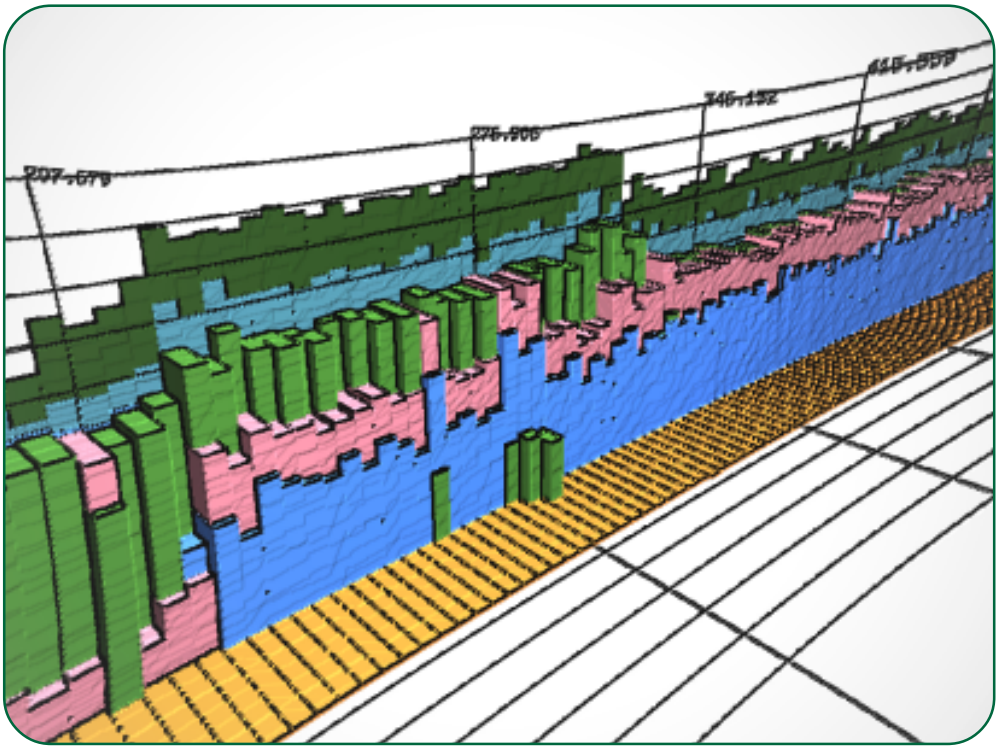
ESPOO 2008 VTT PUBLICATIONS 685



Tuukka Miettinen

Resource monitoring and visualization of OSGi-based software components

VTT PUBLICATIONS 685

Resource monitoring and visualization of OSGi-based software components

Tuukka Miettinen

ISBN 978-951-38-7104-8 (soft back ed.)
ISSN 1235-0621 (soft back ed.)

ISBN 978-951-38-7105-5 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 3, PL 1000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 3, PB 1000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 3, P.O. Box 1000, FI-02044 VTT, Finland
phone internat. +358 20 722 111, fax + 358 20 722 4374

VTT, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde 020 722 111, faksi 020 722 2320

VTT, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel 020 722 111, fax 020 722 2320

VTT Technical Research Centre of Finland, Kaitoväylä 1, P.O. Box 1100, FI-90571 OULU, Finland
phone internat. +358 20 722 111, fax +358 20 722 2320

Edita Prima Oy, Helsinki 2008

Miettinen, Tuukka. Resource monitoring and visualization of OSGi-based software components [OSGi-pohjaisten ohjelmistokomponenttien resurssien kulutuksen monitorointi ja visualisointi]. Espoo 2008. VTT Publications 685. 107 p. + app. 3 p.

Keywords resource consumption, resource monitoring, software visualization, performance analysis

Abstract

This work introduces a novel approach for the resources consumption analysis of OSGi-based software components. OSGi Service Platform provides a component based and service-oriented Java environment that is especially emerging in environments with constrained computational resources. OSGi Service Platform enables the cooperation of multiple Java based components within a single Java Virtual Machine. Existing JVM analyzing tools typically monitor the resource consumption of the whole Java environment, which is not sufficient in an OSGi environment since the JVM conceals the resource consumption information of separate OSGi components. This emphasizes the need for monitoring solutions that are able to provide a detailed view of the resource consumption of the Java environment.

Tools implemented in this work enable the effective resource consumption analysis of individual software components executed on a OSGi platform. A monitoring tool that is able to identify the resource consuming component was developed to extract both component and environment specific data from the Java environment. An existing visualization tool was extended in order to provide an easy to understand view of the resource consumption behaviour of both single component and component compositions. Two novel visualizations were introduced to facilitate the analysis of software resource usage. The tool produces 3D visualization that simultaneously illustrates the time related CPU utilizations and memory consumptions of all desired components executed on the OSGi platform. The other novel visualization presents the amount of resources required by a component to operate normally. In addition, it enables the comparison of resource consumption information to desired usage boundaries. The OSGi-based resource monitoring service was also developed in order to

provide runtime resource consumption information for components that are able to adapt their behaviour according to available computing resources.

The applicability of the tools was demonstrated with two use cases. Firstly, an OSGi component's resource usage boundaries were detected and validated. Secondly, multiple components were monitored and use of the resource monitoring service was demonstrated with an adaptive OSGi component. It was proved that implemented tools effectively reveal how the components behave inside the OSGi environment from a resource consumption perspective.

Miettinen, Tuukka. Resource monitoring and visualization of OSGi-based software components [OSGi-pohjaisten ohjelmistokomponenttien resurssien kulutuksen monitorointi ja visualisointi]. Espoo 2008. VTT Publications 685. 107 s. + liitt. 3 s.

Avainsanat resource consumption, resource monitoring, software visualization, performance analysis

Tiivistelmä

Tässä työssä luotiin uudenlainen lähestymistapa OSGi-pohjaisten ohjelmistokomponenttien laskentaresurssien käytön analysointiin. OSGi-palvelualusta tarjoaa komponenttipohjaisen ja palvelusuuntautuneen Java-alustan, joka on herättänyt kasvavaa kiinnostusta erityisesti resurssirajoitteisten tietokoneympäristöjen markkinoilla. OSGi-ohjelmistoalusta mahdollistaa useiden Java-pohjaisten ohjelmistokomponenttien yhteistoiminnan samassa Java-virtuaalikoneessa. Olemassa olevat Javan analysointityökalut tarkkailevat koko Java-ympäristön resurssien kulutusta, mikä ei ole riittävää OSGi-ympäristössä, koska virtuaalikone kätkee yksittäisten ohjelmistokomponenttien resurssienkulutuksen. Tämän vuoksi tarvitaan uusia resurssien käytön seurantaratkaisuja, joiden avulla saadaan yksityiskohtaisempi kuva Java-ympäristön resurssien käytöstä.

Tässä työssä kehitetyt työkalut mahdollistavat yksittäisten ohjelmistokomponenttien laskentaresurssien käytön tehokkaan analysoinnin. Kehitetty resurssien monitorointityökalu tarkkailee koko Java-ympäristöä ja pystyy erottelemaan laskentaresurssien käytön komponenttikohtaisesti. Olemassa olevaa visualisointityökalua laajennettiin, jotta kerätty tieto voidaan esittää helposti ymmärrettävässä muodossa. Työssä esitellään kaksi uudenlaista visualisointia, jotka helpottavat ohjelmiston resurssien käytön analysointia. Visualisointityökalu tuottaa kolmiulotteisen näkymän, joka yhtäaikaisesti esittää haluttujen OSGi-komponenttien tuottaman prosessorikuorman ja muistinkulutuksen. Toinen uusi visualisointi esittää laskentaresurssien määrän, jotka ohjelmistokomponentti vaatii toimiakseen. Tämä myös mahdollistaa komponentin resurssienkulutuksen vertailun haluttuihin käyttörajoihin. Työssä kehitettiin myös OSGi-pohjainen laskentaresurssien monitorointipalvelu, joka mahdollistaa resurssien käyttötiedon ajonai-

kaisen hyödyntämisen. Tämä taas mahdollistaa vapaana oleviin laskentaresurs-seihin mukautuvat ohjelmistokomponentit.

Työkalujen hyödyllisyys osoitettiin kahdella erilaisella käyttötapauksella. Ensimmäisessä etsittiin ja vahvistettiin erään OSGi-komponentin resurssien kulutusrajat. Toisessa tapauksessa useita komponentteja monitoroitiin ja havainnollistettiin resurssienmonitorointipalvelun käyttöä mukautuvan komponentin avulla. Näin pystyttiin osoittamaan, että kehitetyt työkalut paljastavat tehokkaasti komponenttien käyttäytymisen OSGi-ympäristössä resurssien kulutuksen näkökulmasta.

Preface

The research work for this thesis was made as a part of the ITEA-ANSO (Autonomous Networks for SOHO Users) and ITEA2-CAM4Home (Collaborative Aggregated Multimedia for Digital Home) projects at the VTT Technical Research Centre of Finland.

I am deeply grateful to Mr. Mika Hongisto for providing valuable discussions, reviews and suggestions during the work. I wish to thank my supervisor at the University of Oulu, Professor Tapio Seppänen for finding time to review and comment on this work.

My colleagues at the VTT Technical Research Centre of Finland deserve my warmest thanks and especially all members of the Performance Architectures team for pleasant moments both during and after working hours. I would like to express my gratitude to Mr. Juho Perälä for his excellent technical assistance throughout the implementation part of the work. In addition, I owe my gratitude to Mr. Daniel Pakkala for discussions and suggestions provided during both the research and writing processes.

Finally, I thank you Johanna, for sharing your everyday life with me and for your encouragement and faith in me that has pulled me through times of desperation during the writing of this work.

Oulu, May 22th, 2008

Tuukka Miettinen

Contents

Abbreviations

API	Application programming interface
BCI	Byte Code Insertion / Byte Code Instrumentation
CBSD	Component-based software development
COTS	Commercial off-the-self
CPU	Central Processing Unit
GCM	Generic Communication Middleware
GUI	Graphical User Interface
IDE	Integrated Development Environment
JAR	Java Archive
J2ME	Java 2 Platform, Micro Edition
J2SE	Java 2 Platform, Standard Edition
JDK	Java Development Kit
JMX	Java Management Extensions
JNI	Java Native Interface
JRE	Java Runtime Environment
JSR	Java Specification Request
JVM	Java Virtual Machine
JVMPI	Java Virtual Machine Profiler Interface
JVM TI	Java Virtual Machine Tool Interface
MB	Megabyte
MBean	Managed Bean
OS	Operating System
OSGi	Open Service Gateway Initiative
PCCT	Partial Calling Context Tree
QoS	Quality of Service
RTT	Round-trip time

1. Introduction

A rapidly increasing number of diverse devices, functional demands, shortened product cycles and pervasive networking has increased the complexity of software development [1, 2, 3, 4, 5]. Platform-independent, component-based, and service-oriented middleware-based software systems have been proposed as one way in addressing these challenges [2, 5]. In this work, focus is on the OSGi Service Platform [6] targeting to fulfil the above proposal. OSGi enables the integration of functionally pre-tested software blocks but creates new challenges to testing and monitoring the software system in contrast to the traditional system. We must be able to observe and evaluate the behaviour of software components when integrated with the other components and deployed in the actual execution platform [3, 5]. These challenges raise problems that are the main research goals of this thesis work:

- Provide a tool that measures the resource consumption of a single software component.
- Provide a visualization view to enable the analysis–both single component and component composition–from a resource consumption behaviour perspective.

Software development is typically performed without much concern laid on the actual target platform. For this reason, run-time resources utilized by software will remain unclear to the software designer. The OSGi environment emphasizes this problem since it runs on top of both the Java 2 Standard Edition (J2SE) and Java 2 Micro Edition (J2ME), and the OSGi applications should be executable on both of these platforms. OSGi applications are supposed to collaborate with other components implemented by different vendors. The design of good quality component compositions requires information on the component's performance in addition to the knowledge of its functional behaviour. Therefore, it is essential for designers to gain a detailed and individual view of the software component's resource consumption in its real executing environment. Without this knowledge, it is hard to determine whether the implementation of the application meets the resource requirements set at the specification phase or deployment phase. This

leads to an unawareness of the correct behaviour of an application and makes it deployment essentially more difficult.

OSGi is a Java based middleware. The Java programming language has advantages in resource constrained devices, considering its portability, enhanced safety and the potential for run-time optimization [7]. Java is an object-oriented and interpreted language with automated memory management, which offers reusable software components. Object orientation and a high abstraction of internal activities are the main reasons for these benefits; however these also bring weaknesses when dealing with resource constrained devices. An erratic use of objects and the transparency of the target platform in the programming phase typically lead to an excessive consumption of computing resources. This results in an undesired behaviour of the applications that eventually ruins the performance of the whole system.

The current profiling and monitoring tools for Java, such as [8, 9], observe the whole Java Virtual Machine (JVM) and not individual applications. This observation level is not sufficient in the OSGi environment. Therefore, we require a tool that can extract application specific behaviour information out of the JVM, in order to establish a view of the application's behaviour. Collecting this detailed information using pure Java is currently impossible, without modifying the JVM itself, due to Java's lack of low-level abilities. Fortunately, since the version 5.0 Java has provided a native interface for monitoring and debugging purposes: Java Virtual Machine Tool Interface (JVM TI). Using this interface, a developer can create monitoring tools written in native language like C/C++ to collect runtime information from the JVM and directly from the underlying operating system (OS). Even though this enables the writing of monitoring tools that collect detailed runtime behaviour data, it is still a challenging task to gather fine-grained statistics without eating away at the performance of the JVM.

Even the most detailed runtime data is useless without a way to represent it. Collected resource consumption data must be effectively represented to get the entire benefit out of the monitoring tools. Typically, a behavioural log simply consists of ASCII debug print data. It is time consuming and aimless to go through the logs without interpreting those to a form that a human can easily analyze. If the

representation is confusing or unpractical, it leads to difficulties in analyzing, or in the worst case it leads to the fact that no one is interested in analyzing these statistics. Behaviour statistics that have more than one measured attribute increase the complexity of representation. 3D visualization can be a powerful tool when representing statistics that holds more than one attribute. An appropriate 3D graph of software resource consumption highlights the actual runtime behaviour and correlation of different attributes on one single graph.

1.1 Motivation

The fact that all OSGi applications run on a single JVM introduces a monitoring problem. With the current monitoring tools, it is almost impossible to identify the actual source of resource consumption without a detailed knowledge of all the running applications. This situation leaves application developers with just a trace of the resource consumption behaviour of their application in its real executing environment and creates the need for new monitoring solutions.

A method for gathering individual resource consumption statistics would be a powerful tool, when integrated into the whole cycle of application development. With a proper monitoring tool and a practical visualization of the gathered data, we can without excessive effort validate the application's resource usage against boundaries set during the specification phase. This would enable a rapid loop from prototype implementation back to re-implementation or even back to the redesign.

Monitoring and analyzing tools that are able to provide resource consumption statistics of individual software components can be used to define the need for available resources in order to ensure the functioning of a component. These boundaries could be used as one attribute in the component deployment phase. Boundaries would enable the comparison of the component's performance to other components, which provide the same functionality. In addition, these enable resource consumption assessment when building component compositions.

Adaptive applications that can conform to the constant changes of available resources require support from the execution environment; otherwise, portability

will be lost because a platform-specific code is always required for precise resource monitoring. An OSGi compliant resource monitor has been introduced [10]. This implementation is based on monitoring the whole JVM resource consumption and self-adapting applications. In a case where the framework controls adaptive software from exhausting the whole environment, there has to be detailed information of the application's usage of physical resources.

The OSGi platform provides a unique environment to test applications targeted at small devices. Limitations in profiling support in the J2ME make it difficult to analyze their behaviour in this environment. Since software developed for the OSGi platform is supposed to run unmodified in different execution environments, it creates the possibility to analyze their behaviour in the J2SE. Naturally, measured resource consumption information must be scaled according to the corresponding platform.

1.2 Scope and structure of this work

The focus of this thesis work is on discovering a resource consumption measuring method and finding an effective way in representing this information. The aim is not to analyze the behaviour of specific OSGi bundles, but rather to develop tools that enables comprehensive and precise analyses.

To achieve these research targets, the thesis is structured as follows. Chapter 1 introduces the related software environments and the motivation for this work. In Chapter 2 the OSGI environment, different computing resources, and the available techniques for monitoring Java applications are studied. These are discussed particularly from an OSGi perspective. From Chapter 3 and onwards, the actual work done by author begins. Chapter 3 discusses the requirements and implementation issues of the tools developed in this thesis work. Chapter 4 introduces implemented tools for monitoring resource consumption of an OSGi bundle and the post-processing of this information. Chapter 5 presents two different use cases where the implemented tools are used. Chapter 6 presents the evaluation of the implementation and a discussion on the applicability of the whole work. Finally, Chapter 7 summarizes the thesis work, Chapter 8 represents the references, and Chapter 9 represents the appendices.

2. Related technologies and research

2.1 Component-based software development

Emerging component-based software architectures promise to provide the better re-use of software components, greater flexibility, scalability and a higher quality of services (QoS). Over the last decade, component based software development has changed from a purely scientific research field to a widely used technique. The component-based software technology converts the system design into assembling the existing components or creating new reusable ones. This makes a system easy to implement, convenient to manage and flexible to update. Although component-based development offers many potential benefits, it also raises several issues that developers need to consider, a comprehensive classification of the issues have been made in [11]. [12]

By definition, software components are the units of software deployment. A software component is a closed module that is responsible for implementing application behaviours. It communicates with its interfaces. In designing a component, the implementation is separated from the interfaces. The implementation is secret while the interface is public, hence hiding the implementation of a component from its environment. A software component can also be independently deployed and is subject to third-party composition. [13]

As was previously mentioned, the software development process in component-based software development (CBSD) differs from the traditional process. It is obvious that the developer of the component must deal with the performance, re-usability and portability issues in order to provide qualitative software. However, components are typically developed in a heterogeneous environment, which means that their actual target platform is not known during the time of development. Individual components can also be used by many assemblers for innumerable purposes in multiple applications [13]. This leads to an increasing complexity of the testing process, as the developer is forced to test the component without knowing exactly how it ultimately will be used and by whom.

CBSD introduces a completely new role for the development process, component integrator or assembler. The main responsibility of the component integrator is to integrate individual components into working compositions. The biggest challenge on the assembling of compositions is the selection of components. There are numerous aspects that the integrator must consider in the process of component selection, it would be impossible to go through all of these comprehensively. Therefore, we will focus on the quality aspects of the components and particularly the performance aspects.

The selection process includes a comparison among the components, which provide the desired functionality. When considering quality issues, it is not sufficient for the integrator to compare the quality of individual components. Integrators must consider how the composition inherits the quality attributes of the separate components. Combining the performance attributes of the components is unpredictable and causes a major challenge to integrators. Approaches have been introduced to make the predictive performance analysis of component-based systems, such as [14, 15].

In order to enable the quality comparison in the selection process, a characterization of the component's quality attributes must be made. Bertoa and Vallecillo have defined a set of quality attributes and their associated metrics for effective evaluation components in [16]. This set is made for commercial off-the-self (COTS) components but the set applies to all software components. The set is comprised of functionality, reliability, usability, efficiency and maintainability. Efficiency, in this classification, can be seen as a performance attribute. This is because efficiency is comprised of the runtime performance attributes of a component, such as response time, throughput and resource utilization.

A uniform quality model for all available components would enable a comprehensive comparison and evaluation of the components in the selection process. The kind of assurance that software components conform to well-defined standards is called certification. A certificated component is evaluated and validated to be trustworthy by a third party. Academic research has been made in order to provide a standard certification process for a component. Alvaro et al. have made an extensive survey in the state-of-the-art component certification area [17]. Based

on the survey, they proposed a software component certification framework [18] in order to achieve a well-defined software component evaluation process.

2.2 OSGi

OSGi Alliance, formerly known as the Open Service Gateway Initiative, is a global consortium of about 80 companies with a diverse branch of businesses. Together, they create specifications and reference implementations aimed at non-proprietary universal middleware. The OSGi specifications define a standardized and component oriented computing environment for networked services that is the foundation of an enhanced service oriented architecture [19]. OSGi was originally designed to fit the needs of embedded devices and home service gateways, which require to be managed remotely. Due to an increasing interest among developers, it has recently been used far beyond the context of mobile embedded gateways. The OSGi Service Platform has become an attractive environment for both application and system developers, as it has interesting features: portability, deployment format, resource sharing and life cycle management, to name but a few. These features added to the fact that the OSGi Service Platform can be executed on top of the J2ME, has enabled it to emerge in the field of embedded systems.

2.2.1 OSGi Service Platform

The OSGi Service Platform Specification consists of three parts: the framework, core services and optional service compendium. A key component of the specifications is the framework. The framework handles all the necessary actions needed to transform the JVM from a single application environment to a multiple application environment where all the applications run inside the same JVM. Figure 1 presents the transformation enabled by the OSGi framework.

