO registers services, instead it registers an instance of our dynamic proxy class. The proxy class is a class that implements a list of interfaces specified at runtime when the class is created, i.e the service interface. When the proxy instance is created a *java.lang.reflect.InvocationHandler* is passed as argument; when the proxy methods are invoked they will be dispatched to the *invoke* method of the *InvocationHandler*, that executes the proxied service in a different thread.

Recall the example presented in the Architecture chapter, where a bundle A calls service S (provided by bundle B) that calls service T (provided by bundle C).

If all the services are isolated the execution of this invocation is done using three threads:

- a thread executing bundle B code that calls service S;
- a thread executing service S method that bundle B called;
- a thread executing service T method that service S called;

This approach has a non-negligible overhead as it requires two context switch for each service invocation, requiring four thread context switches in the above example.

Finally, this approach may cause deadlocks in services with synchronized methods, since each service object has its associated thread that executes its methods. In the example above, if service T calls a service S method a deadlock will occur since service S thread is waiting for service T to return. This could be tackled with the use of a thread pool for each service object that could grow indefinitely, however when considering services where its methods are synchronized, this thread pool only mitigates this issue.

As already described, we chose a configuration approach where the services to be isolated can be configured. This approach allows a number of alternatives when configuring the monitorization of the deployed bundles that are more expressive than a complete direct or indirect resource accountancy.

Although, the isolation approach requires a careful configuration, it is able to provide enough feedback to support the required information to implement many relevant autonomic behaviors.

With thread isolation, CPU usage can be calculated iterating over the threads associated to a bundle ThreadGroup and sum all the threads CPU time. A JVMTI agent was developed to perform this task, that is controlled by the resource consumption sensor. This agent is responsible for maintaining a history of cpu consumptions in order to compare them between two periods of time.

The same approach was extended to monitor memory usage detecting the allocation of objects and assigning those allocations to the thread that is performing the operation.

4.3.3 OSGi Platform Sensor

This sensor provides the already existent events in the OSGi platform concerning the bundle life cycle and service registration. The binding between a bundle and a service (to which we called "client registration") is monitored by leveraging in iPOJO. We modified iPOJO so whenever it binds a bundle to a service it publish in OSGi platform that is being subscribed by his sensor.

4.3.4 MAC analysis

As described in the previous chapter the MAC does not provides any explicit support for event correlation or detection of macroscopic scenarios. The only analysis functionality that the MAC provides is to periodically check the data from the sensors, and if there is a change in the monitored data (for instance: CPU consumption percentage of a bundle, memory consumption, throughput or latency), it generates an appropriated event, so the interested components can act accordingly (in our prototype only the PIE component).

The complete list of events currently provided by the A-OSGi MAC is listed in Table 4.1.

4.4 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

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

Table 4.1: A-OSGi MAC Events

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 mechanisms:

- 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 developer of the bundle exposed as a service property.

The complete list of actions supported by the EC component is listed in Table 4.2. In order to implement the EC component we have augmented the OSGi service layer. In A-OSGi, this layer was modified to maintain, for each bundle, the associated service binding obligations and prohibitions. This information is used in run-time to filter the services a bundle can bind, in order to satisfy the constraints defined at each moment. We resort to iPOJO functionality to ensure the correctness of bindings, accordingly to the prohibitions and obligations defined.

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

Table 4.2: A-OSGi EC Actions

In the future, with the release of the OSGi specification 4.2 (OSGi Alliance 2009) and its service hooks mechanism, this behavior can be implemented without modifying the platform. These hooks provide the ability to register methods that will be executed when a specific bundle registers services, searches ServiceReferences and binds to services (although with the limitations described). To implement our obligations/prohibitions mechanism a method should be registered on ServiceReferences searches, that filters the results that will be returned to the searcher bundle.

4.5 KC Implementation

The KC provides a set of methods that allows the consulting of runtime information about the installed bundles and the registered services, as well as the dependencies between bundles and services.

The dependencies that can be queried are the services a bundle provides, the services interfaces it requires and the services it currently uses. It is also possible to discover the bundles a bundle is using (through services), i.e the bundles that are using a service provided by the specified bundle. The service related methods allow to consult the bundle that is providing a specific service, the services that implement an interface, and the bundles that use a service. The KC also provides methods to retrieve the current set of service obligations or prohibitions as well the service properties.