The framework provides a general-purpose, secure, and managed Java framework that supports the deployment of extensible and downloadable applications known as OSGi bundles [20]. Technically, the OSGi framework can be seen as a custom, dynamic Java class loader and a service registry that is globally accessible within a single Java virtual machine. An architectural view of the OSGi environment is shown in Figure 2.

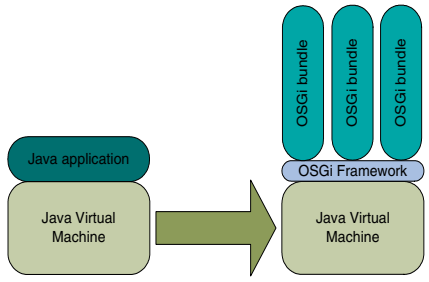


Figure 1. The OSGi Framework transforms the JVM from a single to a multiple application execution environment.

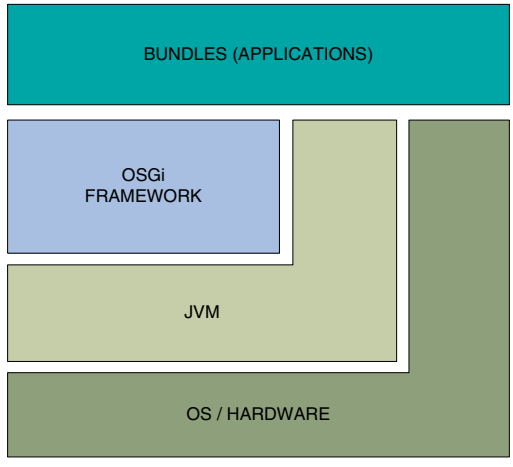


Figure 2. Architecture of the OSGi environment.

An OSGi bundle, shown in Figure 1, is a Java Archive (JAR) that encloses an executable code, code libraries and a description of the bundle into a one packet. In addition to these, JAR can also enclose the resources needed by an application. Typical resources are images and sounds. With the information that the description provides, the framework can manage the bundle's life cycle. Figure 2 shows that bundles are not restricted to use in only the OSGi Framework, but they can directly access the resources provided by the standard Java libraries and the operating system of the underlying hardware. Bundles are just like normal Java

applications with the exception that the usual starting point of the main-method must be replaced with an activator-class, implementing the OSGi BundleActivator interface. From now on, applications running in the OSGi environment will be referred as bundles.

The functionality of the Framework can be divided into three layers. The module layer specifies a class loading model which adds modularization to the Java. The life cycle layer adds bundles that can be dynamically installed, started, stopped, updated and uninstalled. This layer also specifies the dependency resolving between the bundles. The service layer defines the dynamic cooperation model for bundles. The OSGi uses the publish, find and bind model. All these layers also extend the security model of the Java 2 to provide a secure environment for collaborating bundles.

The OSGi Alliance has specified a number of services that run on top of the framework. There are two main configurations for these services, Core Specification and Mobile Specification. These services are left as an implementer's choice; even core services are marked as optional in the specification. Table 1 lists the mandatory services as defined in the specifications [20, 21, 22] with a short description of the service and if the service is mandatory in the other configuration. Table 2 lists all the optional services and their availability on different configurations. These services have been defined by the OSGi Alliance in the specifications [21, 22].

2.2.2 Implementations

There are several open-source projects and commercial products, which have implemented OSGi specification. The following is a short status analysis of the existing implementations of the OSGi R4 core specification. All the mentioned implementations are or will be compliant with the latest OSGi specification, OSGi R4 Service Platform. The implementations of former OSGi specifications have been excluded. At this point, there are no implementations of the OSGi R4 Mobile specification, at least not any open-source or published implementations.

Table 1. OSGi specified mandatory services on different configurations

Service	**Core spec.**	**Mobile spec.**	**Description**
Package Admin	yes	yes	Provides information on the sharing status of all packages and access to the package sharing mechanism.
Start Level	yes	no	Sets the current start level, assigns a bundle to a start level, and interrogates the current settings.
Permission Admin	yes	yes	Provides access and control to bundles permissions.
Conditional Permission Admin	yes	optional	Extends Permission Admin Service with permissions that can be applied when certain conditions are fulfilled.
URL Handlers	yes	no	Enables bundles to dynamically contribute new scheme or content handlers to the URL class.
Log	optional	yes	General purpose message logger. Consists of two parts; logging service and log reader service.
Configuration Admin	optional	yes	Provides a model to get and set configuration info.
Event Admin	optional	yes	Provides a mechanism to publish and subscribe events.
Service Tracker	optional	yes	Provides a class that tracks services for applications.
Declarative Services	optional	yes	Can read service declarations from a bundle and then register those services on behalf of the bundle.
Metatype	optional	yes	Provides a unified access point to the Meta Type information that is associated with bundles.
Deployment Admin	no	yes	Standardizes the access to the life cycle management of interlinked resources on an OSGi Service Platform.
Application Admin	no	yes	Simplifies the management of an environment with many different types of applications that are simultaneously available.

Table 2. OSGi specified optional services on different configurations

Service	**Core spec.**	**Mobile spec.**	**Description**
Device Access	optional	no	Mechanism for dynamically finding the bundle that implements a driver for a new device.
User Admin	optional	no	Service for authentication and authorization purposes.
IO Connector	optional	optional	Allows bundles to extend Generic Connection Framework (J2ME).
Preferences	optional	no	Provides access to the database of properties.
HTTP	optional	no	Web server. Runs servlets, which are provided by bundles and need to be available over HTTP.
UPnP Device	optional	no	Maps UPnP devices to the OSGi Service Registry and vice versa.
XML Parser	optional	optional	Allows bundles to locate a parser with desired properties and compatibility with JAXP.
Wire Admin	optional	no	Connects different services together as defined by a configuration.
DMT Admin	no	optional	Set of services for managing a device using concepts from the OMA DM specifications.
Monitor Admin	no	optional	Provides unified access to the Status Variables in the system, which reflect the status of an application or a device.

Apache Felix Apache Felix is an open-source implementation of the OSGi R4 specification made by the Apache Software Foundation. Felix is a successor to the former Oscar OSGi platform. By the time of making this thesis the latest Felix 1.0.0 release is not fully compliant with the OSGi R4 specification as it only partly implements security issues and lacks two core services: permission admin and conditional permission admin. [23]

Knopflerfish Knopflerfish is an open-source implementation of the OSGi R4 specification maintained and sponsored by Makewave, formerly known as Gatespace Telematics, which has several developers assigned to develop and maintain it. The latest release, Knopflerfish 2.0.0, is an almost complete implementation of the OSGi R4 specification; a minor security item and a conditional permission admin service are still missing. Knopflerfish 2 also implements almost all of the optional services specified in the service compendium [21], three optional services are missing. [24]

Eclipse Equinox Equinox is an open source implementation of the OSGi framework, managed by the Eclipse Foundation. The latest release is Equinox 3.3, which fully implements the OSGi R4 specification and almost all of the optional services, three optional services are missing. Equinox can be used as a standalone OSGi framework, but its main purpose is acting as the Eclipse's runtime environment, handling the life-cycle of all the Eclipse components and plug-ins. Equinox is the only open-source implementation of the R4 specification that has been certified by the OSGi Alliance. [25]

Knopflerfish Pro Knopflerfish Pro is a commercial OSGi framework developed by Makewave. Makewave is the sponsor and maintainer of the open source Knopflerfish framework and the Knopflerfish Pro can be viewed as the commercial version of Knopflerfish, as large parts of it are built on top of the open source alternative. The commercial product adds additional services to the open-source framework, such as UPnP, WireAdmin and Jini. [26]

mBedded Server mBedded Server is a commercial OSGi framework developed by ProSyst. The latest release, the mBedded server Professional Edition 6.1, provides the full implementation of the OSGi R4 specification and parts of the service compendium. There is also a free version of the mBedded Server available called the Equinox Edition. The latest release of this version also fully implements the OSGi R4 specification. The Equinox Edition is based on a source code from the Eclipse Equinox project with extensions from the commercial product. The mBedded Server has been certified by the OSGi Alliance. [27]

2.2.3 Computing resources of an OSGi bundle

As OSGi applications are emerging in mobile and other small devices, which are constrained in performance and physical memory, resource consumption is an essential factor in developing bundles. This chapter studies the primary resource attributes that characterize the performance of an OSGi bundle. Computing resources are physical or virtual components offered by an underlying platform, consisting of hardware and software. Software utilizes these resources to achieve such a functionality that it was designed to perform.

Resource consumption itself can be comprised as a quality or performance attribute of software. Resource utilization can be seen as defined in [28]: *the capability of software to use appropriate amounts and type of resources when the software performs its function*. Despite extensive functionality testing of software, performance testing including resource usage behaviour is often ignored or left without much concern. However, primary problems with field released software are not often crashes or other functional errors but rather performance related problems [29]. These problems are emphasized in resource constrained environments where increasing the performance by adding hardware is impossible or not practical due to high costs, energy dissipation and space issues.

We have focused thus far on three main resource utilization attributes; Central processing unit (CPU) utilization, memory consumption and network usage. These have been identified as all of these are measurable and we have concrete units of measurements for these resources. The same resources are also defined as a typical characterization of software performance in [29, 28] and the utilization of these should be taken into consideration, when trying to determine the performance of software.

Resources are observed from the perspective of a single OSGi bundle. This reflects on the resources and metrics we have chosen to observe. The metrics are chosen to represent the utilization of resources that a bundle can directly affect on its own. For example, a congestion of networking media and re-sending messages due to collisions are not things that one single bundle can affect directly.

Instead, the amount and the size of the messages sent and received by the bundle can be counted as directly bundle related issues.

CPU utilization The ability to monitor CPU consumption is a basic requirement for a run-time monitoring tool. It is also probably the most challenging resource to observe. This due to its continuous nature, as we cannot identify explicit places of CPU consumption in the source code. It is also a major challenge not to exhaust the measured system with excessive overhead when finding a way to measuring CPU consumption. [30]

CPU utilization provides the measure of time that the system CPU's are servicing an application. The purpose of measuring this attribute is to get a proportional view of the time that the CPU is busy processing during a specified interval [31]. In the OSGi environment, we can also get a valuable perspective to the bundle CPU utilization when we measure the CPU time consumed by a bundle and also the CPU time consumed by the whole JVM. Based on these measurements, we can calculate the proportion of CPU time taken by a bundle comparing to the time taken by the JVM.

CPU utilization is an attribute that is an average over the measurement period. Every application has instantaneous high CPU utilization peaks at some time during their execution and therefore the gathering of these peaks is not useful. A typical interpretation for an application's CPU utilization of a CPU intensive application is provided in Figure 3.

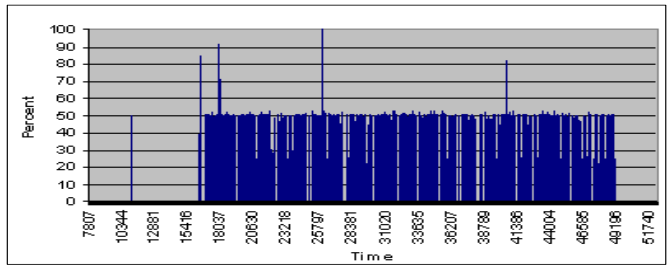


Figure 3. CPU utilization of an application.

As shown in Figure 3, an application reserves the CPU steadily when it is performing some of its activities. Without averaging the CPU utilization over the measurement period, the figure would be hard to analyze because it would be full of high peaks. Therefore, critical time spans and actual behaviour concerning CPU reservation would remain hard to identify. A measurement period must be adapted correctly, otherwise averages may not be meaningful.

The CPU execution time of an application is highly dependent on the execution platform. It gives a good view of the load caused by an application, though the measurements are only comparable within the same platform. In Java, there is a platform-independent measurement unit: bytecode. Counting the number of executed bytecodes gives a view to the CPU load of an application that is comparable with measurements made with other applications on a different platform. Counting the number of executed bytecodes has been introduced in [30].

Memory consumption Memory consumption provides a measure of the size that an application has reserved from the system memory at a specified point of time. These days, the memory consumption of an application is often ignored by developers due to its low cost in desktop environments. Nonetheless, a memory footprint is an essential factor when trying to identify the resource consumption behaviour of an OSGi bundle running in resource constrained environments. The main reasons follow:

- Many mobile devices have tight memory requirements as memory has severe implications on the cost and physical size of a device [32, 33].
- Java executions are expected to stress the memory system more than traditional programs [34, 35]. This is due to JVM features, such as garbage collection which make Java programs much more memory-intensive than normal programs.
- It has been observed [36, 37] that the memory system can produce a large proportion of the overall energy dissipation of the whole software system.

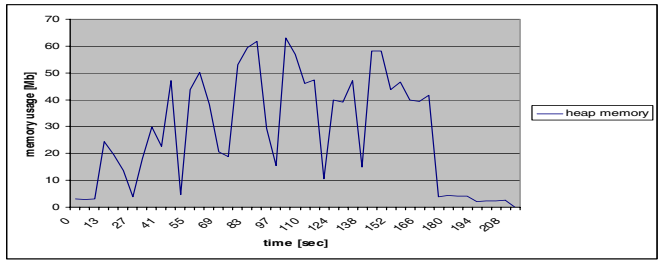
In Java, JVM manages two kinds of memory areas: heap and non-heap memory. Both of these memory pools are reserved from the OS when JVM is started. The heap memory is the runtime data area, from which the memory for all class instances and arrays is allocated. The heap can be either of a fixed or variable size. An automatic memory management system reclaims unused objects from the heap memory. Usually, the heap is the largest memory area inside the JVM as can also be seen in Figure 4. The non-heap memory is the storage area for the compiled source code and memory required for the internal processing or optimization for the JVM. Like the heap, the non-heap area may be of a fixed or variable size. Depending on the implementation of the JVM, non-heap memory may be subjected to a automatic memory management system. [38]

The use of non-heap memory can be seen as a static memory consumption of an application. Static items such as constant pool entries, bytecodes of methods, interface and inheritance information and method information are invariable with respect to the application's runtime behavior. Rather, the non-heap memory consumption represents the static size of the distributable application. Usually mobile devices have a fairly limited size for storing applications. Thus, the storing of more applications or applications with more functionality and richer resources is enabled by a reduction of the static memory consumption [39].

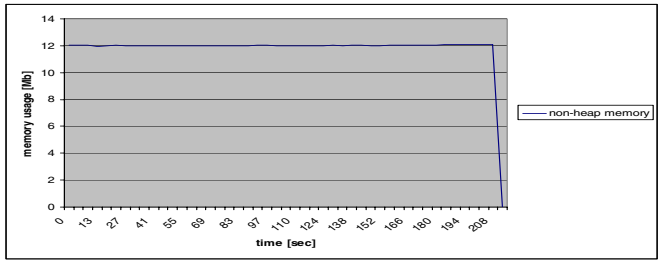
The use of heap memory can be seen as a dynamic memory consumption of an application. The heap contains objects and arrays that are created during application execution and these allocations can be released by automatic memory management. Therefore, an application's heap memory allocations are constantly varying with respect to time. Considering that building some objects requires lots of work and reserving lots of heap might lead to increasing the activity of automatic memory management, use of a dynamic memory directly reflects on other performance attributes. Thus, it is essential to understand the heap memory footprint of an application.

Th memory consumption is an attribute that is measured at some point of time. To understand the application's behaviour regarding memory consumption, there has to be consecutive measurements taken during an interested activity period of an application. A typical interpretation for the Java application's memory con-

sumption data is illustrated in Figure 4, which presents the memory consumption of a simple board game. This clearly shows how the application uses heap and non-heap memory pools during its active period. An application's heap memory allocations are constantly varying with respect to time, so it is important that we can measure it at any point of time in order to understand the memory consumption behavior of the application.



(a) Heap memory consumption



(b) Non-heap memory consumption

Figure 4. Memory consumption of a Java application.

We chose to concentrate on observing the heap memory. This is the more interesting part of the two memory spaces as it holds every object that is dynamically allocated, and also because the memory requirement of a Java application is shaped mainly by the heap space required for executing the application [32, 40]. The use of dynamic memory is directly linked with the runtime behaviour of an application, so observing just the heap memory serves our goals.

The non-heap memory is not in our interest. As shown in Figure 4b, non-heap memory is not subject to constant variation with time and the space allocated from non-heap memory space by applications can be calculated with all of the

currently existing monitoring tools. In the OSGi environment, calculations can be difficult, since we have to use comparison and statistics. For example, we can measure non-heap allocations before a certain application has been started and after it has started and compare the results. This comparison can be repeated and calculate the statistical size of reserved non-heap memory. These calculations are time consuming and a little imprecise, nevertheless these still can be made.

Network utilization Network utilization provides a measure of the size and amount of packets sent and received by an application in a networked environment. The purpose of this attribute is to get a concrete view of the network activity caused by an application during a specified interval. In the case of a congestion in the networking media, we can get an inside view of the networking behaviour of bundles.

2.3 Monitoring Java Virtual Machine

Runtime monitoring tools observe and provide statistical information on the execution of the software system or applications. Monitoring tools are used to identify root-cause problems in applications and to get an inside view into the run-time behaviour of an application. Monitors can be either software or hardware based. Software monitors are programs that operate independently from the software to be measured and hardware monitors are external devices that are attached to the system to be monitored through external wires or probes [41]. This thesis work concentrates on software monitors.

Monitoring tools are typically highly platform dependent, as useful runtime information is available only at the OS level. Although the high abstraction level of Java introduces monitoring challenges, the use of a virtual machine helps to make monitoring tools more portable. J2SE provides built-in interfaces for managing and monitoring the JVM. This chapter introduces different measurement techniques and represents the basics of the provided interfaces, which have a different approach for collecting information.

2.3.1 JVM Measurement Techniques

Monitoring tools typically use three different measurement techniques; source code instrumentation, event- and time-based monitoring. These are not exclusionary; all of these can be exploited in a monitoring tool. As was stated in [42], most useful results can be achieved by combining different techniques. When considering the JVM, the techniques remain the same but, as Java bytecodes are platform independent it is more portable than normal source or low-level machine code instrumentation. The measurement can be internal or it can be executed external to application. While external measuring does not need additions to the code of the measured application, internal measuring requires codes within the application to detect events and convenient performance data. All the measurement techniques enable a range in the level of detail. One requirement for the detail level needed is derived from our interest in measuring a single bundle and not the whole software system. Choosing the measuring technique is always some sort of trade-off between the following things; the detail level of data, amount of collected data and disturbance caused to the measured system. Clearly, these are not exclusionary but rather interdependent. To ensure that the measured data is meaningful and its level of detail is desired, we must be able to control the measuring process. The following primary aspects were highlighted by Smith in [41], and these must be taken into consideration to get reliable and meaningful measuring results:

- The granularity of events must match the resolution of the system clock used to time them. The events should not be too short compared to the time units of the clock. For example, if some method executes in 20 milliseconds it is obvious that it does not make sense to measure it with a clock that only measures to milliseconds.
- The sampling interval of the monitor must be adapted correctly. The monitor will not detect all occurrences of the interested matters if the interval is too long. On the other hand, interval that is too short causes explosion of the collected data size as well as overhead injected to the system by the measurement.

The following gives a more detailed introduction to the JVM monitoring techniques.

Bytecode instrumentation (BCI) The most common approach for collecting desired runtime information is modifying the source code of the application. These additions to the application code can be inserted at compile time, runtime or programmer can do insertions directly to the source code. In this approach, a developer replaces or adds bytecodes to desired locations in the original code. At the runtime, these additions are responsible for logging actions or raising specified events. Instrumentation can reduce disturbances caused by monitoring, as only the desired regions of code can be instrumented, rather than instrument the whole application.

Event-based monitoring An event is a happening of interest which occurs instantaneously at a specific time [43]. In event-based monitoring, we define the events of interest, and record their occurrence time stamp and appropriate performance data. In Java, a great benefit is that the JVM is internally instrumented to provide previously defined events such as a thread start or stop. If these events are sufficient, this relieves the programmer from instrumenting the underlying platform or operating system by himself.

Time-based monitoring Time-based monitoring tools are also called sampling monitors, and these activate at predetermined time intervals and record the current state and the appropriate performance data. The simplest form of sampling typically causes less overhead than instrumentation because it does not need any additions to the executing software. A sampler simply copies information, such as a function call stack or execution location, to memory. If a more complex form of sampling is used, the software system may need to be interrupted to record the needed information.

Time-based techniques can be considered as statistical monitoring because; to obtain a consistent view of an application behaviour a significant number of runs must be performed. This number varies from application to application because

of the different kind of behaviour of the application. In addition, the sampling period may need to be adjusted for a particular application. Although sampling is less intrusive than event-based techniques, it does not provide as precise data as those do.