To implement these functions, we use the OSGi module layer to extract information about services that a bundle is using and the services it provides; the OSGi service layer was used to extract information about the bundles being used by a service. To consult the interface names a bundle requires we leverage in iPOJO descriptions of a bundle service dependencies. The full interface of the KC component is listed in Table 4.3 and 4.4.

Function	Parameters	Returns
getAllBundles		BundleID[ ]
getWebBundles		BundleID[ ]
getBundleName	BundleID	BundleName
getBundleID	BundleName	BundleID
getProvidedServiceNames	BundleID	ServiceName[ ]
getProvidedServiceIDs	BundleID	ServiceID[ ]
getProvidedServiceIDbySpecification	BundleID, ServiceName	ServiceID
getRequiredServiceSpecification	BundleID	ServiceName[ ]
getUsedServiceIDs	BundleID	ServiceID[ ]
getUsedServiceIDsbyName	BundleID, ServiceName	ServiceID[ ]
getAllUsedServicesIDs	BundleID	ServiceID[ ]
getUsingBundles	BundleID	BundleID[ ]
getAllUsingBundles	BundleID	BundleID[ ]
getClientProhibitions	BundleID	ServiceID[ ]
getClientObligation	BundleID	ServiceID

Table 4.3: A-OSGi Bundle related KC functions

Function	Parameters	Returns
getAllServices		ServiceID[ ]
getServiceSpecification	ServiceID	ServiceName
getServiceSpecifications	ServiceID	ServiceName[ ]
getServiceBundle	ServiceID	BundleID
getServiceImplementations	ServiceName	ServiceID[ ]
getUsingBundles	ServiceID	BundleID[ ]
getAllUsingBundles	ServiceID	BundleID[ ]
getAllUsingWebBundles	ServiceID	BundleID[ ]
getServiceProperty	ServiceID, Property	Value

Table 4.4: A-OSGi Service related KC functions

4.6 PIE Implementation

For implementing the PIE component we have used the Ponder2 policy interpreter (Twidle, Lupu, Dulay, & Sloman 2008). Ponder provides an ECA rule interpreter and enforcer implemented in Java.

The basic entities in Ponder2 are the *Managed Objects* and *PonderTalk*. *Managed Objects* are written in Java and use Java annotations to describe the association between the java methods of the *Managed Objects* and the PonderTalk messages that can be sent to the *Managed Objects*.

PonderTalk is the language provided by Ponder2 to interact with the Managed Objects, where its syntax is based in SmallTalk, as its name suggests. It is an object oriented language that allows sending messages to objects, i.e the *Managed Objects*. It supports assignment and a number of basic types exist such as number, string, boolean or array that are also *Managed Objects*, since they are created from their Java counterparts. Other important basic type in PonderTalk are blocks that are objects that contain code that will only be evaluated when they are executed.

We implemented our *Managed Objects* that we used as adaptors to interact with the KC and EC components and created *Ponder Events* that route the events received by the MAC component(using the corresponding JMX MBeans). The *Ponder Events* are created when the Ponder2 is launched by the PIE component.

ECA rules are Ponder objects created from the ECA policies factory. Three messages must be sent to the ECA policy object to define the ECA rule: "event", "condition" and "action". The "event" message specifies an event from the available events that will trigger the evaluation of the ECA rule. The "condition" message specifies a block to be evaluated that must return a boolean value, that if true triggers the execution of actions. The "action" message specifies a block with the actions to be executed, in such a case.

Instead of having two *Managed Objects* instances representing each of the A-OSGi components interfaces (the KC functions and PIE actions), our *Managed Objects* represent the Bundle object and the Service object. Listing 4.3 and 4.4 represents these *Managed Objects* classes. We think this design is a more pure object oriented approach being more suitable to PonderTalk.

The use of Ponder also allows the dynamic definition of the policies, a property very useful in a OSGi system due to the dynamic nature of the platform. Although the PonderTalk language was created to ease the description of ECA rules, we think a better interface for the creation and management of ECA rules was a candidate for Ponder future improvements. An interface that makes ECA rules a first class entity in Ponder would better enable the Autonomic Computing vision where autonomic elements manage other autonomic elements creating Autonomic

```
public interface IBundleObject {

  public P2String getBundleName();
  public P2Number getBundleID();

  /*Execution Methods*/
  public abstract void startBundle();
  public abstract void stopBundle();

  public void setClientProhibition(P2Object service);
  public void removeClientProhibition(P2Object service);
  public void removeClientProhibitionForServiceName(String serviceName);
  public void setClientObligation(P2Object service);
  public void removeClientObligation(P2Object service);

  /*Knowledge Methods*/
  public P2Array getProvidedServiceNames();
  public P2Array getProvidedServiceIDs();
  public P2Object getProvidedServiceIDbySpecification(String
     specification);

  public P2Array getRequiredServiceSpecification();
  public P2Array getUsedServiceIDs();
  public P2Array getUsedServiceIDsbySpecification(String specification);
  public P2Array getAllUsedServicesIDs();

  public P2Array getUsingBundles();
  public P2Array getAllUsingBundles();

  public P2Array getServiceProhibitions(String serviceName);
  public P2Object getServiceObligation(String serviceName);
}
```

Listing 4.3: Bundle Ponder2 Managed Object

Computing systems.

4.7 Framework Modifications

As stated in the beginning of this chapter, in order to implement A-OSGi, some modifications to the OSGi platform were necessary. These modifications are summarized as follows:

- *JVM level*, a JVMTI agent was implemented to support the monitoring of CPU and memory usage with a per-bundle grain.
- *Life Cycle Layer*, the execution of the bundle start method was modified in order to execute this method in a new Thread with a corresponding ThreadGroup;

```
public interface IServiceObject {

  /*Basic Attr*/
  public P2Number getServiceID();
  public String getSpecification();
  public P2Array getSpecifications();
  public IBundleObject getBundle();

  /*Knowledge*/
  public P2Array getUsingBundles();
  public P2Array getAllUsingBundles();
  public P2Array getAllUsingWebBundles();

  public String getServiceProperty(String property);
}
```

Listing 4.4: Service Ponder2 Managed Object

- *Service Layer*, to implement the prohibitions and obligations mechanism in order to filter services a bundle can find, so the services that a bundle can discover respect the currently defined constrains.
- *iPOJO*, to understand a bundle service dependencies and also the service proxing mechanism.

Summary

This chapter has described some relevant implementation details of our A-OSGi prototype. The prototype supports the installation of legacy bundles and implements the architecture described in Chapter3, namely sensors to monitor and actuators that perform actions over the OSGi platform and bundles.

5 Evaluation

In this chapter we illustrate the use of the prototype and evaluate the potential of A-OSGi to build autonomic OSGi-based applications. We implemented an OSGi application that supports an on-line store, described some adaptation policies, and evaluated the application with a dynamic workload.

5.1 Case Study

Our case study uses a Web Application that supports an on-line store, that has been implemented as a normal OSGi application using iPOJO.