2.3.2 Java Management Extensions

The Java Management Extensions (JMX) define an architecture, the design patterns and the Application programming interfaces (API) for management and monitoring applications, devices, services and the JVM. It provides the means for changing application configuration, gathering behavioural statistics of an application and error and state change notification. It also provides remote access, so these resources can be monitored and managed remotely. The JMX has been part of the core platform since the version 5.0 of J2SE. [44]

The JMX specification provides a framework that can be separated into three layers: the instrumentation level, the agent level and distributed services level. The key components of the JMX are manageable resources, dynamically extensible agents and distributed management services. In the JMX technology, the abstraction of manageable resource is called Managed Bean (MBean). Figure 5 shows the key components of the JMX architecture and their relations within the three levels of the JMX architecture.

The instrumentation level provides a specification for implementing the JMX manageable resources. A JMX manageable resource must comply with the MBean standard defined in the JMX specification and may be dynamically added to or removed from the JMX agent. MBeans encapsulate manageable objects as attributes and operations through their public methods and utilize design patterns to expose them to management applications. There are four types of MBeans: standard, dynamic, open and model MBeans. Each of these address to a different instrumentation need [44]:

- Standard MBean provides a static management interface which defines methods for reading and writing attributes and for invoking operations.

- Dynamic MBeans conform to a specific interface that exposes the management interface at runtime.
- Open MBeans is a dynamic MBean that relies on a small, predefined set of universal Java Types to describe managed objects and they advertise their functionality.
- Model MBeans are dynamic MBeans that are configurable and self-described at runtime. These can be used in instrument dynamically almost any resources.

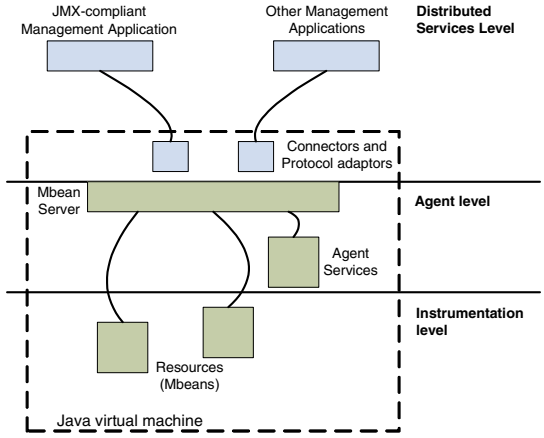


Figure 5. The JMX architecture.

The agent level provides a specification for implementing the JMX agents that control MBean resources and make them available to management applications. A JMX agent consists of an MBean server, a set of agent services and additionally requires at least one communication connector or adaptor. An MBean server is a registry for MBeans in the agent and it handles all the management and monitoring requests and dispatches them to the appropriate MBean. JMX agent services are implemented as MBeans and these provide services for the management and monitoring applications or for other MBeans. Protocol adaptors and connectors provide remote access to the agent for management applications. All these components of the agent level are mandatory in an implementation of the JMX specification. [44]

The distributed services level defines the interfaces and components for implementing the JMX managers. These interfaces and components allow remote management applications to perform operations on agents and expose a management view of an agent and its MBeans.

JMX provides out-of-the-box means for the management and monitoring of JVM. The JVM is highly instrumented using JMX technology and the JMX agent provides access to the built-in JVM instrumentation, and thereby it is possible to monitor and manage the JVM remotely. By using JMX technology, we can collect not only JVM level but also thread level resource consumption information. One main advantage of the JMX is that it allows remote monitoring of the system, and thereby, the monitoring application does not consume the resources of the monitored system as much as with the local monitoring. Another benefit of the JMX is that it does not need to add anything to the application code. However, it lacks the means to trace the thread's heap allocations, network utilization and file system usage. These are especially emphasized in a OSGi environment, where all the applications run in the same virtual machine. The disadvantages are largely derived from the fact that the JMX is not designed just for profiling purposes, but rather for dynamic extension mechanism and for enabling distributed management system.

2.3.3 Java Virtual Machine Tool Interface

JVM TI is a programming interface used by debugging and monitoring tools. It provides both a way to inspect the state and to control the execution of applications running in the JVM [45]. This interface was brought to the Java specification in version 5.0 and it replaced the Java Virtual Machine Profiler Interface (JVMPI) that was included as an experimental feature since version 1.1. JVM TI is described as a part of the Java Specification Request (JSR) 163. It is not guaranteed that the JVM TI is available in all the implementations of JVM but major vendors, like Sun and IBM, have included it in their JVM implementations.

JVM TI is a native interface and all clients of this interface must be written in language that supports C-like calling conventions. The clients of the JVM TI are called agents, and therefore, all clients of the JVM TI will be referred toas agents from now on. The JVM starts the agent early in the initialization phase, before any

bytecodes have been executed or classes have been loaded. An agent is attached to the same process as the monitored JVM, thus, minimizing the communication costs with the JVM. The communication model of JVM TI is presented in Figure 6. Communication with the JVM happens through direct calls to functions provided by the JVM TI, events raised by the JVM and BCI.

The JVM TI specification supports more than just one agent running simultaneously but each agent has its own JVM TI environment. This restricts the changes made in one environment having an affect on the state of others. The state of the environment includes the following properties; *capabilities*, *event notifications* and the *event callbacks*. These properties are set by the agent at its load up phase to achieve the needed functionality. With *capabilities* the agent defines the functionality available in this JVM TI environment. Because *capabilities* may incur a cost in processing speed or memory space, initially all *capabilities* are disabled. Enabled *event notifications* define the desired JVM events to be listened. The *event callback*-function specifies callback function for the occurrence of corresponding event.

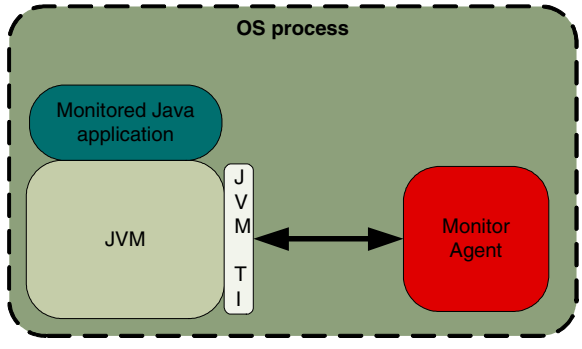


Figure 6. JVM TI agents' communication model.

The JVM TI provides support for BCI, the ability to alter the Java virtual machine bytecode instructions, which form the target program. Added bytecodes are usually an additive method call to capture some specific data and these additions do not modify the application state or behaviour. BCI is actually necessary because a basic assumption of the JVM TI is that information that can be collected with the BCI, should be collected with that technique. As was stated, JVM TI provides the required support for instrumentation but it does not provide the means

for the actual addition of bytecodes. This has to be performed by writing a necessary code or using external BCI libraries. The following list presents supported instrumentation means as defined in [45]:

- Static Instrumentation: The class file is instrumented before it is loaded into the VM - for example, by creating a duplicate directory of *.class files which have been modified to add the instrumentation. This method is extremely impractical, and in general, an agent cannot know the origin of the class files, which will be loaded.
- Load-Time Instrumentation: when a class file is loaded by the VM, the raw bytes of the class file are sent for instrumentation to the agent. The *ClassFileLoadHook*-event, triggered by the class load, provides this functionality. This mechanism provides an efficient and complete access to one-time instrumentation.
- Dynamic Instrumentation: A class that is already loaded is modified. This optional feature is provided by the *ClassFileLoadHook*-event and triggered by calling the *RetransformClasses*-function. Classes can be modified multiple times and can be returned to their original state. The mechanism allows instrumentation, which changes during the course of execution.

As the JVM TI is designed for developing monitoring and debugging tools, it provides the means to collect very detailed information from the JVM. Because the JVM TI is capability-based the performance impact of the JVM TI agent can be reduced by only selecting the desired *capabilities* unlike in the JVMPI which was more of an "all or nothing"-approach. The BCI approach provides complete control to the agent: the instrumentation can be restricted to desired portions of the code and can be conditional. The BCI instrumentation adds standard Java bytecodes, so the virtual machine is able to also optimize the instrumented code and the overhead, added by the measurement agent, is minimized. However, if the instrumentation involves switching from bytecode execution, expensive state transitions are required and additional bytecodes can exhaust the JVM. [45]

2.3.4 Existing monitoring approaches

There is wide variety of tools available for monitoring JVM, both commercially and non-profit. It is impossible to make a comprehensive list of the currently available tools. Therefore, the tools mentioned here should be taken as representative entities. Typically, monitoring and profiling tools are used by developers to track functional problems but we are interested in the resource consumption information of an application. Therefore, tools, which create function call trees and calculate method execution times have been excluded from the review.

Java 2 SE 6 Java Development Kit (JDK) tools and utilities The JConsole is a JMX-compliant monitoring tool and it is delivered with the Java 2 SE 6 JDK. It provides the performance and resource consumption information of applications running on the JVM by using JMX instrumentation. The JConsole can be connected to the JVM either locally or remotely. It obtains its information from JVM MBeans in the connected JMX agent. The JConsole provides information on memory usage, thread usage, class loading and garbage collection statistics of the monitored JVM. The JConsole also provides a generic way to manage and monitor applications if these implement the MBean interfaces. [8]

Jstat is an experimental tool that monitors the JVM performance and resource consumption. The Jstat is attached to an instrumented HotSpot Java virtual machine to collect and log performance statistics as specified by the command line options. The Jstat collects information on memory usage, class loading and garbage collection statistics of the monitored JVM. The instrumentation is designed such that it does not require a separate starting, yet has negligible performance impact. This monitoring interface, added to the HotSpot JVM, is proprietary and it is not guaranteed that it will be supported in the future.

The J2SE provides a simple command line profiling tool called HPROF. The HPROF collects information on heap memory usage, CPU utilization and it can dump the whole heap to a file. These heap dump files can be analyzed with a tool called the Jhat. These tools are meant to be used as an aid for the developer to track bugs and memory leaks. The HPROF itself is not targeted to be a capable analyzing tool, but rather to provide a foundation for building a performance analyzer.

The NetBeans Profiler The NetBeans Profiler provides profiling functionality for the NetBeans integrated development environment (IDE). It relies on JFluid technology, developed by Sun Microsystems, and bytecode instrumentation. The JFluid is a technology that provides a mechanism to dynamically start or stop profiling and select only the desired portion of code to be profiled. The NetBeans Profiler can monitor thread state, CPU utilization and memory usage. CPU utilization can be seen on both thread level and method level. The NetBeans profiler is a solution for debugging purposes, for finding memory leaks or unoptimized code and is tightly integrated into the IDE work flow. [9]

Academic approaches Whaley have presented a sampling-based online measurement system for Java in [46]. It uses periodic thread sampling to collect every thread's stack, program counter and CPU time. This information is used to create partial calling context tree (PCCT), a data structure for efficiently encoding approximate context-sensitive profile information. The system provides an effective way for detecting not only the most CPU time consuming code blocks but also the call context of its occurrence. This information is not however appropriate for our means, because even if we can identify the calling application from the call stack it is inefficient. Also all the other resources that we are interested in are ignored.

There have been various proposals to realize a resource control in Java, although these are not just monitoring systems, these provide interesting monitoring and accounting solutions as a basis for their resource control system. Czajkowski and von Eicken proposed the JRes [47], which is a resource control system. It uses per-thread accounting for tracking the consumption of CPU time, heap memory and network resources. Resource usage information is collected with a mixture of bytecode instrumentation and native code. This approach lacks the linkage between the threads and application as all the threads are treated and accounted individually.

Binder et al. have long studied the fully portable profiling approach both separately [48, 49] and as a part of the resource accounting framework [30, 50, 51]. These rely on bytecode instrumentation in order to account the used resources. The profiling approach uses self-accounting for approximating CPU usage, which

means that each thread counts its own executed bytecodes and reports these to the account shared between all the threads of a component. These approaches not only use a fully portable Java code but also use platform independent profiling metrics, such as bytecodes. Introduced CPU accounting for all the threads of the component provide a linkage between the threads and application, and therefore, it presents the CPU usage of the whole component. However, the mapping of the threads to the component is not discussed in detail.

2.3.5 Existing visualization approaches

The resource consumption visualization of software has not been a subject of extensive research. There have been a large number of different systems that provide visualizations of the dynamics of a Java program, for example [52, 53, 54]. These concentrate on the program flow, behaviour and interactions of threads and class allocations. Typically, existing resource consumption visualization approaches use simple 2D-graphs for representing the usage of one single resource, as shown in Figures 3 and 4. This is an inefficient way of interpreting resource consumption behaviour because a human analyst has to inspect all the graphs separately. It is time consuming and the correlations between different graphs are often difficult to notice.

3. Monitoring and visualization approach

This Chapter presents the goals and requirements of our monitoring approach. In addition, challenges and possible difficulties are identified. The monitoring system consists of two main processes, the monitoring and visualization processes as shown in Figure 7.

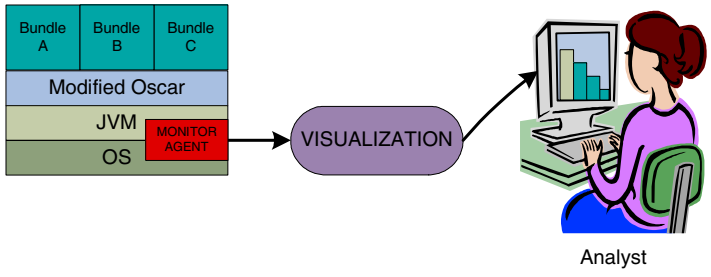


Figure 7. High level perspective to the monitoring approach.

The high level Figure 7 presents the key components of the monitoring and data post-processing approach. The monitoring tool collects runtime behavioural information of bundles and the visualization tool interprets this information to the user of the tools. A log file and analyzable graph are the end products of these tools.

Considering Figure 7, both processes of the monitoring system are equally important and useless without the other. As the monitoring process produces a textual log file, without proper interpretation it is impossible for the analyst to gain a clear behavioural view of the whole lifetime of the bundle. Even the most state-of-the-art visualization is also useless without exact measurements. At the first phase, monitoring collects information to a log file and all the calculations are performed after desired test cases have been executed. This reduces the interference caused by the monitoring system than if the visualization had been performed online. In this case, the visualization should be distributed to another machine for minimizing the overhead caused by the time-consuming calculations.

We have largely derived the requirements for the monitoring and visualization approach from the target environment and the requirements of target end-users. As

the tools are targeted to be used not only by application developers but also by external analysts, tools should be usable even without deeper knowledge of the monitored applications or the underlying hardware. As the monitoring system consists of two different processes, both have individual requirements. The following sections discuss the basic requirements and implementation issues of the monitoring system.

3.1 Monitoring process

The monitoring process aims at revealing the inner resource consumption behaviour of the JVM. This can be achieved by identifying the actual sources of the resource consumption inside the JVM and collecting these statistics. Chapters 2.2.3 and 2.3 provide a base for the following fundamental features of the monitoring process that we have identified.

Monitoring process requirements

1. Most of the collected resource consumption information is bundle-level and not JVM-level information. This feature creates a requirement for a detailed level of data that we must be able to extract from the underlying system.
2. Multiple bundles can be monitored at the same time.
3. The first prototype of the measuring tool collects the following resource consumption attributes:

   (a) Heap memory consumption of a bundle.
   (b) CPU utilization of a bundle.
   (c) CPU utilization of the whole JVM.

   The following attributes are optional:

   (d) Network utilization of a bundle.
   (e) Mass storage consumption of a bundle.

4. Bundles that are monitored can be selected at the startup phase of the tool.
5. Monitoring does not require any changes to the source code of the monitored bundle.
6. Overhead caused by the monitoring process is measured or must be at least assessable.

At first, the monitoring tool is implemented to the Windows platform, but there should not be any problems in porting the implementation to other platforms as well.

The most crucial requirement of the monitoring process is the identification of the actual source of resource consumption. As the OSGi environment is composed of multiple collaborating bundles that execute in a single JVM, it will be a challenging task to identify which is consuming computing resources. This raises the first question, how can we identify a resource consuming bundle? In addition, the OSGi environment introduces another challenge for identification: service providers. Bundles that provide services for other bundles provide a public interface for other bundles to use. This leads to the second problematic question, should we account resources consumed during service interaction to the service provider or the user of the service?

Bundle identification As was studied in Chapter 2.3.4, thread based tracking enables the bundle level identification of resource consumption. This gives us the answer to the first question. We still need to solve the mapping threads to the bundle. Figure 8 presents a high-level algorithm for tracking the bundle's resource consumption.

We considered different choices for bundle identification such as tracking bundles separately for every resource consumption event, but concluded that the least disruptive way for identification is to identify new threads at their time of creation. As shown in Figure 8, a newly created thread is identified and if it belongs to a bundle that is in our interest, we add this thread to the set of monitored threads of the bundle. Otherwise, the resources consumed by this thread are ignored. As we

EVENT: Thread_Start
if *the thread belongs to a monitored bundle* **then**
| add the thread to monitored threads;
else
| the thread is ignored;
end

Figure 8. Bundle identification, right after a new thread is spawned.

have identified all the threads of the bundle and tracked the resources consumed by these threads, it is possible to account the consumed resources to exactly one bundle.

Accountancy of consumed resources To answer the second question about resource consumption accounting in case of service interaction, we need to consider the accountability of the consumed resources. As services are used through public interfaces, it can be seen as a direct sharing of objects that provide some service. Two possibilities can be defined to account the consumed resources during service interaction: direct and indirect. The following discusses these choices and their impact on resource consumption monitoring.

The indirect accounting of resources, consumed by a bundle, means that all the resources consumed by threads belonging to one bundle are accounted to this same bundle. Thus, the bundle is held accountable for all actions made on behalf of the bundle. By using this approach, the resources used during service interaction will be accounted to the bundle using the service. This is because the service user directly calls methods offered by the service provider. This leads to the fact that during service interaction, the service provider is not even active in a sense of resource consumption. This approach is not completely misleading since resources consumption is accounted to the bundle that actually requires these resources.

The direct accounting of resources that are consumed by a bundle means that the resources consumed by computing inside one bundle will be accounted to this corresponding bundle. Thus, the bundle is held accountable for all actions made by it. Using this approach, the resources used during the service interaction will be accounted to service provider. So, when a user of the service calls the service

provider, the accounting of the consumed resources changes from the service user to the service provider though a thread that performs the service interaction belongs to the service user. This approach isolates the service providing component from the service user and both components will be accounted with the appropriate amount of resources consumed.

Let us consider a situation where a service providing bundle is poorly implemented and uses excessive amounts of computing resources. Using the indirect accounting approach, it seems that the service using bundle is consuming all those resources and the consumption of service providing component is concealed. With direct accounting approach, the monitoring reveals that the service provider consumes all those resources. Thus, direct accounting provides a distinction for the resource usage of a component and the resource usage that a component cannot affect directly.

We have chosen the direct accounting approach because it makes a clear distinction between the resources used by the bundle and other bundles. To achieve this, we must be able to provide a mechanism to isolate applications from each other, such as in KaffeOS [55], which uses process abstraction for every application like the traditional OS. The isolation of different components in the OSGi environment must be achieved without interfering with the system's normal functioning because in this case we are only interested in monitoring and not in managing the resource consumption of components. In addition, the monitoring should be possible without changing the monitored component. This sets a requirement for isolation: it must be invisible for every collaborating component.

Although with thread-based identification, we are able to account resources to a corresponding bundle, it is not a complete solution. One disadvantage is that it is impossible to account the resources that are consumed by the JVM on behalf of a particular bundle: for example, CPU time spent while performing garbage collection to the objects allocated by a bundle from the heap memory.

3.2 Visualization process

As the log file, produced in the monitoring process, can be rather cryptic to examine, the visualization process is needed to transform the collected resource consumption data in some form that a human can easily analyze. Informative and clarity are the keywords in achieving analyzable representation. From these keywords and the studies of earlier chapters, especially Chapter 2.3.5, we have derived the main requirements of the visualization process.

Visualization process requirements

1. Outcome is easily analyzable scenery where all the monitored bundles and their measured resource consumption attributes will be seen from a single scenery.
2. CPU time consumed by a bundle can be compared to both, elapsed time on the sampling period and to the CPU time consumed by the JVM as a process.
3. Besides the visual scenery, numerical information is also produced to support analyzing. This is produced from the desired time interval of the log file. At the first phase, the textual information consists of:
   (a) Selected time frame in wall clock time.
   (b) Averaged CPU time consumed by a bundle compared to both, selected time interval and to the CPU time consumed by the JVM in this interval.
   (c) Heap memory consumption statistics from the selected time interval. This includes the bundle's largest, smallest and average consumption.

Existing visualization and analysis tool The visualization and analysis tool is responsible for interpreting the logged measurement data in a human analyzable form. The tool produces 3D-visualization and analysis, presenting the resource

consumption behaviour of monitored OSGi bundles. In the first phase, the monitoring collects information to a log file from which visualization tool reads the data and produces an off-line visualization.