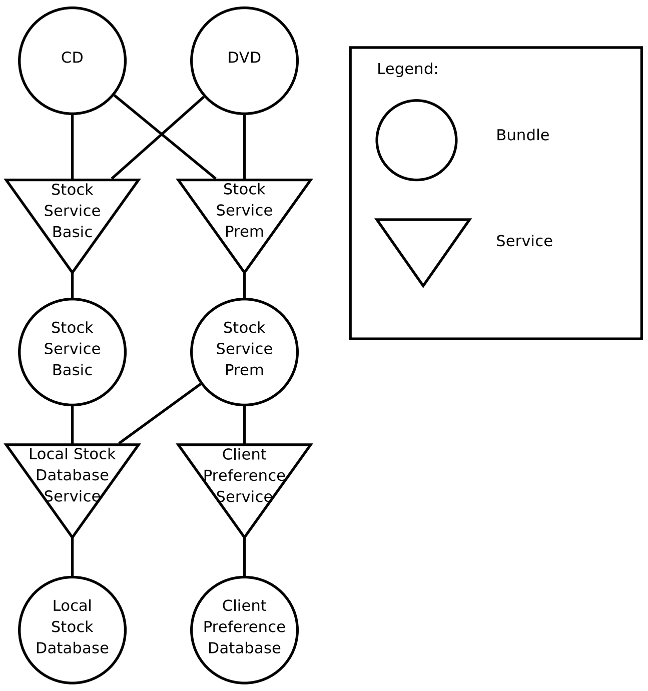

Figure 5.1: Case Study Components

The set of OSGi bundles used by our application is depicted in Figure 5.1. We consider two bundles that implement the presentation layer for an on-line store that sells CDs and DVDs, the bundles *CD* and *DVD*. These bundles are implemented as individual bundles that register servlets and resources with our altered version of the Pax Web bundle. 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 provides an interface to search for items that exist in a local database. There are two independent implementations of this service provided by two different bundles that offer this service with distinct trade-offs between quality of service and resource consumption.

In more detail, the first implementation of the stock service, simply named *Stock Service Basic*, only resorts to the internal database to provide information about products, using the *Database Service* provided by the bundle *Local Stock Database*. The second implementation of this service, named *Stock Service Premium*, additionally relies on a costumer preferences service (the *Client Preference Service* provided by the bundle *Client Preference Database*), 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.

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 *Stock Service Basic* implementation, than to provide the Premium service to a subset of clients and drop the remaining requests. Naturally, when the workload 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 Service* implementation used by this service to use the *Stock Service Basic*, and continue to use the *Stock Service Premium* implementation for *DVD* buyers. As we will show, the A-OSGi architecture provides support to specify and implement this sort of policies.

5.2 Using A-OSGi

We now describe how A-OSGi can be used to implement the policy described above for our case study.

Since A-OSGi offers the flexibility to choose which services should be monitored, we only isolated the web bundles, i.e the *CD* and *DVD* bundles in order to achieve a indirect accounting of resource consumption. We chose this approach since we want to account the resource consumption to the *CD* or *DVD* bundles only, so the resources consumed by the other bundles (*Stock Service*, *Database Service* and *Preferences Database*) are accounted to the web bundle that invoke them. This approach also reduces the monitoring overhead to a minimum.

The system goal policies described above can be described by only two ECA rules, depicted in Listings 5.1 and 5.2, respectively. Generically, the first rule presented in Listing 5.1 simply prohibits any web bundle (i.e the *CD* or *DVD*) that is consuming more than 35% of CPU from using the *Stock Service Premium* implementation of the *Stock Service.* A more detailed description follows:

- In line 1, a new ECA rule instance is created, using the ECA rule factory object and is assigned to the *newpolicy* variable.
- In line 2, the event that will trigger the evaluation of the ECA rules is set, sending the message "event" with the event object as parameter to the ECA rule object.
- In lines 3-7, the condition is set sending a "condition" message with a PonderTalk block as parameter; this block has two parameters, the same of the event, i.e *bundleID* and *value*. It is only evaluated when the event occurs and it verifies if the *Stock Service* implementation that the target bundle is using is the *Stock Service Premium* and if the CPU consumption value is greater than 35%. This condition could be further optimized, evaluating the bundles binding after the CPU consumption test. However, for the sake of simplicity it is presented this way.
- In lines 8-12, the action is set sending an "action" with a block as parameter. This action sets a prohibition on the target bundle for the currently using service, that is the *Stock Service Premium* because we tested it in the "condition" phase. With this prohibition set the target bundle will use the other *Stock Service* implementation, i.e the *Stock Service*

Basic. However in our prototype is necessary to restart the target bundle in order for it to find the new service.

```
 1 newpolicy := root/factory/ecapolicy create.
 2 newpolicy  event: root/event/bundleCPU;
 3   condition: [:value :bundleID |
 4     usedstockservice := ((bundleID getUsedServiceIDsbyName: "pt.
          mediaportal.stock.StockService") at: 0).
 5     usedstockbundle := (usedstockservice getServiceBundle).
 6     stockPremi := (framework getBundleID: "pt.mediaportal.stock.Premium"
          ).
 7     (value > 35) & (usedstockbundle == stock1bundle) ];
 8   action:    [:value :bundleID |
 9     usedstockservice := ((bundleID getUsedServiceIDsbyName: "pt.
          mediaportal.stock.StockService") at: 0).
10     bundleID setClientProhibition: usedstockservice.
11     bundleID stopBundle.
12     bundleID startBundle ].
```

Listing 5.1: Case Study ECA rule 1

The second rule presented in Listing 5.2 removes this prohibition when a web bundle uses less than 5% CPU, in order to return the sytem to initial state, where both the *CD* or *DVD* bundle use the *Stock Service Premium.* A more detailed description follows:

- In line 1, a new ECA rule instance is created, using the ECA rule factory object.
- In line 2, the event is set the same way as explained before.
- In lines 3-7, a condition is set that verifies if the used service by the target bundle is the *Stock Service Premium* and the CPU consumption value is lesser than 5%.
- In lines 8-12, an action is set that removes the existing prohibition on the target bundle for the used *Stock Service*, that as tested before is the *Stock Service Basic.* The target bundle is restarted so it finds the former prohibited service and binds to it.

These policies ensure that the most expensive implementation is used, if and only if, the resources are enough to sustain the current load. The adequate thresholds for the CPU usage were determined experimentally.

Adaptation is performed with bundle-level granularity, i.e the adaptation is performed over specific bundles and not to all the application. The way the rules are specified does not require

```
newpolicy := root/factory/ecapolicy create.
newpolicy  event: root/event/bundleCPU;
  condition: [:value :bundleID |
    usedstockservice := ((bundleID getUsedServiceIDsbyName: "pt.
       mediaportal.stock.StockService") at: 0).
    usedstockbundle := (usedstockservice getServiceBundle).
    stockBasic := (framework getBundleID: "pt.mediaportal.stock.Basic").
    (value < 5) & (usedstockbundle == stock2bundle) ];
  action:    [:value :bundleID |
    usedstockservice := ((bundleID getUsedServiceIDsbyName: "pt.
       mediaportal.stock.StockService") at: 0).
    bundleID removeClientProhibition: usedstockservice.
    bundleID stopBundle.
    bundleID startBundle ].
```

Listing 5.2: Case Study ECA rule 1

the CD or DVD web bundles to be named explicitly. Therefore, in run-time, depending on the system load, they may be applied to the CD service, to the DVD service, or both. This is possible because the KC component maintains updated information about each bundle, specifically on their bindings.

As stated earlier it is necessary to restart the target bundle so the prohibitions defined are reevaluated. This is done to force iPOJO to reevaluate the bindings of the target bundle, taking into consideration the new set of rules in the system. This is a practical approach for the implementation of our prototype, that we could be avoided introducing modifications in iPOJO that would monitor the setting of prohibitions/obligations.

5.3 Performance Evaluation

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 described above. The other machine is used to generate the workload using Apache JMeter 2.3.2 to emulate clients executing requests to the HTTP 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. Hence two types of requests are defined: one that requests a list and an item from the DVD bundle and other that requests a list and a item from the CD bundle.

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.
- The CD/DVD+ workload, in addition to the previous requests, imposes a load of more 1500 requests per second to the DVD service.
- Finally, the CD+/DVD+ workload includes an excess of 700 requests per second to the CD service.

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 5.2 (time is measured in seconds).

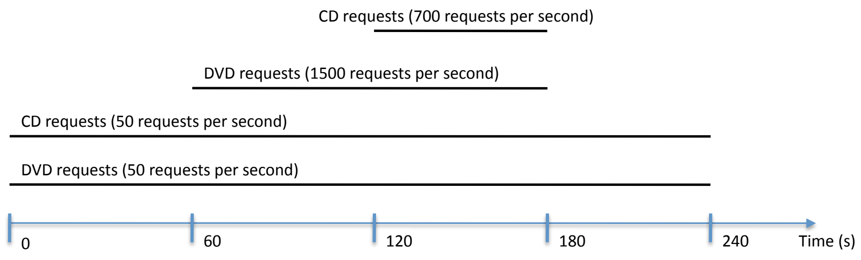
CD requests (700 requests per second)
DVD requests (1500 requests per second)
CD requests (50 requests per second)
DVD requests (50 requests per second)
0
60
120
180
240
Time (s)

Figure 5.2: Workload Description

The CD/DVD workload is low enough such that the Premium implementation of the stock service can be used to answer all requests without overloading the system.