The visualization and analysis tool should provide an inside view to the resource consumption of the JVM during the monitored time frame. The time frame is not limited and it should be possible to efficiently visualize both short and long time frames. These can range from milliseconds to tens of minutes. To enable analysis, the tool should produce not only visual but also numerical results of the desired time frame. These numerical results will be displayed in visualization and these are stored in text files for later analysis.

The tool that will be used for analysis and visualization has been designed and implemented by Yrjönen [56]. The tool was initially designed for and used in analyzing performance scaling techniques and the system wide execution and performance of embedded devices. It produces 3D-scenery from runtime traces, which allows a free movement; the analyst can view the model from any angle and at any scale. This tool will be extended in this thesis work to be capable of visualizing and to aid the resource consumption analysis of OSGi bundles.

Visualization models Presenting software runtime behaviour with 3D-graphics is not revolutionary, since it is easy to transform a graphical model from 2D to 3D. The main point is how the third dimension is used in visualization. In the first phase, we concentrate on memory consumption and CPU utilization. In visualizing resource consumption, we combine CPU utilization with the memory consumption of a OSGi bundle. This effectively illustrates the resources used by a component. In addition, it gives a clarifying view to the inner behaviour of the JVM.

Figure 9 illustrates how each time slice is composed in a 3D graph. This figure presents the statistical CPU load of each bundle and the whole JVM in a certain time interval. Three bundles have been monitored in this example. These are sorted by their memory consumption, the lowest block in the graph consumes the largest amount of memory, in this case bundle A. Their relative CPU utilization can be seen on the vertical axis, the largest share is consumed by bundle B in this example.

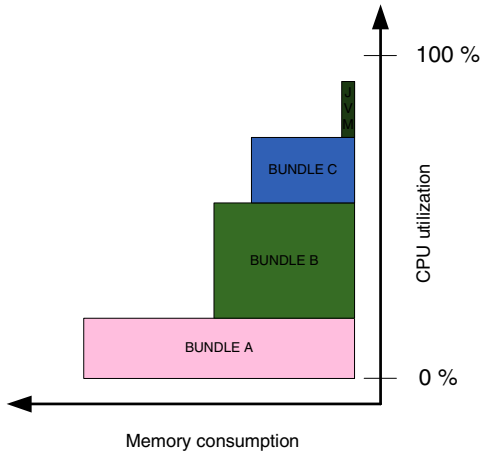


Figure 9. CPU utilization of bundles sorted by memory consumption in a single time slice.

The illustration shown in Figure 9 gives an insight view to the processor activities during a specified time interval and it also opens up the JVM as a process for analyzing. The total vertical size of the coloured area illustrates the CPU load in the processor caused by the JVM. In this example, it has reserved the processor for almost the whole time interval. The uppermost block illustrates the CPU time of the JVM that has been reserved outside of the monitored bundles. Below the JVM block, we can see how the CPU time taken by the JVM has been divided between the monitored bundles.

When a series of illustrations, shown in Figure 9, is created over time and arranged into a 3D-graph one after another, we obtain the illustration shown in Figure 10. This presents the changes in the CPU utilization and memory consumption of bundles over time as processing continues. In Figure 10 the time is represented along the x-axis, resources are in the same way as in Figure 9, CPU utilization along the y-axis and memory consumption along the z-axis.

In Figure 10, there are two OSGi bundles monitored, A and B. C represents the CPU utilization of the JVM. This gives us a revealing view of the CPU load distribution inside the JVM. This view enables the comparison of the CPU load caused by bundles either to the time interval or to the JVM itself. As the memory con-

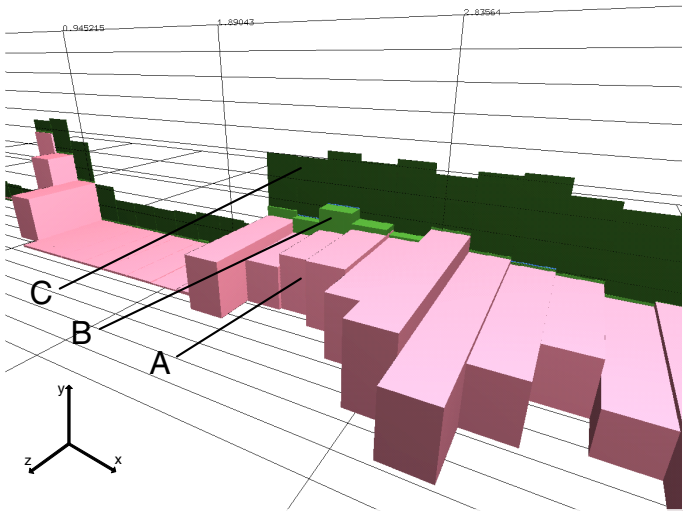
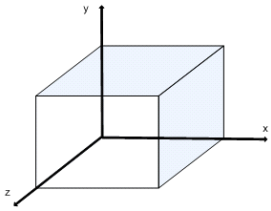


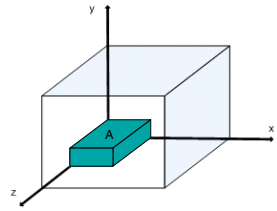
Figure 10. 3D-graph presenting resource consumption of applications.

sumption of bundles is represented in the same graph, the resource utilization of a software component is effectively illustrated. This helps the analyst to gain a clear view of the executed bundles inside the JVM by analyzing only one single graph.

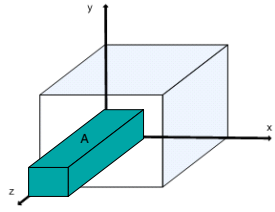
As the final goal is to be able to monitor the usage of more than two resources and then evaluate these against predefined resource boundaries, we must construct another effective illustration. Let us consider that we have three resource attributes, for example, CPU utilization, memory consumption and networking activity. Figure 11a illustrates the predefined boundaries for these resources. The boundaries form a cage that defines the maximum usage of different resources that are allowed for a component. In this illustration, the CPU load is represented along the y-axis, memory usage along the z-axis and networking activity along the x-axis. The time is not represented at all. When a component is executed, we can form the same kind of block as the boundary cage from the resource usage values of the component. The monitored software component can now be inserted inside this cage and see if the requirements set by these boundaries are met. Figure 11b illustrates a software component that meets the resource usage requirements. Figure 11c represents a case where the application violates its resource requirements.



(a) Boundaries of three resource attributes.



(b) Bundle meets the specification.



(c) Bundle violates its resource usage boundaries.

Figure 11. Evaluating bundle behaviour against pre-defined resource consumption boundaries with 3D visualization.

The illustration shown in Figure 11 aids the analysis of an application's resource consumption behaviour. It enables a simple and effortless evaluation of the performance of a software component. Like in Figure 11c, although the boundaries set for the CPU utilization and networking activity are met, the memory consumption limit is exceeded. This would lead to a re-evaluation of the implementation or design. This type of 3D representation that captures all the attributes on a single graph is far more effective and clearer than three 2D graphs, as discussed in Chapter 2.3.5.

3.3 Support for dynamic adaptation

In the previous sections, we have discussed the static usage of the monitoring tool that does not affect the execution of the software excluding the overhead introduced by the tool. This section discusses how the monitoring tool can enable decision making that affects the runtime behaviour of the software components.

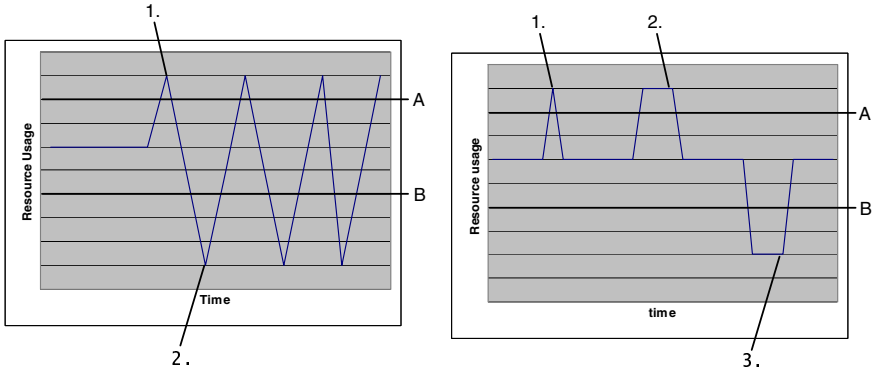
An important aspect when developing the OSGi-based platform independent software is the availability of computing resources. This is emphasized in resource-constrained computing environments unlike with the desktop computing environments. The behavioural adaptation of software itself means that a program can adapt its internal behaviour to meet the dynamically varying availability of computing resources. Two adaptation schemes can be distinguished: self-adaptive and platform controlled. The self-adaptive software gains the resource availability information itself and the platform controlled receives an adaptation request from the underlying platform.

Regardless whether we are considering either a self-adapting or platform controlled adaptive software, the adaptation needs support. The application or the platform must be able to access resource usage information, in order to make adaptation decisions. This thesis work presents an OSGi-based service component that provides the dynamic resource usage information of different components running in the environment. In addition, information on the availability of different computing resources in the whole environment will be provided. Support for dynamic adaptation creates a need for a runtime access to the monitoring tool. Therefore, we must develop an OSGi bundle that communicates with the monitoring tool and provides a service that supplies the resource consumption information of the components and the JVM. This type of approach separates the monitoring and minimizes communication between the bundles and the monitoring tool. This bundle only makes resource usage information available, it does not generate any adaptation requests neither does it take any part in the adaptation process itself.

Bundles that are capable of behavioural adaptation will be registered to the monitoring service. The registration provides also their resource usage boundaries. The monitoring service forwards the registration to the monitoring tool, so it can add these bundles to the set of monitored bundles. During execution, the monitoring service notifies every time when a bundle has crossed its boundary. After that, it is up to platform or the bundle itself to perform the necessary adaptation request or actions.

False notifications from a monitoring service can waste the advantages introduced by dynamic adaptation and ruin the performance of the software system. Sudden

but short-term changes in resource usage can lead to premature corrective actions. Premature adaptation actions cause resource usage oscillation though the goal is to achieve a steady resource usage level. To avoid false notifications, the monitoring service must have the means to correct sudden changes in resource usage levels without notification. For example, averaging is a simple and efficient way to avoid false notifications. Figure 12 clarifies the application's behaviour with premature and correct adaptation.



(a) Premature adaptation. *(b) Correct adaptation.*

Figure 12. Application's dynamic adaptation to available resources.

Figure 12a illustrates the application's resource consumption behaviour when adaptation requests are premature. A and B represent the resource usage boundaries, the target of the adaptation is to keep the resource usage level between these boundaries. At first, the resource usage is at the target level and then at the point 1 there occurs a peak in the resource consumption. Since it is only a peak, the resource usage would return near the earlier level without any actions. In this case, there has been a notification of resource boundary violation sent and the application performs adaptation actions to lower its resource usage. These premature activities lead to a large drop in resource usage and it drops below the low-level boundary. Again, at point 2, notification has been sent that there are a great deal of available resources to use. These premature actions can lead to an unstable behaviour as can be seen after point 2. These false notifications are sent due to fact that, without any correcting means the monitoring end can not determine whether the changes are permanent, short-term peaks or incorrect measurements.

The illustration in Figure 12b shows the application's behaviour with correct adaptation requests. A and B present the same boundaries as in Figure 12a. In this case, averaging has been used at the monitoring end. Now, when a peak occurs at the point 1, notification is not sent because the averaged resource usage stays below the level A. At points 2 and 3, notifications of changes in resource usage levels are sent and the application makes correcting activities to reach the desired level of resource consumption. These result in a desired behaviour since the notifications are now sent correctly, only when the change in resource usage level is permanent. It is clear that in this case, the application reacts slower to changes of resource usage levels but even so, this leads to better results than actions that are faster but not as accurate. There are also more sophisticated algorithms than simple averaging that can improve the reaction time.

4. Implementation

This Chapter presents the actual monitoring approach implemented in this work and also introduces a visualization tool and extensions which were made to enable data post-processing. First of all, a short overview of the implementation is presented. Then the modifications that enable the OSGi bundle identification are described, the actual monitoring tool and extensions to the visualization tool [56] are laid out. Finally, the implementation of the OSGi-based monitoring service is laid out.

4.1 Overview of the approach

In implementing the prototype, we use Oscar, the predecessor of the Apache Felix introduced in Chapter 2.2.2, as a OSGi platform. The monitoring tool was designed to collect bundle specific resource usage information. The main difficulties and requirements were identified in Chapter 3.1. We need to be able to identify the source of the resource consumption from the information we get by observing the JVM. The overhead introduced by the monitoring is also a major issue to overcome since the desired information is so detailed that its extraction can consume a great deal of processing power. These monitoring related issues were discussed in Chapter 2.3. An overview of the measuring resource consumption in the OSGi environment can be derived from Figure 7, and is presented in Figure 13.

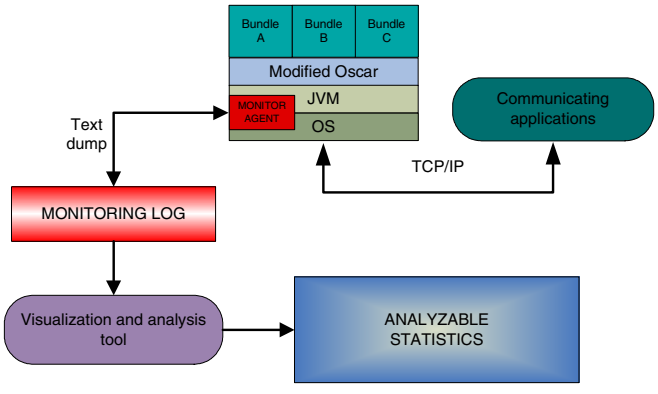


Figure 13. Overview of OSGi bundle monitoring in the OSGi environment.

The monitoring agent collects bundle-specific resource usage information from the JVM and directly from the underlying OS as all the required information, as was identified to be necessary in Chapter 3.1, is not available from the JVM. Figure 14 illustrates an example sequence of the monitoring and visualization approach.

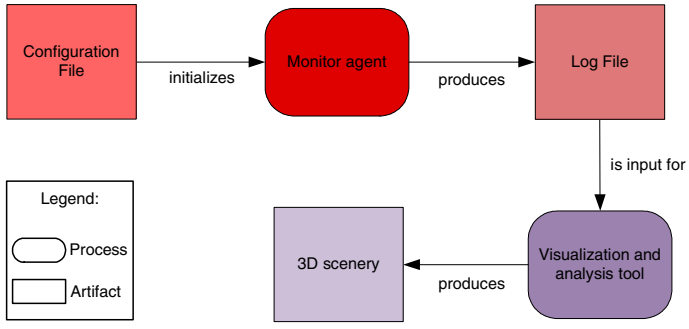


Figure 14. An example sequence of the monitoring and visualization approach.

The *Configuration file* initializes the monitor agent. The monitor agent periodically stores the measurement data to the file system. To reduce the disturbance to the executed applications, the time consuming calculations will be performed after JVM shutdown and monitoring has ended. The *Log file* contains numerical data representing the behaviour of monitored components. The *Visualization and analysis tool* processes the *log file* to provide *3D scenery*. The *3D scenery* illustrates the resource consumption behaviour of the monitored bundles. In the visualization phase, desired calculations can be performed and mathematical statistics can be gathered. The outcomes of this chain are a graphical scenery and statistics, which gives a clear behavioural view of the bundles whole lifetime. It provides the resource analysis of the bundle with as little effort as possible.

In order to achieve the goals set for the monitoring approach in Chapter 3.1, we had to modify and extend the OSGi implementation. The relations between the Oscar modifications and the monitor agent are shown in Figure 15.

The following sections discuss the implementation details of these modifications, the monitor agent and the visualization tool.

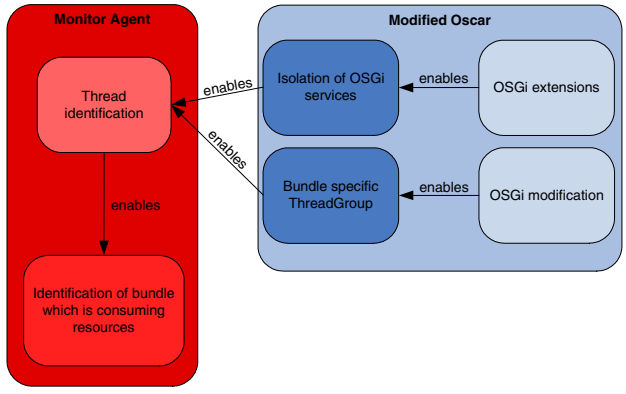


Figure 15. OSGi implementation modifications enable the resource monitoring.

4.2 OSGi implementation modifications

Modifications made to the Oscar, are required in order to identify the bundle that is consuming resources and to isolate the OSGi services as illustrated in Figure 16.

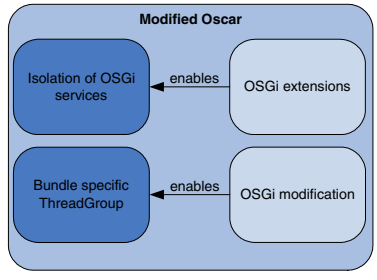


Figure 16. OSGi implementation modifications.

This modification and the extensions are discussed and laid out in the following chapters.

4.2.1 Bundle specific ThreadGroup

The identification of the actual source of resource consumption is based on identifying the thread that uses computing resources as described in Chapter 3.1. It was also designed so that the threads are identified already during the startup-phase of a new thread, thus, minimizing runtime data processing.

One way to implement thread identification is to inspect its stack trace, which is a chain of method calls that reveal where the execution is in the code. Typically, this is used to sort out where an application has crashed or hung. Stack traces are represented as strings of text. This approach was discarded because the thread's stack trace can be so long that it is very inefficient to perform a thorough inspection. A string processing overhead would be unsustainable when processing these stack traces at runtime.

We ended up implementing the actual identification from the *ThreadGroup* object of the Java thread. It is originally meant for collecting multiple threads into a single object and performing managing operations to those threads all at once rather than individually. Every Java thread is a member of a thread group. A thread group presents a set of threads and the thread group can include other thread groups. The top most groups in a Java application are named *system* and *main*. The thread groups form a tree hierarchy of threads and thread groups as shown in Figure 17.

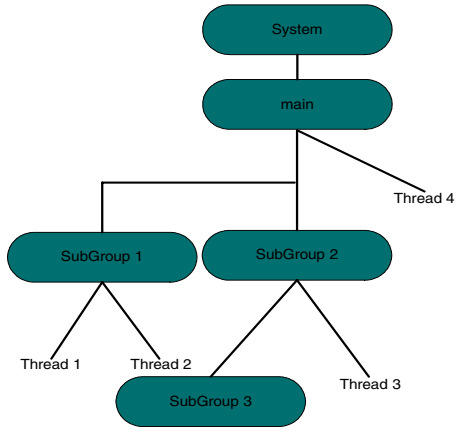
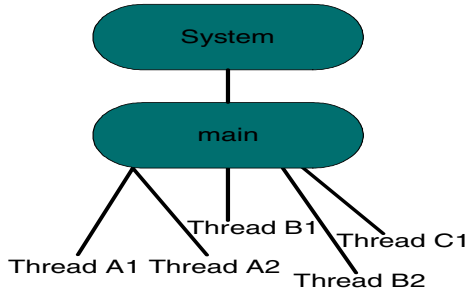


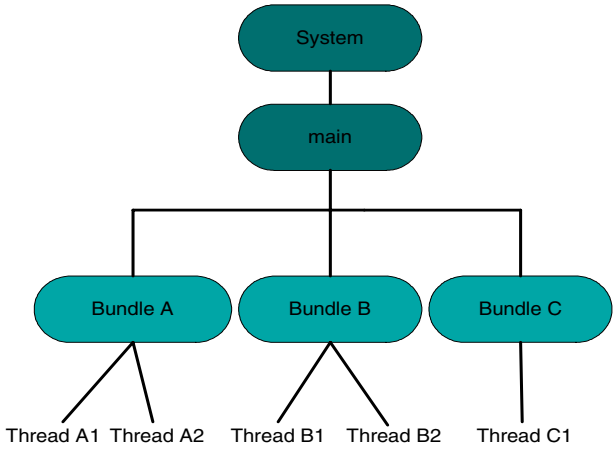
Figure 17. Thread group tree of a Java application.

The newly created thread is placed to the same thread group as the thread that has spawned this new thread unless the thread group is not separately specified. If there is a specified new thread group in the creation of a new thread, the new thread group will be placed as the child of the current thread group. If the developer does not specify new thread groups all threads will be placed in the *main*-group as *Thread 4* in Figure 17.

Figure 18 presents the Java thread group trees in the OSGi environment. The thread group tree in the unmodified OSGi environment is shown in Figure 18a. Threads A1, A2, B1 and so on, illustrate threads, which belong to different bundles. This shows that all the threads are placed in the *main*-group if the developer of the bundle has not specified otherwise. Therefore, the thread groups cannot be used in the thread identification.