To sustain the CD/DVD+ workload, one is required to adapt the DVD bundle to change its use of the *Stock Service* implementation to the *Stock Service Basic* implementation (the CD bundle do not need to be affected by the adaptation at this point).

When the CD+/DVD workload starts, both the DVD and CD bundles are required to use the *Stock Service Basic* implementation of the *Stock Service* to sustain the heavy load.

The results obtained are depicted in the following figures, where we compare different metrics between executing the case study in A-OSGi and in an original OSGi platform without modifications and consequently without adapting the usage of different *Stock Service* implementations.

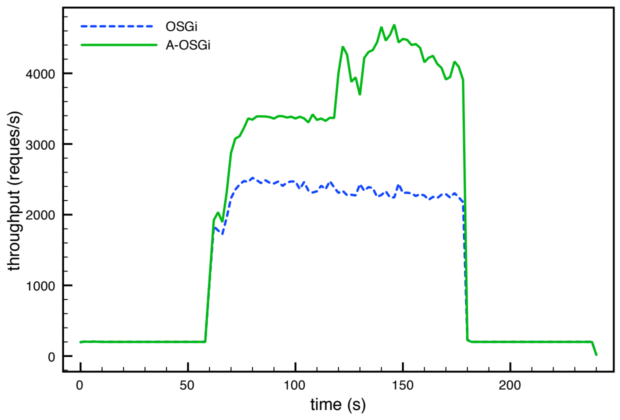

Figure 5.3: Throughput

The first plot 5.3 compares the performance (in terms of throughput) of original OSGi in a static configuration that provides the *Stock Service Premium* with the autonomic configuration provided by A-OSGi. Until the 60 seconds of the case study the two configurations behave very similar, since they are using the same services implementation. However at 60s and 120s adaptation occurs and the A-OSGi configuration is capable of responding to the increasing requests rate. 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 service.

The plot 5.4 represents an estimate on the quality of service (QoS) provided to the client. The adaptations performed by A-OSGi approximately at 60 and 120 seconds reduce the QoS since the *Stock Service* implementation used is changed to the *Stock Service Basic*, while at 180 seconds the QoS is reverted to the original value, since the implementation used is changed to the *Stock Service Premium*. The better throughput observed in Figure 5.3 is a result of the lower QoS offered.