(a) Thread group tree in the original OSGi environment.



(b) Thread group tree in the modified OSGi environment.

Figure 18. Java thread group trees in the OSGi environment.

In our implementation we create a unique *ThreadGroup* object for every bundle deployed to the OSGi environment. All the threads of the bundle will be placed in

this thread group. To enable this, we had to modify Oscar so that whenever there will be bundle deployed, it will be started by a thread that has a unique *ThreadGroup* object that identifies the currently starting bundle. After the bundle has started all the threads that it spawns will be in this same group as was previously explained and is illustrated in Figure 18b.

Figure 18b presents the tree of thread groups in the modified OSGi environment. All threads within the same group belong to exclusively one bundle. This enables the unique identifying of the threads and therefore the bundle that is consuming the computing resources. As was previously explained, the only modification to achieve this is that the bundle has to be started in a separate thread. To start a bundle, a thread – with a unique *ThreadGroup* object – executes the bundle's *start*-method of the *Activator*-class as was studied in Chapter 2.2.1. After this modified startup-phase, all threads can be identified to a corresponding bundle.

4.2.2 Isolation of OSGi services

Different resource accounting strategies were discussed in Chapter 3.1 and a direct approach was chosen. It was also identified that we need to be able to isolate the OSGi services in order to account the resources that these consume. To achieve complete isolation we use a proxy pattern. Figure 19 presents an overview of our approach.

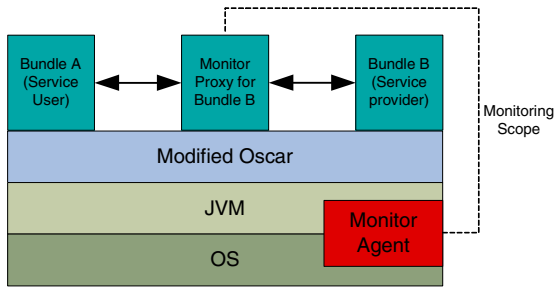


Figure 19. Proxy model for OSGi services.

In Figure 19, *Bundle A* and *Bundle B* represent service interaction participants. The *Monitor Proxy* is the intermediary in the service interaction and it is a normal

OSGi bundle that provides the same service interface as the *Bundle B* in Figure 19. The *Monitor Proxy* is the key component in enabling the monitoring of the *Bundle B* resource consumption.

Figure 20 illustrates the action flow of the service interaction with monitoring proxy:

1. Proxy captures a service call of the service user.
2. Proxy forwards the call.
3. Proxy captures the response to a service call.
4. Finally, the proxy forwards the response back to the service user.

The main responsibilities of the monitoring proxy are:

1. Perform actions that enable the resource monitoring of the service provider.
2. Perform inverse actions that enabled the resource monitoring of the service provider.

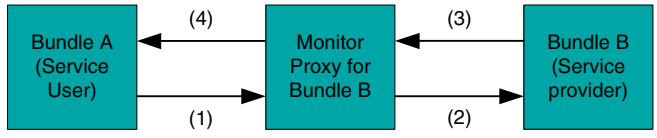


Figure 20. Service interaction sequence with monitoring proxy.

In other words, the proxy isolates the OSGi service and defines the monitoring scope for the monitoring agent. The implementation issues that enable this action flow and the extensions made to Oscar are discussed in the following paragraphs.

Oscar extensions To be able to use proxies for OSGi services, we extended the chosen OSGi implementation Oscar. Extensions were needed in order to manage the life cycle and activation of the proxies. The management of the proxy life

cycle includes the registration and unregistration of proxies. The activation of a proxy makes it possible for others to use proxies. The most important classes of the modified Oscar are shown in Figure 21. This paragraph describes shortly these modifications.

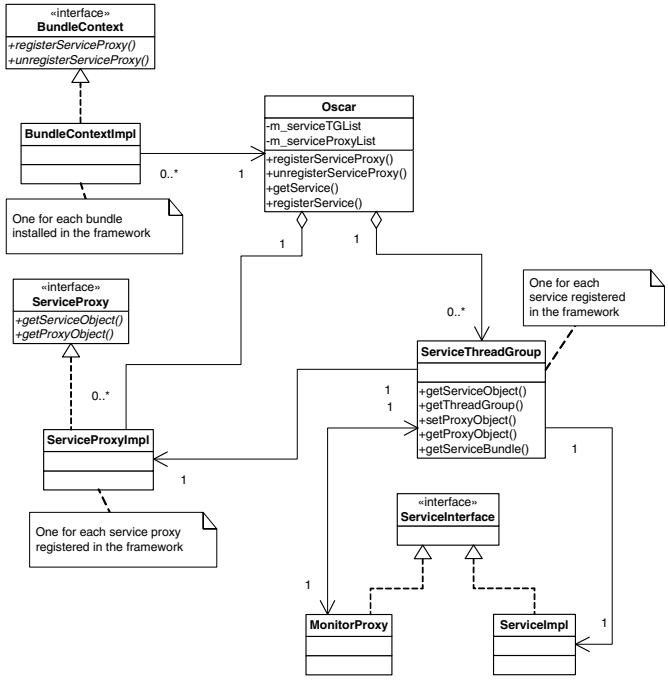


Figure 21. Class diagram of the modified Oscar.

To realize a transparent proxy, we need to be able to register the proxy to the OSGi framework and unregister it. We added two new methods to the *BundleContext* interface in the *org.osgi.framework* package: *registerServiceProxy* and *unregisterServiceProxy*. One new interface *ServiceProxy* was also added to this package. With the implementation of these methods and this interface in *org.ungoverned.oscar* package we are able to maintain the list of proxies that are present in the framework.

For every service that registers to the OSGi framework, we create an additional object that stores the information of this service. This object is an instance of the *ServiceThreadGroup* class that we have added to the *org.ungoverned.oscar* package.

It has references to the service object, service providing bundle and *ThreadGroup* object of this corresponding bundle. These *ServiceThreadGroup* objects will be placed in a vector called *m_serviceTGList* containing one object for every service registered in the framework.

The registration procedure is much like registering a normal OSGi service. A method called *registerServiceProxy* takes the proxy object, the service object and its service reference as its argument. For every proxy registered, an instance of *ServiceProxyImpl* will be created containing information on this proxy, and this instance is added to the vector of registered proxies called the *m_proxyList*. The newly created *ServiceProxyImpl* object is also linked with the corresponding *ServiceThreadGroup* object. This *ServiceThreadGroup* object will be returned after the registration procedure, to be used by the proxy that requested its registration.

The service interaction call normally returns the object that provides the corresponding service. We have modified the *getService* method so that it first goes through the list of proxies. If the list contains a registered proxy object that corresponds to the requested service the proxy object is returned to the service user. Due to this procedure, both the service provider and the service user are completely unaware of the intermediary in the service interaction. In other words, this ensures the transparency of the monitor proxy.

The unregisteration of service or monitor proxy releases corresponding objects and clears those out of the maintained vectors. After a monitor proxy has unregistered, the corresponding service operates in a normal way without the interfering intermediary. The monitor proxy itself becomes unusable if the corresponding service has been unregistered before the proxy.

Proxy implementation A monitor proxy is a bundle that implements the same interface or interfaces that the service provider has made available for other bundles to use. As was previously described, in order to act as an intermediary the proxy must register itself to the OSGi framework. The following Figure 22 illustrates the registration procedure. Firstly, the bundle that implements a monitor proxy gets the corresponding service object and registers itself as a proxy for this service. The registering method returns the *ServiceThreadGroup* object as was previously

explained. Among other information on the service provider, the returned object has the reference to the service providing bundle's thread group. No further actions are required by the bundle to operate as a proxy.

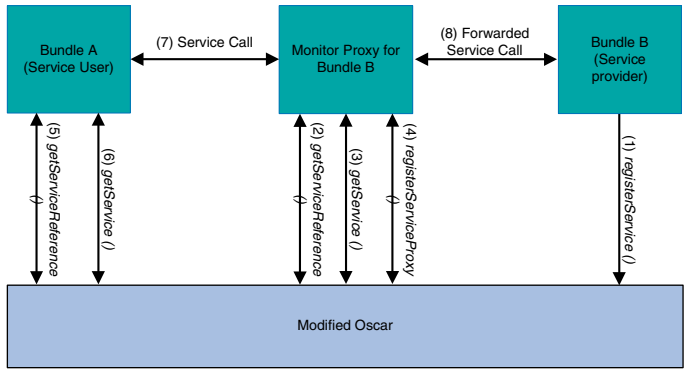


Figure 22. Monitor proxy registration procedure and usage.

When the service user desires to use a service, instead of calling a method of a service provider, it calls a method of the monitor proxy as in Figure 20. To enable the monitoring of the service provider the proxy needs to change the *ThreadGroup* of the thread as was explained in Chapter 4.2.1. Java does not allow the changing of the group of the thread, and so, we need to spawn a new thread in the service provider's group that performs the service interaction. This is illustrated in Figure 23, which is a snippet from a source code taken from the *Monitor Proxy* shown in Figure 19.

```
public class MonitorProxy implements BundleActivator,
                    BundleBService {
    private Data data = null;
    public Data service() {
        Thread serviceThr = new Thread(BundleBGroup,new serviceThread());
        serviceThr.start();
        serviceThr.join();
        return data;
    }
    class serviceThread implements Runnable {
        public void run () {
            data = BundleB.service();
        }
    }
}
```

Figure 23. Source code snippet from Monitor Proxy.

As shown in Figure 23, the parent thread of the *serviceThr* waits for the service interaction to complete. After the service provider has responded to the service call, the thread responsible for the service call will die. The death of this service thread unblocks its parent thread and the execution goes on. There will be no need for further actions since the execution now returns to the service user and the parent thread is already in corresponding thread group, hence the consumed resources will be tracked to the appropriate bundle.

4.3 The monitor agent

The previous section discussed the implementation details that enable the resource consumption monitoring of a separate bundle. This section opens up the actual monitoring agent implemented in this thesis work. The main responsibility of the monitor agent is shown in Figure 24. The actual identification of the threads is performed in the monitor agent. As this identifies the resource consuming bundle, it is the most crucial task of the agent.

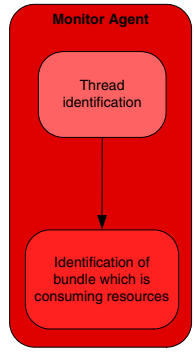


Figure 24. Thread identification enables the identification of the resource consuming bundle.

Various approaches to JVM monitoring approaches were studied in Chapter 2.3. We implemented a monitoring agent by using the JVM TI and BCI. The use of this interface was chosen as it provides the best approach to collect runtime data. The BCI is needed in order to get detailed monitoring data. The JMX based implementation was discarded because it did not provide the means to extract detailed bundle specific information.

The actual implementation is a dynamically linked library (DLL) written in C and it uses two demo library files delivered with the JDK. These are the agent_util.c that provides a few utility functions for the JVM TI agents and the java_crw_demo.c that provides the functionality required for performing the basic BCI. The implementation utilizes both time- and event-based monitoring techniques and the BCI. Utilizing all the different techniques was studied to be the most efficient approach in Chapter 2.3.1. The monitoring agent reacts on the events that it receives from the JVM TI. In addition, the monitoring agent has its own thread for periodically checking the CPU usage of the threads and for collecting heap memory usage information.

The following chapters open up the agent implementation from the initializing phase to the actual monitoring phase.

4.3.1 Initializing the agent

The monitor agent is loaded right at the startup phase of the JVM and the Agent_OnLoad-function is called. In this phase, the agent defines the subjects of interest and prepares itself before the actual execution. This function parses the configuration file, finds out the number of processors in the execution environment, gets the JVM process handle from OS and sets the state of the JVM TI environment, as was described in Chapter 2.3.3.

A configuration file defines the desired bundles to be monitored and additional options. An example configuration file is included in Appendix 1. Only the bundles that have been defined in this file will be monitored, the remainder are ignored by the agent. With additional options, the user of the agent can the define following optional properties of the agent:

- Classes which will be instrumented with new bytecodes.
- CPU sampling interval.
- Memory sampling interval.
- Garbage collector interval.
- Filename of the monitor log.

To collect the desired information, the state of the JVM TI environment of the agent must be set. This includes defining and registering events of interest and actions taken when a certain event occurs. The events of interest and desired functionality provided by the JVM TI are defined with *capabilities*, *event notifications* and *event callback*-functions. The following lists these and their purposes.

The following *capabilities* are added:

- can_generate_all_class_load_hook_events: The event is raised for every class loaded by the JVM. This is to enable the BCI.
- can_tag_objects: To tag the desired allocated objects in order to identify them from the heap memory.
- can_generate_vm_object_alloc_events: The event is raised when the JVM allocates an object that is not visible with BCI.
- can_get_thread_cpu_time: To get the CPU times used by the threads.

The following *event notifications* and corresponding *event callback*-functions are set:

- JVMTI_EVENT_VM_START: In the start phase of the JVM, the native methods needed for memory allocation tracking are registered.
- JVMTI_EVENT_VM_INIT: Start an agent thread that periodically samples the CPU usage and logs the heap usage of bundles.
- JVMTI_EVENT_VM_DEATH: Run down the monitor agent.
- JVMTI_EVENT_VM_OBJECT_ALLOC: Log object allocations that are not caught with BCI.
- JVMTI_EVENT_CLASS_FILE_LOAD_HOOK: BCI for every desired class.
- JVMTI_EVENT_THREAD_START: Check every new thread and add to monitored threads if desired.
- JVMTI_EVENT_THREAD_END: Clear thread from the monitored threads.

The number of processors available is needed in the data processing phase to calculate the CPU utilization and current process handle to get the information out of the OS concerning the whole JVM. After this initializing phase, the monitor agent starts tracking the resource consumption of the OSGi environment. The actual logging of the data, however begins after the JVM has been completely initialized and the agent receives the *JVMTI_-EVENT_VM_INIT*-event.

```
typedef struct BundleInfo {
    jlong       hashCode;
    /* name of the threadgroup == {bundlename}.jar */
    char        name[MAX_FILE_NAME];
    /* Total space taken up by objects allocated from this threadgroup/
       bundle */
    jlong       totalSpace;
    /* total cputime taken by this threadgroup/bundle */
    jlong       cpuTime;
    /* list of threadIDs running in this threadgroup, -1 if thread has died
       */
    DWORD       threadIds[MAX_THREADS_PER_BINFO];
    /* index of last thread id */
    int         threadIndex;
    /* pointers to the prev/next BundleInfo in the list */
    struct      BundleInfo  *next;
    struct      BundleInfo  *prev;
}BundleInfo;
```

Figure 25. Structure to store bundle information.

For every bundle that will be monitored, a *BundleInfo*-structure will be created that stores the necessary information of the bundle. These structures are created in the initializing phase of the agent right after the configuration file is read. The *BundleInfo*-structure is presented in Figure 25. This structure will store all the threads for this bundle, and so, these define all the monitored threads. As the bundle's thread group is a string of characters, a unique hash will be created for every bundle in order to reduce the processing time of finding the right bundle during the runtime. All the *BundleInfo*-structures are stored in a linked list that is a global variable inside the monitor agent.

4.3.2 Resource monitoring

In the first prototype, we concentrated on observing the CPU usage and heap memory consumption of bundles as was designed in Chapter 3.1. Figure 24 illustrated the key activity of the agent, the thread identification. This chapter shortly presents the implementation of the runtime monitoring performed by the monitor agent.

Thread identification A high level algorithm for bundle identification was presented in Figure 8. This algorithm is expanded in Figure 26, in order to demonstrate the activities done by the agent.

EVENT: JVMTI_EVENT_THREAD_START
get thread group;
get hashcode of the group;
search for corresponding group;
if *thread belongs to thread group of monitored bundle* **then**
  get thread id;
  bundle→threadIds[bundle→threadIndx]=thread id;
else
  ignore thread;
end

Figure 26. Thread identification, straight after a new thread is spawned.

As this is done to all of the threads spawned in the JVM, the *BundleInfo*-structures store all the thread id's that we are interested in. The thread id is the most convenient way to represent a thread because it costs less processing time than character strings. Only the events caused by the threads, which are in our interest, will now be taken into account.

Heap memory monitoring The monitoring of heap memory allocations is performed by the BCI. All classes are instrumented with additional bytecodes straight after they are loaded by the JVM. These additional bytecodes are placed after every new object and array allocation. Additional bytecodes are method calls to monitoring agent with the newly created object as an argument. As was previously explained in Chapter 4.3.1, not all of the allocations are visible with the BCI, and so, there is an *event notification* enabled for these allocations. The actions taken by the agent are the same in both cases. Figure 27 illustrates the action sequence of a monitoring agent when there is a new object allocated.

Whenever a new object allocation occurs, the bundle that made this allocation will be searched. If the bundle responsible for the allocation is in our interest the newly created object will be tagged with a pointer to the corresponding *BundleInfo*-structure. Now all the objects in the heap memory area that are allocated by bundles, which are in our interest, are tagged with a unique bundle identifier.

```
EVENT: JVMTI_EVENT_VM_OBJECT_ALLOC
get current thread id;
search for corresponding group;
if thread belongs to monitored bundle then
  | tag object with BundleInfo;
else
  | ignore object;
end
```

Figure 27. Object tagging, straight after new object has been allocated.

The agent thread periodically traverses the heap memory and counts the aggregate size of the objects, which are tagged to individual bundle. The inspection of the heap memory is done by the *IterateThroughHeap*-function provided by the JVM TI. This function can filter out all the untagged objects, thus, only the objects that are in our interest are counted.

CPU time monitoring The CPU utilization monitoring of the bundles is performed by the agent thread. The purpose of the sampling thread is to periodically sample the running threads, count the aggregate CPU time taken by one bundle and update it in the corresponding *BundleInfo*-structure. A pseudo code illustrating thread sampling is shown in Figure 28.

```
get all the running threads;
for each running thread do
  | get thread info;
  | if thread belongs to monitored bundle then
  |   | get thread CPU time;
  |   | add to bundle CPU time;
  | else
  |   | ignore thread;
  | end
end
```

Figure 28. Basic operation of thread sampler.

All the used functions in thread sampling are provided by the JVM TI. Threads are not suspended during sampling, thus, there is a possibility that a thread could have

died before we get its CPU time. This is acceptable as it would be too interfering to suspend all the threads for sampling and the resolution of the thread CPU time provided by the JVM is not high anyway.

Besides sampling all the running threads, the agent thread also monitors the CPU time consumed by the JVM. The JVM is a one process in the OS, and so, we periodically retrieve this process CPU time with a system call.

4.3.3 File output

The monitor agent creates an output file that consists of numerical values representing the behaviour of monitored bundles and the OSGi environment.

As was explained in Chapter 4.3.2, the agent thread periodically samples both the heap memory and CPU time usage of the monitored bundles. The result of the sampling is written to the output file with a time stamp. The CPU time used by the JVM is presented in 100 nanosecond units, the CPU time used bundles and time stamp are represented in nanoseconds. The heap memory usage of bundles is presented in bytes.

A snippet from an example log file is included in Appendix 2. The first number in the file reveals the number of the processors in the monitored platform. This is used later on to calculate the CPU usage statistics.

4.4 The visualization and analysis tool

Two novel resource consumption visualization models were identified in Chapter 3.2. In order to realize these models, the visualization tool must read the log file and create a visualization based on the monitoring data. An existing visualization and analysis tool was extended in this thesis work as was explained in Chapter 3.2.

The visualization and analysis tool is written in C++. It uses an object-oriented approach, where all the main functionalities are separated into their own classes. The separate elements consist of a class for inputting data, a class for creating

drawable graphics from the data and an application logic class to implement the tool from these components. Actual implementation and implemented extensions are presented in this section.

4.4.1 Class structure

Figure 29 shows the most important classes of the original software [56]. The new implementations and classes that need to be extended in order to handle the resource consumption data of OSGi bundles are marked with red ellipses. The most relevant classes and their responsibilities will be then shortly described. In addition, the new derived classes and extensions are explained.

PerfFileReader This is an abstract base class. Its responsibility is to read input files of a predetermined form. Implementations of the reader class are registered by a registration mechanism in the class PerfFileReaderFactory. In order to handle the resource consumption data of the OSGi bundles, an implementation of a reader class was made. This class is the OSGIFileReader. It takes the output file of the monitor agent as its input.

DataChunk A DataChunk is the internal representation of all the data used in the visualization and analysis. All the data read into the memory by the reader class is inserted into the DataChunk. This class was extended with attributes needed to represent the OSGi bundle related data. These attributes include the CPU usage, memory usage, networking activity and the resource boundaries of individual bundles. In addition, the DataChunk holds general information on bundles, such as names.

ModelTriangleObject This is an abstract base class. The ModelTriangleObject forms the basic building blocks of the graphics. Two different implementations of this class were made in this thesis work, one for both of the visualization models. In the resource usage model, the basic block consists of averaged resource usage statistics over a corresponding time interval. In the resource boundary model, the

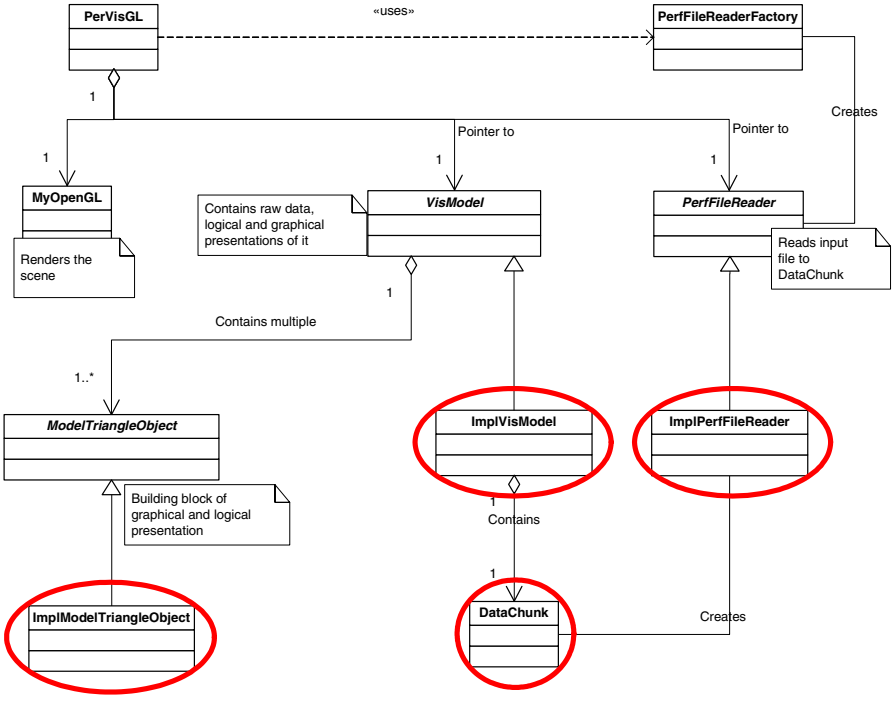


Figure 29. Class diagram of the original software.

basic block consists of the maximum instantaneous resource usage statistics of a bundle and its resource boundaries. Both implementations calculate the graphics based on the input data that is read into the DataChunk by the reader class.

VisModel This is an abstract base class. It manages the vector of the ModelTriangleObjects, calculates reports and analyzes them. This class is responsible for transforming the real-world measurement values to the OpenGL coordinates. Two different implementations of this class were made in this thesis work, one for both visualization models. The VisModel implementations contain the original DataChunk. It is used for counting statistics, when the user selects some portion of the visualization.

PerVisGL This is the main class of the program containing the glue logic that is first created and first called from the main function. This class also implements the Graphical User Interface (GUI) functionality. [56]

Figure 30 shows the most important classes of the software modified in this thesis work. The class diagram only shows the classes relevant to this work.

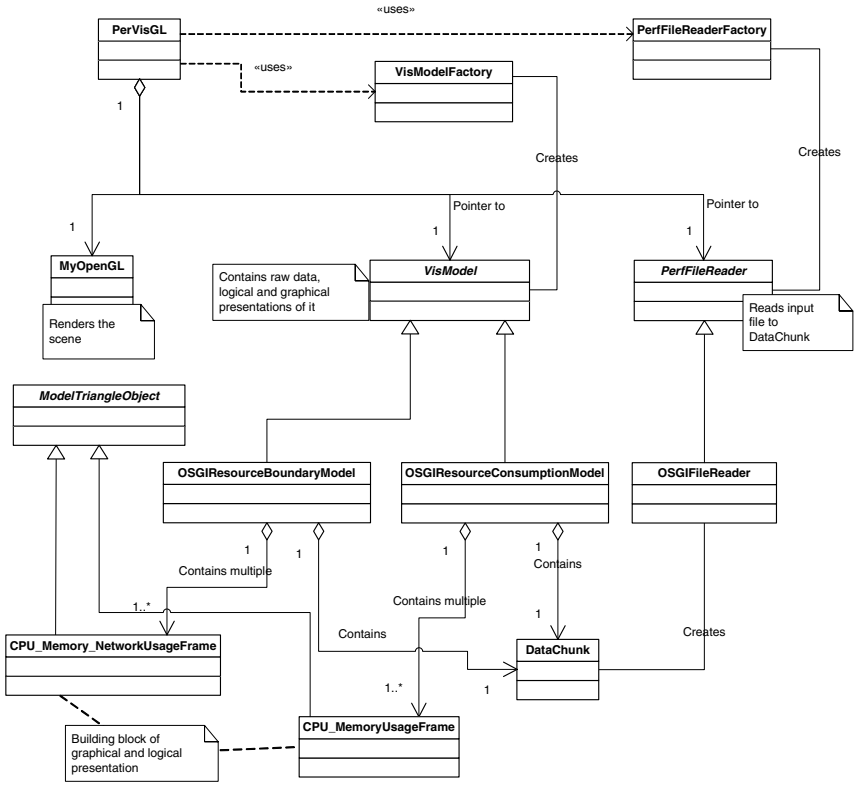


Figure 30. Class diagram of the extended software.

All of the classes and their responsibilities were described before, except the VisModelFactory. This class was created in order to improve the scalability of the software. It provides a registration mechanism for different VisModel implementations and brings them available to the user via the GUI, in the same way as the PerfFileReaderFactory does with different implementations of the reader class.

4.4.2 File output

A visualization and analysis tool creates an output file, which consists of numerical values representing the behaviour of the monitored bundles and the OSGi environment. The output is calculated automatically from the portion that the user selects from the visualization.

In the resource consumption model, the main information in the output is the length of the selected time frame, CPU utilization and the memory consumption of individual bundles. Both are averaged over the selected time frame, and the maximum and minimum memory consumption values are also included. The CPU utilization of a bundle is also compared to the CPU utilization of the JVM.

In the resource boundary model, the output information includes maximum instantaneous CPU utilization, memory consumption and networking activity, all the corresponding resource usage boundaries and relations to these are included. In addition, the information includes the time distribution of the CPU utilization that clarifies CPU utilization behaviour of a bundle.

Appendix 3 shows two examples of an output file, one for both models.

4.5 OSGi-based resource monitoring service

Chapter 3.3 discussed the dynamic adaptation support in the OSGi platform. It was identified that we need to implement an OSGi service to provide the resource consumption information to the components that can adapt their behavioúr. This service provides unified access to the data collected by the monitor agent.

A resource monitoring service is a component written in Java although we need to use the Java Native Interface (JNI) to gain access to the monitor agent since it was written in C. The service component introduces native methods that are introduced as native methods in the monitor agent. These native methods are used as normal methods, although, these reside in the monitor agent.

Using the OSGi-based resource monitoring service requires that the JVMT TI monitor agent, described in Chapter 4.3, has been started. Figure 31 clears the relations between the components, which are participating in the dynamic adaptation interaction.

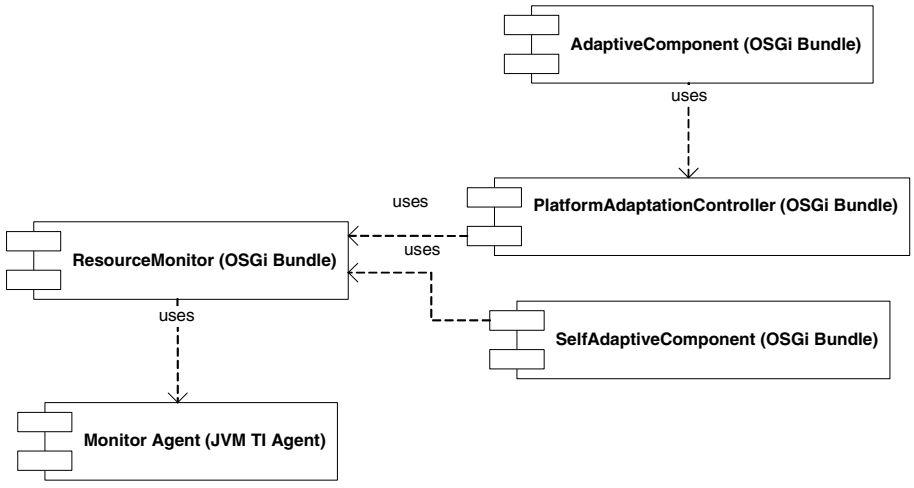


Figure 31. Component diagram of the OSGi-based dynamic adaptation.

Various types of adaptation schemes were discussed in Chapter 3.3. The ResourceMonitor bundle does not consider the adaptation scheme. The implementation of the service supports both self-adapting and platform controlled adaptive software. The responsibility of resource monitoring service is to provide resource consumption information and not to handle the adaptation itself. The following section presents the actual implementation of the service.

4.5.1 Class structure

Figure 32 shows the class structure of the service component. The most relevant classes are then described.

AdaptivityServiceInterface This interface is registered to the OSGi Framework as a service provided by the ResourceMonitor class. It offers registration mechanisms for bundles that need to obtain resource consumption information of their own or the JVM.

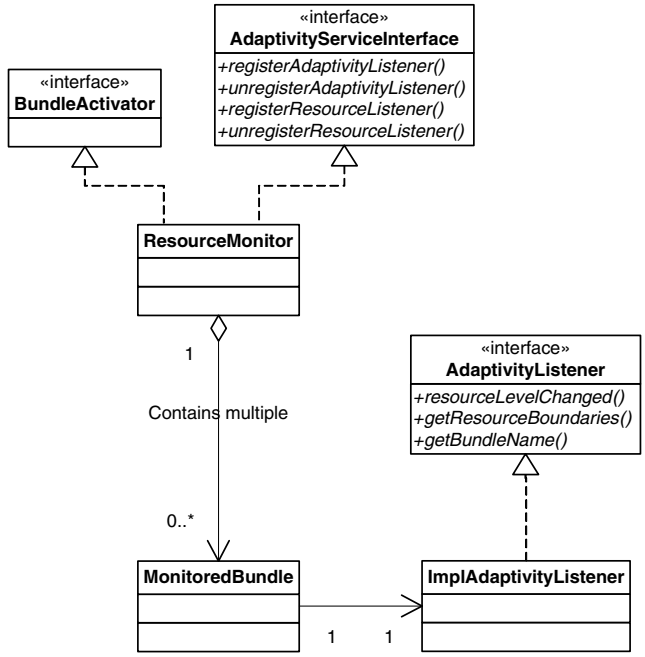


Figure 32. Class diagram of the resource monitoring service.

MonitoredBundle The ResourceMonitor creates an instance of the MonitoredBundle class for every bundle that registers as a user of this service. The MonitoredBundle class contains all the relevant information on this bundle. This includes reference to implementation of the AdaptivityListener, the bundle name and its resource usage boundaries. The implementation of the AdaptivityListener interface is used to notify the bundle of changes in its resource usage levels.

ResourceMonitor This class is the main class of the monitoring service. The ResourceMonitor handles the list of registered bundles and registers these to the monitor agent. It updates the resource consumption information of all the registered bundles periodically by accessing the monitor agent. These are updated to the corresponding MonitoredBundle.

5. Experimentation

This chapter presents the usage of implemented monitoring and visualization tools with two different test cases. These validate the applicability of the tools from both monitoring and analyzing perspective. Firstly, an example of monitoring a single OSGi bundle and the related visualizations are presented. Secondly, the monitoring tool is used with adaptive components and visualizations are used to demonstrate how these tools can be used to evaluate the dynamic adaptation. The second test case also demonstrates how the implemented tools can be used in analyzing multiple bundles.

The tools were evaluated with an AOpen miniPC that was using an Intel T2500 2.0 Gigahertz Core Duo processor and one gigabyte memory. The PC was running a 32-bit Microsoft Windows XP OS and a modified Oscar was executed on top of Sun's Java Runtime Environment (JRE) version 6. The maximum heap memory size for the JVM was defined to be 64 megabytes (MB).

5.1 Monitoring single OSGi bundle

At first, the implemented tools were used to validate the resource usage boundaries for an OSGi bundle. The monitoring tool collected run-time behavioural data of the chosen bundle during the various test cases. These test cases included different configurations of the chosen bundle. All these different configurations had unique resource requirements. By testing different configurations, we were able to capture the typical resource usage levels of the bundle. These configurations include the low usage of computing resources as well as the worst case scenario from a resource consumption perspective.

This test case shows an example of, how the implemented tools can be used to detect and validate the resource consumption boundaries for a software component. It was explained in Chapter 3.2 that a goal of these tools is to evaluate software behaviour against predefined resource usage boundaries. There were no predefined boundaries for this component and we specified the boundaries with the aid of our analyzing tools. Although it is not possible to design test cases that will be comprehensive and capture all the possible resource usage levels of a component, we

were able to specify the typical resource consumption behaviour of the software component.

5.1.1 Monitored OSGi bundle

We applied implemented tools to the Indexing Service component that was responsible for acquiring, indexing and providing dynamically changing data. The Indexing Service, and the data it provided, simulated continuously changing sensor data. The Indexing Client was a distributed component that utilized the sensor data. These communicate via Generic Communication Middleware (GCM) [57]. Figure 33 presents the test case and the relevant bundles.

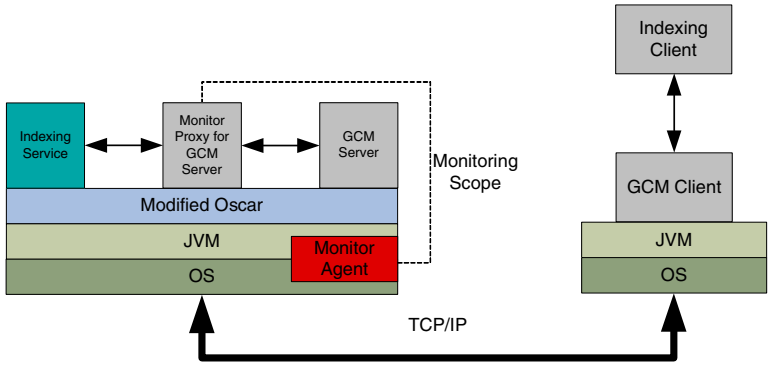


Figure 33. Simulation arrangement for Indexing Service.

We needed to use a Monitor Proxy because the GCM server and Indexing Service are OSGi services as was previously introduced in Chapter 4.2.2. The communication flow in this test case is simple: the Indexing Service acquires sensor data, indexes it and periodically the Indexing Client acquires the indexed data from the Indexing Service.

As was stated in the previous section, we tried to distinguish the typical and different behaviour levels of the Indexing Service from a resource consumption point of view. These levels were identified with simulations of real life use cases of the service component. The Indexing Service has two different operation modes which differ in resource consumption, round-trip time (RTT) and the consistency of the dynamically changing data. The resource consumption boundaries were identified for the Indexing Service in both operation modes.

The operating modes of the Indexing Service define the scheme, which is used in storing the constantly changing data. The first operation mode uses memory to store the sensor data. The second operation mode stores the sensor data to a database. Obviously, the first operating mode provides better QoS but needs more computing resources from the platform.

The quality attributes of the Indexing Service are data consistency and RTT. The data consistency of the provided data was determined by the update time of the continuously changing data. The RTT was the time elapsed in the client side from request message send and response message received. These were measured in order to show the changes in the QoS with two different operation modes. Both cases were also executed without a monitoring agent, so we were able to estimate the disturbance caused by the monitoring tool to the test system.

5.1.2 Resource consumption measurements

As was stated in the previous section, the Indexing Service has two operating modes. The resource consumption of the bundle was observed in both of the modes. These observations were made by individual test cases. The Indexing Client sent 120 request messages in both cases. After each response message received, the client waited for half a second until it sent a new request. To provide a view of the quality changes of the service, the RTT and update time of sensor data was also measured. The RTT and the update time of the constantly changing data was averaged over all the 120 messages, in order to obtain a statistical view to the response time and the data consistency.

The sequence of one test message is illustrated in Figure 34 and the following presents the action sequence of the figure:

1. The Indexing Client sends a message that requests a sensor data corresponding one index.
2. The GCM is used for message delivery to the Indexing Service.
3. The Monitor Proxy forwards the request message.

4. The Indexing Service encapsulates the requested sensor data to the response message and sends it back to the Indexing Client.

5. The Monitor Proxy forwards the response message.

6. The GCM delivers the response message to the Indexing Client.

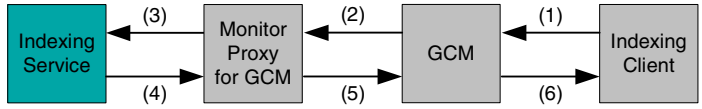


Figure 34. Sequence of one test message.

This section presents the measuring results with the views provided by the previously introduced monitoring and visualization approach. The provided views were explained in Chapter 3.2. Both of the operation modes of the Indexing Service are included.

Operation mode 1 The Index Service running in operation mode 1, uses heap memory to store sensor data. Figure 35 illustrates the resource usage of the Index Service during the test case. It can be seen that the CPU utilization remained steadily near 50 % utilization. This shows that the Index Service reserved one core of the processor throughout the test case, as it was stated that the test system consisted of dual core processor. The Figure also shows that monitored bundle reserved nearly all the CPU taken by the JVM. This indicates that there were no other bundles running in the OSGi environment. The heap memory consumption increased steadily to 3 MB, and then it dropped to the normal usage of the monitored bundle and began to grow again. This is normal behaviour for a Java application as the heap memory usage typically grows until the JVM performs a complete garbage collection.

Figure 35 presented a time-dependent view of the test case. Figure 36 provides a different view of the same test case. This encapsulates the averaged maximum instantaneous usage values of different computing resources. These can be interpreted as what the computing platform should be able to provide for a bundle, in order to function properly.

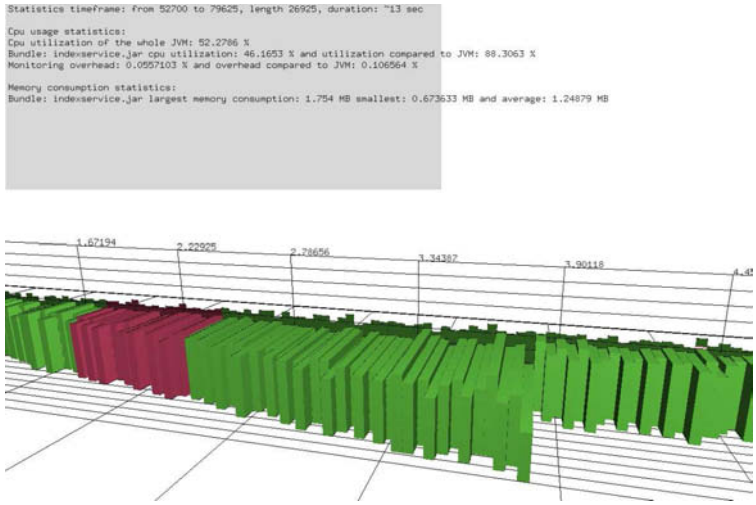


Figure 35. Resource consumption of the Indexing Service in operation mode 1.

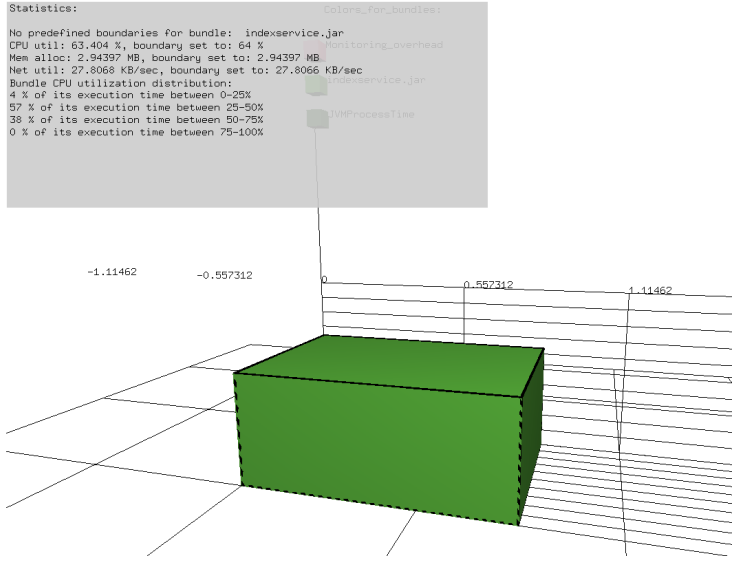


Figure 36. Indexing Service resource boundaries in operation mode 1.

The Index Service, which was running in operation mode 1, consumed 64 % of the CPU at its maximum, nearly 3 MB of heap memory and the network bandwidth usage was about 28 kilobytes per second. Statistics also show that the CPU

utilization of the monitored bundle was between 25 % and 75 %, throughout its execution. The QoS attributes in operation mode 1 were the following:

- Average RTT was 25.6 milliseconds.
- Average update time of the sensor data was 0.136 milliseconds.