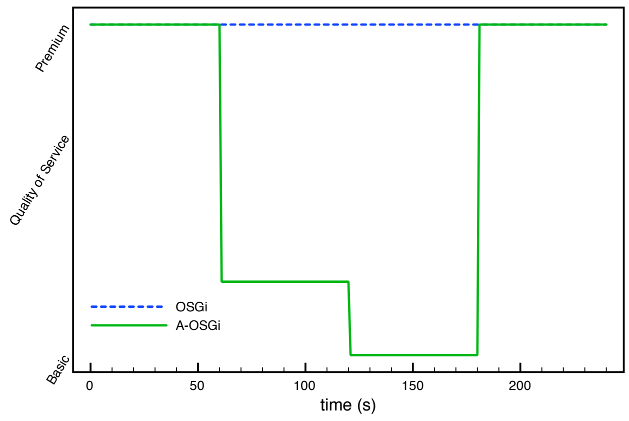

Figure 5.4: Quality of Service

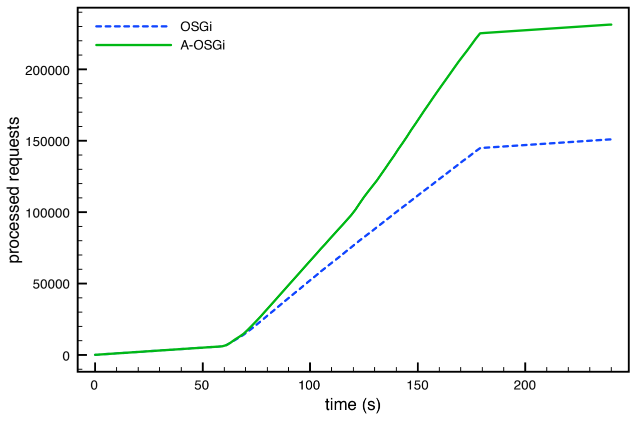

Figure 5.5: Processed Requests

Plot 5.5 depicts the total number of requests processed by both configurations. As the throughput comparison the number of processed requests presents a better evolution in the A-OSGi configuration than the static OSGi configuration, as can be perceived by the function gradient, that increases at 60 and 120 seconds and decreases at 180 seconds when adaptations occur. Clearly the number of requests processed in this period is greater with the A-OSGi configuration.

Finally, plot 5.6 compares the average request latency of the application running in the A-OSGi framework but without policies active, i.e no adaptations occur, against the same application, running in a plain OSGi framework. The workload is constant and the request rate

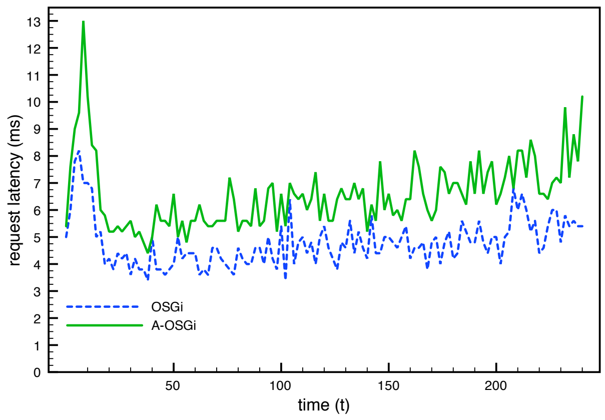

Figure 5.6: Overhead

does not overload the server.

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 fully optimized (in particular the isolation mechanisms required for detailed bundle grain monitoring).

5.4 Other Alternative Policies

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 in runtime making available or unavailable some functionality of the system.

Summary

In this chapter we have presented an experimental evaluation of our A-OSGi prototype. This evaluation was performed emulating a dynamic workload over an OSGi application, namely an online store implement with OSGi. We described adaptation rules that change the service implementation used by some bundles in order to increase the number of requests processed according to the resource consumption of the target bundles.

6 Conclusions

In this dissertation 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 described architecture and mechanisms have been implemented in the form of a prototype. The A-OSGi prototype was experimentally evaluated and the results have illustrated the benefits of the approach: we were able to selectively adapt the implementation of a service used by different bundles, in order to augment the system performance in face of dynamic workloads.

As future work, we consider the optimization of 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 monitorization mechanisms.

Still in the MAC component, the improvement of its analysis functionality would be an important improvement by providing the ability to express event correlation with a high level language, similar to the Event Distiller of the Kinesthetics eXtreme project (Kaiser, Parekh, Gross, & Valetto 2003).

References

Ananthanarayanan, R., M. Mohania, & A. Gupta (2005). Management of conflicting obligations in self-protecting policy-based systems. *Autonomic Computing, International Conference on 0*, 274–285.

Bigus, J. P., D. A. Schlosnagle, J. R. Pilgrim, & W. N. Mills III (2002). Able: A toolkit for building multiagent autonomic systems. *IBM Syst. J. 41*(3), 350–371.

Brewer, E. A. (2001). Lessons from giant-scale services. *IEEE Internet Computing 5*(4), 46–55.

Diaconescu, A., J. Bourcier, & C. Escoffier (2008, Oct.). Autonomic ipojo: Towards self-managing middleware for ubiquitous systems. In *Networking and Communications, 2008. WIMOB '08. IEEE International Conference on Wireless and Mobile Computing,*, pp. 472–477.

Diao, Y., N. Gandhi, J. Hellerstein, S. Parekh, & D. Tilbury (2002). 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*, 219–234.

Diao, Y., J. L. Hellerstein, S. Parekh, & J. P. Bigus (2003). Managing web server performance with autotune agents. *IBM Syst. J. 42*(1), 136–149.