Operation mode 2 The Index Service running in operation mode 1, uses a database to store sensor data. Figure 37 illustrates the resource consumption of the Index Service during the test case. It can be seen that the CPU utilization of the bundle was nearly zero throughout the test case. This is due to the usage of a database as storing and reading operations take so much time that the bundle is usually waiting for this operation. The heap memory usage of the Index Service is also very small; it varies between 0.1 MB and 0.5 MB.

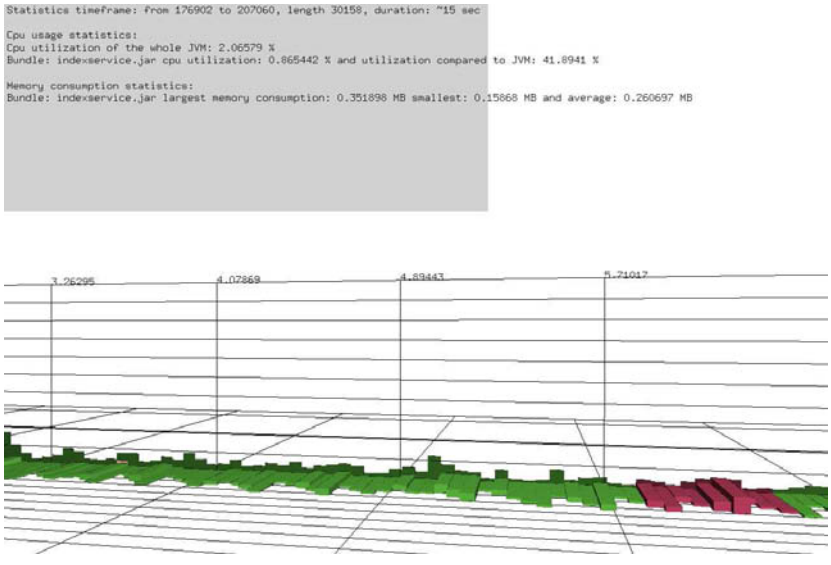


Figure 37. Resource consumption of the Indexing Service in operation mode 2.

Figure 38 illustrates the averaged maximum instantaneous usage values of the different resources in this test case. The boundaries around the resource consumption box are taken from the previous test run, where the Index Service was running in operation mode 1.

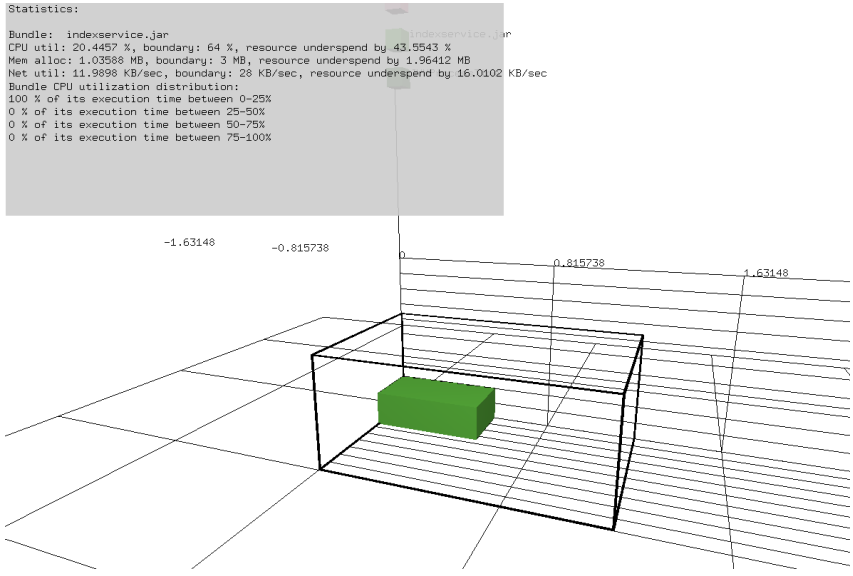


Figure 38. Indexing Service resource usage in operation mode 2 with boundaries from operation mode 1.

The Index Service that was running in operation mode 2 consumed 21 % of the CPU at its maximum, about 1 MB of heap memory and the network bandwidth usage was about 12 kilobytes per second. Statistics also show that the CPU utilization of the monitored bundle remained below 25 % throughout its execution. The QoS attributes in operation mode 2 were the following:

- Average RTT was 78.5 milliseconds.
- Average update time of sensor data was 11.61 milliseconds.

Summary To get a reliable view of the resource consumption of the Indexing Service in both test cases, both were run 5 times. Table 3 presents the averaged results of these five cases. It summarizes the consumption of the monitored resources and the quality of service attributes in two different operation modes of the Indexing Service. The performance ratio in this table means the switch from operation mode 1 to operation mode 2.

Table 3. Summary of test cases.

Attribute	Operation mode 1	Operation mode 2	Performance ratio
CPU	63.8 % ± 0.75 %	22.6 % ± 1.36 %	1 : 0.35
Memory	2.03 MB ± 0.72 MB	1.09 MB ± 0.07 MB	1 : 0.54
Network	20.1 kB/s ± 6.18 kB/s	11.4 kB/s ± 2.57 kB/s	1 : 0.57
Data Update	0.136 ms ± 0.01 ms	11.61 ms ± 0.07 ms	1 : 85.43
RTT	25.6 ms ± 2.91 ms	78.5 ms ± 3.81 ms	1 : 3.06

The standard deviations of these 5 runs are shown in Table 3. It shows that the deviation of the measured attributes between the test runs is very low, except that the deviation in the networking activity is considerable. This indicates that we had been able to capture the behaviour of the Index Service reliably and repeating test cases would be unnecessary.

5.1.3 Discussion on the results

The test cases introduced the Index Service in two fundamentally different operation modes. Table 3 shows a great difference in resource usage between these two modes. The difference is clearly highlighted in Figure 38. It can also be seen that the savings in resource usage, costs highly in the quality of the provided service. The utilization of the CPU especially varies greatly between the operation modes. This difference is highlighted when comparing Figures 35 and 37, which illustrate the resource consumption throughout the whole test cases.

Results include the resource boundaries for the Index Service in two operating modes. From the boundaries, we are able to determine the resource consumption savings we will achieve, if we switch from operation mode 1 to operation mode 2. These are shown in Table 3. The CPU utilization drops almost to one third, the heap memory consumption and network bandwidth utilization almost to a half of the original levels. These savings cause a great reduction to the QoS provided by the Index Service. The RTT is 3 times longer and the sensor data update time is over 85 times longer than the original times. These can now be taken into consideration, if the computing platform is in a situation where it is running out of computing resources.

The resource consumption boundaries of the Index Service, found out with previous test cases, can be included as one performance attribute of the Index Service. These are now validated and can be used when comparing two individual implementations of the bundle providing the same service. The boundaries can also be used in determining whether the target platform is able to provide a sufficient amount of appropriate resources in order to execute the Index Service bundle. Obviously, for example, the CPU utilization gives just a view of how CPU intensive the Index Service is, because an accurate value is correct only in this computing platform.

5.2 Runtime support provided by the monitoring service

In the second use case, the implemented tools were applied to an environment that had multiple bundles running. In this case, the platform used the previously validated resource usage boundaries of the Index Service, presented in the Chapter 5.1. This information was used when computing resources were running low in the platform. The runtime resource consumption information was provided by the monitoring service introduced in Chapter 4.5. This case demonstrated the runtime use of the monitor agent and monitoring service. In addition, it demonstrated the use of implemented tools in monitoring and analyzing the resource consumption behaviour of multiple bundles.

5.2.1 Arrangements and case flow

In this case we used the Midgate platform [58, 59] which extends the service component model provided by the OSGi environment. The Midgate also provides support for adaptive middleware components [60]. Only the Index Service was running on top of the Midgate platform in this case. Figure 39 presents all the bundles of the test case.

There was the same sensor data indexing service and client executed as presented in Chapter 5.1. The Midgate platform is able to control the operation mode of the Index Service. The controlling is based on the resource consumption information that is provided by the Resource Monitor bundle. BundleA and BundleB are

general applications, which do not provide any services. These simulate the user applications executed in the OSGi environment.

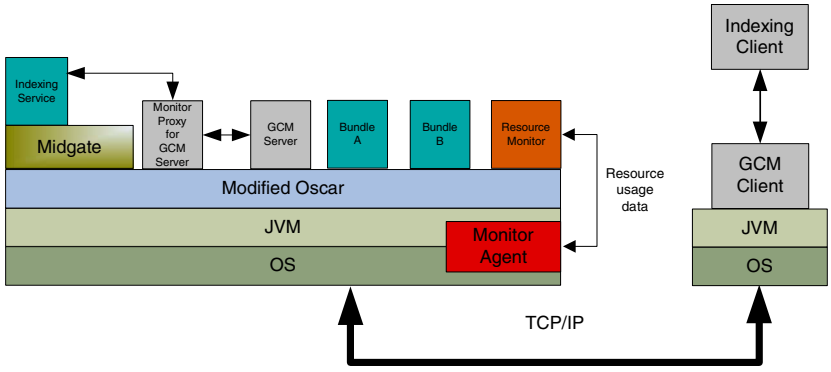


Figure 39. Arrangement for the test case.

The activity periods of the relevant bundles are presented in Figure 40. The Index Service was started in operating mode 1, and so, it provided the best QoS. Initially it was the only bundle executed in the environment. After a while, more bundles were started, at first BundleA and then BundleB. After 1 minute and 30 seconds, bundles A and B were stopped then the situation was the same as it was initially, where only the Index Service was running.

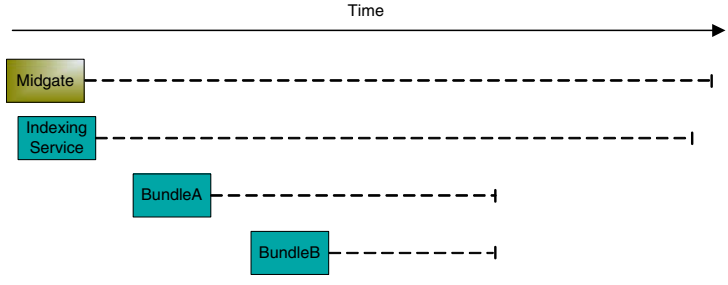


Figure 40. Activity periods of the bundles in the test case.

In the test case, the Midgate platform was responsible for the availability of the computing resources. It was defined that the CPU utilization of the whole OSGi environment must remain below 80 %. The Index Service was the only bundle that could be adjusted according to the availability of computing resources, so

the only way to release resources was to control the operating mode of the Index Service. The Midgate used the resource consumption knowledge that was gathered in Chapter 5.1.

5.2.2 Resource consumption statistics

The resource consumption of the bundles during the test case is illustrated in Figure 41. Letters A–F present the different periods of the case.

At first, the Midgate platform initializes itself and reserves the memory it requires. After initialization, the Midgate's CPU utilization is very low and the heap memory consumption is steady. This is because its only responsibility in the case is to control the operating mode of the Index Service.

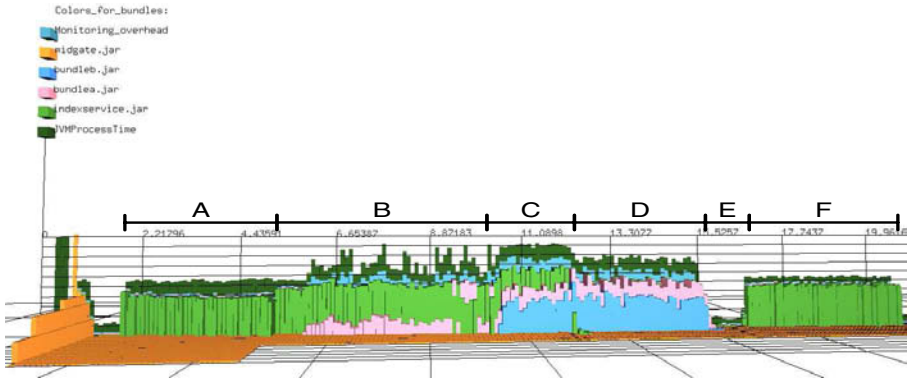


Figure 41. Resource consumption of the bundles.

Figure 42 presents the resource consumption statistics of the test case provided by the visualization and analyzing tool. In Figure 42a, statistics of period A are shown. Period A begins when the Index Service was started. The Midgate was already initialized and its heap memory usage dropped to 5 MB, which is its standard level. The behaviour of the Index Service followed the same pattern as was measured in the Chapter 5.1.2.

At the beginning of period B, Bundle A was started. Bundle A quite steadily utilized about 11 % of the CPU. The CPU utilization of the OSGi environment increased to near 70 %. As it remained below 80 %, the Midgate did not need to

```
Input file: PERIOD_A.txt
Statistics timeframe: from 60880 to 176165, length
115285, duration: ~57 sec

Cpu usage statistics:
Cpu utilization of the whole JVM: 51.6572 %
Bundle: indexservice.jar Cpu utilization: 38.1515 % and
utilization compared to JVM: 73.8552 %
Monitoring overhead: 0.0407685 % and overhead compared to
JVM: 0.0789213 %

Memory consumption statistics:
Bundle: indexservice.jar largest memory consumption:
1.86473 MB smallest: 0.234836 MB and average: 0.989622 MB
Bundle: midgate.jar largest memory consumption: 10.1528
MB smallest: 5.0008 MB and average: 9.94007 MB
```

(a) Statistics of period A.

```
Input file: PERIOD_B.txt
Statistics timeframe: from 172970 to 333112, length
160142, duration: ~80 sec

Cpu usage statistics:
Cpu utilization of the whole JVM: 67.3933 %
Bundle: indexservice.jar Cpu utilization: 37.714 % and
utilization compared to JVM: 55.9611 %
Bundle: bundlea.jar Cpu utilization: 11.4292 % and
utilization compared to JVM: 16.959 %
Monitoring overhead: 3.7217 % and overhead compared to
JVM: 5.52235 %

Memory consumption statistics:
Bundle: indexservice.jar largest memory consumption:
2.12559 MB smallest: 0.275914 MB and average: 1.10579 MB
Bundle: bundlea.jar largest memory consumption: 1.66523 MB
smallest: 0 MB and average: 1.39038 MB
Bundle: midgate.jar largest memory consumption: 5.0008 MB
smallest: 5.0006 MB and average: 5.00079 MB
```

(b) Statistics of period B.

```
Input file: PERIOD_C.txt
Statistics timeframe: from 339539 to 406781, length 67242,
duration: ~33 sec

Cpu usage statistics:
Cpu utilization of the whole JVM: 87.3769 %
Bundle: indexservice.jar Cpu utilization: 19.7853 % and
utilization compared to JVM: 22.6436 %
Bundle: bundlea.jar Cpu utilization: 13.4276 % and
utilization compared to JVM: 15.3675 %
Bundle: bundleb.jar Cpu utilization: 29.5544 % and
utilization compared to JVM: 33.8241 %
Bundle: midgate.jar Cpu utilization: 0.138306 % and
utilization compared to JVM: 0.158287 %
Monitoring overhead: 9.24868 % and overhead compared to
JVM: 10.5848 %

Memory consumption statistics:
Bundle: indexservice.jar largest memory consumption:
2.77591 MB smallest: 0.324242 MB and average: 0.990821 MB
Bundle: bundlea.jar largest memory consumption: 1.66523 MB
smallest: 1.66523 MB and average: 1.66523 MB
Bundle: bundleb.jar largest memory consumption: 2.0252 MB
smallest: 0 MB and average: 1.84072 MB
Bundle: midgate.jar largest memory consumption: 5.10358 MB
smallest: 5.0006 MB and average: 5.01101 MB
```

(c) Statistics of period C.

```
Input file: PERIOD_D.txt
Statistics timeframe: from 410019 to 512480, length 102461,
duration: ~51 sec

Cpu usage statistics:
Cpu utilization of the whole JVM: 71.9093 %
Bundle: indexservice.jar Cpu utilization: 0.995501 % and
utilization compared to JVM: 1.38438 %
Bundle: bundlea.jar Cpu utilization: 14.4679 % and
utilization compared to JVM: 20.1197 %
Bundle: bundleb.jar Cpu utilization: 33.197 % and
utilization compared to JVM: 46.1651 %
Monitoring overhead: 9.56364 % and overhead compared to
JVM: 13.2996 %

Memory consumption statistics:
Bundle: indexservice.jar largest memory consumption:
2.93541 MB smallest: 0.299398 MB and average: 0.698256 MB
Bundle: bundlea.jar largest memory consumption: 1.66523 MB
smallest: 1.66523 MB and average: 1.66523 MB
Bundle: bundleb.jar largest memory consumption: 1.86895 MB
smallest: 1.86895 MB and average: 1.86895 MB
Bundle: midgate.jar largest memory consumption: 5.03273 MB
smallest: 5.02255 MB and average: 5.02427 MB
```

(d) Statistics of period D.

```
Input file: PERIOD_E.txt
Statistics timeframe: from 515952 to 544510, length 28558,
duration: ~14 sec

Cpu usage statistics:
Cpu utilization of the whole JVM: 6.52357 %
Bundle: indexservice.jar Cpu utilization: 0.987464 % and
utilization compared to JVM: 15.1369 %
Bundle: midgate.jar Cpu utilization: 0.108551 % and
utilization compared to JVM: 1.66398 %

Memory consumption statistics:
Bundle: indexservice.jar largest memory consumption: 1.204 MB
smallest: 0.180891 MB and average: 0.502929 MB
Bundle: bundlea.jar largest memory consumption: 1.66523 MB
smallest: 0.00496094 MB and average: 0.457761 MB
Bundle: bundleb.jar largest memory consumption: 1.86895 MB
smallest: 0.00496094 MB and average: 0.51332 MB
Bundle: midgate.jar largest memory consumption: 5.02991 MB
smallest: 5.02151 MB and average: 5.02318 MB
```

(e) Statistics of period E.

```
Input file: PERIOD_F.txt
Statistics timeframe: from 547722 to 679036, length 131314,
duration: ~65 sec

Cpu usage statistics:
Cpu utilization of the whole JVM: 50.8392 %
Bundle: indexservice.jar Cpu utilization: 44.2908 % and
utilization compared to JVM: 87.1193 %
Monitoring overhead: 0.0121845 % and overhead compared to JVM:
0.0239668 %

Memory consumption statistics:
Bundle: indexservice.jar largest memory consumption: 1.60774 MB
smallest: 0.371242 MB and average: 0.870333 MB
Bundle: bundlea.jar largest memory consumption: 0.00496094 MB
smallest: 0.00496094 MB and average: 0.00496094 MB
Bundle: bundleb.jar largest memory consumption: 0.00496094 MB
smallest: 0.00496094 MB and average: 0.00496094 MB
Bundle: midgate.jar largest memory consumption: 5.02479 MB
smallest: 5.02116 MB and average: 5.02158 MB
```

(f) Statistics of period F.

Figure 42. Outputs of the analyzing tool.

make any adjustments to the operating mode of the Index Service. There was still a 10 % gap to the defined limit.

At the beginning of period C, Bundle B was started. Bundle B reserved nearly 30 % of the available CPU time. As the utilization of the CPU was approaching overuse, the system was not able to provide a sufficient proportion of CPU time to the bundles. The CPU utilization of the OSGi environment was nearly 90 %, so the pre-defined limit was exceeded by 10 % during period C. This limit violation lasted for 33 seconds.

Period D started after the Midgate forced the Index Service to operating mode 2. This action dropped the CPU utilization of the Index Service to below 1 %, as was found out already in Chapter 5.1.2. The CPU utilization of the OSGi environment also dropped to an acceptable level. Therefore, it is clear that the Midgate released resources for bundles to use with its action.

Period E was a time frame where bundles A and B were stopped. This reduced the CPU utilization of the environment to below 10 %. Therefore, there was now the capacity to change the operation mode of the Index Service back to its initial mode. This period of low resource usage lasted for about 15 seconds.

Period F started after the Midgate forced the Index Service back to operating mode 1. The system was now back to the same state as in period A. The Index Service was the only active bundle in the environment. In addition, it can be seen that the automatic memory management of Java was not able to release all of the heap memory allocated by bundles A and B.

5.2.3 Discussion on the statistics

The measurements of the last chapter concentrated on the CPU utilization of the bundles. This is because all the bundles in this case consumed very little heap memory and the Index Service was the only bundle that utilized the network. The largest heap memory allocations were made by the Midgate platform at its initialization phase, a little over 10 MB. Other bundles reserved at most less than 3 MB. That is the reason why the heap memory consumption is ignored in the discussion.

Figure 41 shows that the performance requirements were fulfilled at first. The first violation of requirements was made at period C. This violation of the CPU utilization requirement persisted for over half a minute. The Midgate platform was too slow to react to this violation. The blame for the long reaction time can be shifted onto the Resource Monitor bundle since the Midgate reacted immediately after it was notified by the Resource Monitor. The reason why the notification was so slow is the averaging of the CPU utilization values. The rise of the CPU utilization is seen as delayed by the Resource Monitor. This is due to the target of avoiding false notifications, as was explained in Chapter 3.3.