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

Felix Apache. Homepage. `http://felix.apache.org/`.

Ferreira, J., J. Leitão, & L. Rodrigues (2009, September). A-osgi: A framework to support the construction of autonomic osgi-based applications. In *Proceedings of the Third International ICST Conference on Autonomic Computing and Communication Systems*, Limassol, Cyprus, pp. (to appear).

Garlan, D. & B. Schmerl (2002, 18-19 November). Model-based adaptation for self-healing systems. In *ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02),*, Charleston, SC.

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

Gruber, O., B. J. Hargrave, J. McAffer, P. Rapicault, & T. Watson (2005). The Eclipse 3.0 platform: Adopting OSGi technology. *IBM Systems Journal*.

Hargrave, B. J. & P. Kriens. Upgrading osgi. available from `http://developers.sun.com/learning/javaoneonline/sessions/2009/pdf/TS-4966.pdf`.

Huebscher, M. C. & J. A. McCann (2008). A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv. 40*(3), 1–28.

Hulaas, J. & W. Binder (2008). Program transformations for light-weight cpu accounting and control in the java virtual machine. *Higher Order Symbol. Comput. 21*(1-2), 119–146.

IBM (2001). Autonomic computing: Ibm's perspective on the state of information technology. *IBM Journal*.

IBM (2006). An architectural blueprint for autonomic computing, fourth edition. Technical report, IBM.

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

Jung, G., K. Joshi, M. Hiltunen, R. Schlichting, & C. Pu (2008, June). Generating adaptation policies for multi-tier applications in consolidated server environments. *Autonomic Computing, 2008. ICAC '08. International Conference on*, 23–32.

Kaiser, G., J. Parekh, P. Gross, & G. Valetto (2003, June). Kinesthetics extreme: an external infrastructure for monitoring distributed legacy systems. In *Autonomic Computing Workshop, 2003*, pp. 22–30.

Kephart, J. O. (2005). Research challenges of autonomic computing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, New York, NY, USA, pp. 15–22. ACM Press.

Kephart, J. O. & D. M. Chess (2003). The vision of autonomic computing. *Computer 36*(1), 41–50.

Knopflerfish. Homepage. `http://www.knopflerfish.org/`.

Miettinen, T. (2008). *Resource monitoring and visualization of OSGi-based software components.* Ph. D. thesis, VTT Technical Research Centre of Finland.

Minsky, M. (1988, March). *Society of Mind* (Touchstone. ed.). Simon & Schuster.

OSGi Alliance (2007a). *OSGi Service Platform Core Specification, Release 4, Version 4.1.* OSGi Alliance.

OSGi Alliance (2007b). *OSGi Service Platform Service Compendium, Release 4, Version 4.1.* OSGi Alliance.

OSGi Alliance (2009). *OSGi Service Platform Core Specification, Release 4, Version 4.2.* OSGi Alliance.

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

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

Spring Source (2009). Spring Dynamic Modules for OSGi. `http://www.springsource.org/osgi`.

Sun Microsystems (2003). JAR File Specification. `http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html`.

Sun Microsystems (2006a). Enterprise javabeans 3.0 specification. [Online]. Available: http://java.sun.com/products/ejb/. [Accessed: Jun. 19, 2009].

Sun Microsystems (2006b). Java Management Extensions. `http://java.sun.com/javase/6/docs/technotes/guides/jmx/index.html`.

Sun Microsystems (2008). Sun GlassFish Enterprise Server v3 Prelude Release Notes. `http://docs.sun.com/app/docs/coll/1343.7`.

Twidle, K., E. Lupu, N. Dulay, & M. Sloman (2008, June). Ponder2 - a policy environment for autonomous pervasive systems. pp. 245–246.

Valetto, G. & G. Kaiser (2003, May). Using process technology to control and coordinate software adaptation. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pp. 262–272.

van der Mei, R., R. Hariharan, & P. Reeser (2001). Web server performance modeling. *Telecommunication Systems*.

Zhang, S., I. Cohen, M. Goldszmidt, J. Symons, & A. Fox (2005, June-1 July). Ensembles of models for automated diagnosis of system performance problems. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pp. 644–653.