Switching from low resource consumption to a higher level at period E, the Resource Monitor reacted 2 times faster than the opposite switch. This is because the decrease in the CPU usage was much more drastic than the growth between periods B and C. Even though the reaction was faster, there is still lot to improve. Improvements can be achieved by reducing the averaged samples or by using a more sophisticated monitoring algorithm.

6. Discussion

The main goal of this thesis work was to provide tools that enable the analysis of OSGi bundles from a resource consumption perspective. This chapter discusses the implemented tools and the applicability of these. Future improvements of the tools are also under discussion. Firstly, the accuracy of the measurements provided by the monitoring tool, the overhead introduced by the tool and the applicability of the tool are discussed. Secondly, the applicability of the visualizations is discussed. Finally, the applicability of both tools is discussed.

6.1 The monitoring tool

The overall requirements defined for the monitoring tool were achieved rather well. The monitoring tool was shown to be able to collect resource consumption data of individual Java components. This feature provides the basis for revealing the inner behaviour of the JVM and the behaviour of a software component.

At the time of writing this thesis work, not all of the features of the monitoring tool were yet implemented. The required BCI for logging network activity was not performed automatically by the tool. These additional bytecodes were added by hand before compiling.

Although we tried to find an approach that would not have required any modifications to any parts of the applications involved, we were not able to find this kind of approach. Small modifications to the underlying environment are however a small price to pay for detailed statistics. The worst case scenario would be that we would have to modify all the monitored applications by hand, which would also be prohibited by the requirements identified in Chapter 3.1. Modifications made to the Oscar lead to the fact that the monitoring tool is highly dependent on the platform. The monitoring tool is useless without the modified Oscar. This fact reduces the usability of the tool. If some analyst needs to use the monitoring tool, we need to provide not only the tool but also the modified platform. This makes the use of the tool too inflexible for broad scope usage.

6.1.1 Measurement accuracy

One of the prime requirements for the monitoring tool is the accuracy of the measurement data it provides. The whole monitoring tool loses its meaning if it provides inaccurate or even incorrect data. This section provides an estimation of the accuracy of the implemented tool.

Considering the accuracy of measuring the CPU time used by an OSGi bundle, we must take into account the smallest CPU time used that can be detected by our tool. In this work, we used the thread CPU times provided by the Windows XP OS and the CPU resolution time for a thread is 15.625 milliseconds. The resolution time defines the absolute precision that we cannot improve.

The monitoring tool also infects the measured CPU usage values. As the monitoring of heap memory usage and the networking activity was implemented with the BCI, the logging of these events increases the CPU time used by the corresponding bundle. This mostly affects the applications, which are highly heap memory intensive. The infected CPU time distorts and decreases the accuracy of the monitoring results. Nonetheless, the provided data is still useful and comparable with the data acquired by this same tool.

The heap memory consumption values, provided by the tool, give an exact amount of allocated memory by the bundle. The precision is one byte, so it is quite a high-precision value. One disadvantage that is introduced by our monitoring tool is that it samples the heap periodically, this leads to an uncertainty between two samples. In addition, a Java related issue could be seen which is caused by the automatic memory management. When a bundle releases an object, it is still seen by the monitoring tool, until the garbage collector collects it from the heap. This causes faulty statistics between two collections.

Objects, which have not been handled by the automatic memory management, are an arguable issue whether these should or should not be counted to the allocated memory. We could use continuous counting for heap memory allocations to provide an exact memory usage value at any point of time. This would require the catching of all the object free events, which would increase the overhead caused by the heap memory consumption monitoring. We ended up with the conclusion

that it would cause too much disturbance to the target system, so the continuous counting was discarded.

The networking activity was monitored with the BCI. This value represents all the bytes sent and received by the bundle. This value does not include the overhead introduced by the used transport protocol. Only the payload is counted. This decision was a matter of a point of view and simplicity, as was already discussed in Chapter 2.2.3.

6.1.2 Overhead introduced by the monitoring

The disturbance to the monitored system, caused by the monitoring tool, is a major issue to overcome in developing the monitoring tool. Not only because it slows the execution of the application, but it can also make it perform incorrectly. This section brings the main issues that cause the overhead under discussion and also evaluates the overhead caused by monitoring in the two test cases presented in the Chapter 5. The memory footprint of our monitoring tool is very low, so this chapter concentrates just on the CPU overhead.

The impact of the BCI was already discussed in Chapter 6.1.1 but performing the BCI does not only infer the accuracy of the measurement results, but it also infers the whole monitored system. All the classes loaded by the JVM must be inspected thoroughly and additional bytecodes must be added to the appropriate places in the class source code. The time taken by the BCI depends on the size and nature of the loaded class. Although this can add a significant time to class loading, it still needs to be done one time per loaded class. The once per class nature of the BCI makes it tolerable for the monitoring tool, because in the long run the significance of the overhead caused by the BCI itself is almost nonexistent.

Object tagging, during heap allocations made by bundles, can cause a significant overhead. This also depends on the nature of the monitored bundle. Let us consider a bundle that allocates a very large amount of small objects. For every object allocation, there will be one call to the monitoring agent function. As these calls are much slower than the normal Java method calls, these cause a great overhead. The time taken by one call to the JVMTI function is highly dependent on the execution platform so it cannot be precisely estimated.

The modifications to the OSGi environment also cause an overhead to the system. For every starting bundle, a new thread is created that starts the bundle. The creation of these threads impose almost a nonexistent overhead to the system because this is done only once per bundle. The monitoring proxies required for the OSGi services however can introduce quite a high overhead, as these spawn a new thread for every service transaction. The overhead introduced by these thread creations, depends on the execution platform, the number of proxies and the number of service transactions, so it highly depends on the monitored bundles and executed test case. The average thread creation time on the platform that was used in the test cases was measured to be 0.16 ms.

In this implementation, the log file created by the monitoring tool is a standard text file. The resource consumption data is logged continuously. Continuous file system usage loads the monitored system. This could be reduced in the future with binary format logging or by saving the resource consumption data straight to memory and writing the log file after the test case has been executed.

All the above mentioned sources of overhead are practically impossible to estimate during individual test cases. These can be classified as issues that we just have to tolerate. However, there are two factors that we can measure: the thread CPU time monitoring and heap iteration. Both of these are performed by the monitoring agent thread and the CPU time, taken by this thread, is measured with the same precision as all the other threads.

The periodical iteration over heap is also a great source of overhead. This is also hard to estimate as it is highly dependent on the used platform, the size of heap and the quantity of tagged objects. One way to reduce it in future implementations would be to perform the heap iteration immediately after each garbage collection. This would reduce the amount of heap iterations and also minimize the problem with already released objects, explained in Chapter 6.1.1.

Measured overhead During the test cases presented in Chapter 5., we measured the overhead caused by the monitoring tool by three different methods. The built-in measurement of the monitoring tool is the overhead caused by the sampling thread. In the test cases, we also measured the RTT and the data update time with

and without monitoring. The following discusses the overhead measurements of the test case presented in Chapter 5.1. Table 4 shows the averaged statistics of the test case, where the Index Service was running in operating mode 1.

Table 4. Monitoring overhead of a test case.

Attribute	With monitoring	Without monitoring	Overhead
Data Update	0.136 ms	0.087 ms	56.8 %
RTT	25.6 ms	23.1 ms	11.1 %

The data update time overhead was nearly 57 %, which seems to be quite high. An overhead percentage such as this would be clearly too high in production systems. But if we consider our test case, where the user of the service requests a new data value every half a second, the 0.05 millisecond addition to the data update time by the monitoring does not affect the data consistency at all. A 2.5 ms addition to the RTT was quite reasonable and can be accepted both for development and production usage. The overhead statistics show that the monitoring tool managed very well in this test case. This is partly due to the nature of the bundles monitored, if the Index Service would have been much more heap memory intensive, the overhead cause by the monitoring tool would have been much higher.

The overhead measurement provided by the monitoring tool is shown in Figures 41 and 42. The overhead is negligible at periods A, E and F. When the bundles A and B are running at periods from B to D, the overhead increases to a value that is rather high. The increase of disturbance derives from the nature of the bundles A and B. Both of the bundles utilize 1000 threads, which causes a high load for the monitoring tool since the tool uses thread based monitoring.

6.1.3 Discussion on the monitoring tool's applicability

As the results in Chapter 5. show, the monitoring tool was able to extract bundle specific resource consumption data out of the JVM. This was the primary goal and the tool fulfilled the goal very well. As was explained earlier, data is not very useful 'as is' but forms the basis for the further processing of the data. Chapter 5.2 presents the use of the monitoring tool to provide runtime statistics of the resource consumption of individual bundles. Even though the test case was very simple

and for demonstrative purposes it still clearly shows the great potential of this kind of approach. The variety of computing platforms and diverge of available computing resources that these provide is wide nowadays. Software that is capable of operating on more than one platform requires some kind of dynamic adaptation feature and the knowledge of the resources it uses.

Some disadvantages remain in the monitoring tool. The resource consumption information provided by our tool is excessively platform dependent for comparisons on a large scale. This reduces the applicability of the resource usage boundaries. We should be able to provide fully platform independent statistics, such as studied in Chapter 2.3.4, in order to validate a competitive and comparable performance attribute.

The overhead of the monitoring tool comprises of many separate causes, as was explained in the previous section. The largest problem with the disturbance caused by the tool is that the overhead depends largely on the nature of the monitored bundles. This kind of behaviour makes the overhead hard to estimate and too unpredictable for large scale exploitation. In addition, this behaviour leads to the fact that it is not feasible to use the monitoring tool in a production environment.

6.2 The visualization and analysis tool

The visualization and analysis tool was shown to efficiently utilize the monitoring data and facilitate the resource consumption analysis of bundles. The tool produced a clear view of the inners of the JVM that is easy to understand for a human analyst. Although the provided automatic analysis was simple, these support the analyst's work rather well.

The results and figures presented in this thesis work do not completely reflect the capabilities of the scenery provided by the visualization tool. As the figures are static, the possibility of free movement in the scenery is not emphasized. In addition, the ability to select different parts of the visualization to get statistics that are more detailed was missed due to static figures.

6.2.1 Discussion on the visualization models

Two novel visualizations were introduced in this thesis work. The time dependent model gave a clear view of the distribution of resource consumption between bundles. The time independent model illustrates the bundle's need for different computing resources in order to operate. Both were found to provide very useful information.

Figure 41 demonstrated all the aspects of the time dependent model. It gives a very clear view of the inner behaviour of the JVM from a resource consumption perspective. As the visualization encloses both the CPU utilization and heap memory consumption, its information value is much higher than with traditional 2D visualizations. The encapsulation makes the discovery of bottlenecks much easier.

As the results show, the visualization tool makes it possible to analyze more than just the resource consumption behaviour of different bundles. The tool makes it possible to evaluate the performance of different runtime controlling algorithms. Although the example was simple, it efficiently demonstrated the capabilities of this kind of analysis.

A time independent visualization model was used in order to detect and validate resource usage boundaries for a bundle. It also provided a great possibility for comparison between the different configurations of the monitored bundle. The scenery makes it clear to a human analyst how the switching of the operation mode affects to the resource consumption of the bundle. This visualization model could be further developed to support the design of component compositions. It should provide a possibility to define the available resources of the platform that would form the outer boundaries. The resource usage blocks of the bundles could then be stacked inside the boundaries and this composition of blocks must not violate the boundaries in order to operate on the defined platform.

6.3 Discussion on the tools

The use of the tools of this thesis together as a tool chain enables a much easier analysis of the OSGi bundles than traditional tools. Let us consider the test case presented in Chapter 5.2 and the use of the JConsole [8] as the monitoring tool. Figure 43 presents the CPU utilization statistics provided by the JConsole.

Figure 43. CPU utilization statistics provided by the JConsole.

This figure shows the changes of the CPU utilization in respect to time caused by the OSGi environment. This clearly shows how useless the resource consumption analysis of individual OSGi bundles is with current tools. Without pre-existing bundle resource usage knowledge, it is impossible to know how the consumption of resources is distributed between bundles inside the OSGi environment. Figure 43 is comparable with the previously presented Figure 41. There is an unambiguous distinction with the informative level of these figures. The figure provided by the JConsole does not reveal anything about the resource usage of separate bundles, whereas the illustration provided by our tools makes it clear which bundles are using resources and to what extent.

The use of our tools reveal the inner behaviour of the JVM from a resource consumption point of view. It disperses the uncertainty of the resource usage in the OSGi environment, where there can be multiple software components executed in the JVM. If the JVM appears to be using excessive amounts of physical resources, a clear view of situation can be obtained with the aid of our tools.

It was shown that the tools could be used to detect the resource consumption boundaries of an OSGi bundle. Validated resource usage boundaries can be used at least in three processes in the life cycle of a software component. These processes are testing, integration and marketing. Firstly, in the testing phase, boundaries can be used to check that the component meets the resource requirements set at the specification phase. Secondly, in the integration phase, boundaries can be used as a performance or quality attribute of a software component and these add a concrete comparison attribute to component integrators. Integrators can now compare a component's resource usage with other components, which offer the same service. In addition, the boundary information can be used in estimating a component's suitability to a certain platform. Thirdly, validated boundaries can be used as a tempting quality property of a component when the components are sold to potential users. The assurance of a component's resource usage could be a decisive factor in the decision-making process of a software buyer. However, as was already said, the boundaries detected by our tools are still too dependent on a test platform for usage in large scale comparison or universal assurance processes.

As the results show, the tools act as enablers for the resource consumption analysis of OSGi-based software components. Under no circumstances, are the tools ready for large scale usage and there are still lots of issues to be solved. However, the work done in this thesis provides a good foundation for further research on the resource monitoring of OSGi bundles, the visualization of component resource consumption and dynamically adaptive software components. In addition, this thesis tries to highlight the resource consumption as a comparable performance attribute of a software component in the component integration phase.

There has been one scientific conference paper submitted based on the monitoring tool introduced in this thesis work. Another paper, based on the OSGi-based resource monitoring service that enables dynamically adaptive software components, is currently under work. In addition, there is an undergoing patent investigation based on the visualization and analysis tool introduced in this work.

7. Conclusions

This thesis work presented two tools for the software performance analysis of OSGi bundles from a resource consumption perspective. These, combined with the OSGi-based resource monitoring service, provide the basis for controlling the resource consumption of bundles and enable the analysis of the control effectiveness. All the presented tools were applied to simple test cases, which demonstrated the applicability of the tools. The following briefly presents the main contributions of this work:

- Two novel 3D visualization methods for representing the resource consumption behaviour of both single bundle and multiple bundles in an easy to understand form.
- A monitoring tool that extracts bundle specific resource usage data by monitoring the JVM.
- An OSGi-based resource monitoring service that provides resource consumption information for runtime control decisions.

Test cases proved that the monitoring tool provides detailed resource consumption data on individual OSGi bundles. The 3D visualizations effectively illustrate the CPU utilization, the heap memory consumption and the use of a network of separate bundles in one single view. These enable the human analyst to gain knowledge on the component's behaviour inside the OSGi environment from a resource consumption perspective. In addition, the resource usage boundaries can be detected and validated.

The tools provide a foundation for future research on the component's quality assurance process. The validated resource boundaries provide the quality or performance attribute of a software component. This validated attribute can be used in various processes during the component's life cycle, such as, testing, integration and marketing.

The OSGi-based resource monitoring service that provides the runtime resource usage information forms the basis for the use of runtime resource consumption

controlling algorithms. The possibility for dynamic adaptation, according to available computing resources, enables the software component to be executed on different platforms without modifications. Although the test case was for demonstrative purposes, it gave encouraging results for future research on resource aware software development.

There has been one scientific conference paper submitted based on the monitoring tool introduced in this thesis work. Another paper, based on the OSGi-based resource monitoring service that enables dynamically adaptive software components, is currently under work. In addition, there is an undergoing patent investigation based on the visualization and analysis tool introduced in this work.

Overall, the tools were demonstrated to enable an effective resource consumption analysis of OSGi bundles. It was also shown that the tools could be used as a foundation for a more complete analysis of the collaborating component compositions.

References

[1] Osterweil, Leon J. A future for software engineering? In: *Future of Software Engineering (FOSE'07)*, 2007. 19–21 March 2007.

[2] Issarny, Valerie, Caporuscio, Mauro and Georgantas, Nikolaos. A perspective on the future of middleware-based software engineering. In: *Future of Software Engineering (FOSE'07)*, 2007. 19–21 March 2007.

[3] Bertolino, Antonia. Software testing research: Achievements, challenges, dreams. In: Future of *Software Engineering (FOSE'07)*, 2007. 19–21 March 2007.

[4] Brereton, Pearl, Budgen, David, Bennnett, Keith, Munro, Malcolm, Layzell, Paul, MaCaulay, Linda, Griffiths, David and Stannett, Charles. The future of software. *Communications of the ACM,* 42(12):78–84, 1999.

[5] Jaffar ur Rehman, Muhammad, Jabeen, Fakhra, Bertolino, Antonia and Polini, Andrea. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability,* 17(2):95–133, 2007.

[6] The OSGi Alliance. Osgi. Website, Feb 2007. [Retrieved 27.8.2007] From: http://www.osgi.org/.

[7] Kapfhammer, Gregory M., Soffa, Mary Lou and Mosse, Daniel. Testing in resource constrained execution environments. In: *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 418–422, New York, NY, USA, 2005. ACM Press.

[8] Chung, Mandy. Using jconsole to monitor applications. Website, Oct 2004. [Retrieved 2.7.2007] From: http://java.sun.com/developer/technicalArticles/ J2SE/jconsole.html.

[9] The NetBeans community. The netbeans profiler. Website, Aug 2007. [Retrieved 27.8.2007] From: http://profiler.netbeans.org/.

[10] Van Den Bossche, B., Van Boxstael, K., Goeminne, N., Gielen, F. and Demeester, P. M. An osgi compatible implementation of a java resource monitor. In: *Multimedia on Mobile Devices, Proceedings of the Society of Photo-Optical Instrumentation Engineers (SPIE)*, volume 56840, pages 181–189, March 2005.

[11] Brereton, Pearl and Budgen, David. Component-based systems: A classification of issues. *Computer,* 33(11):54–62, 2000.

[12] Stuckenholz, Alexander. Component evolution and versioning state of the art. *SIGSOFT Softw. Eng. Notes,* 30(1):7, 2005.

[13] Szyperski, Clemens. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing, Boston, MA, 2002.

[14] Chen, Shiping, Liu, Yan, Gorton, Ian and Liu, Anna. Performance prediction of component-based applications. *J. Syst. Softw.,* 74(1):35–43, 2005.

[15] Grassi, Vincenzo and Mirandola, Raffaela. Towards automatic compositional performance analysis of component-based systems. *SIGSOFT Softw. Eng. Notes*, 29(1):59–63, 2004.

[16] Bertoa, Manuel and Vallecillo, Antonio. Quality attributes for cots components. In QAOOSE 2002: Proceedings of the 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002), 2002.

[17] Alexandre Alvaro, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. Software component certification: A survey. In: *EUROMICRO '05: Proceedings of the 31st EUROMICRO Conference on Software Engineering*

and Advanced Applications, pages 106–113, Washington, DC, USA, 2005. IEEE Computer Society.

[18] Alvaro, Alexandre, Santana de Almeida, Eduardo and de Lemos Meira, Silvio Romero. A component quality assurance process. In: *SOQUA '07: Fourth international workshop on Software quality assurance,* pages 94–101, New York, NY, USA, 2007. ACM.

[19] The OSGi Alliance. About the osgi service platform, technical whitepaper. Website, Nov 2005. [Retrieved 2.7.2007] From: http://www.osgi.org/documents/collateral/OSGiTechnicalWhitePaper.pdf.

[20] The OSGi Alliance. Osgi service platform release 4, core specification. Website, July 2006. [Retrieved 2.7.2007] From: http://www.osgi.org/.

[21] The OSGi Alliance. Osgi service platform release 4, service compendium. Website, July 2006. [Retrieved 2.7.2007] From: http://www.osgi.org/.

[22] The OSGi Alliance. Osgi service platform release 4, mobile specification. Website, July 2006. [Retrieved 2.7.2007] From: http://www.osgi.org/.

[23] The Apache Software Foundation. Apache felix. Website, Aug 2007. [Retrieved 8.8.2007] From: http://felix.apache.org.

[24] Makewave AB. Knoplerfish. Website, Aug 2007. [Retrieved 8.8.2007] From: http://www.knopflerfish.org.

[25] The Eclipse Foundation. Eclipse equinox. Website, Aug 2007. [Retrieved 8.8.2007] From: http://www.eclipse.org/equinox.

[26] Makewave AB. Knoplerfish pro. Website, Aug 2007. [Retrieved 8.8.2007] From: http://www.makewave.com/site.en/products/knopflerfish_pro_osgi.shtml.

[27] ProSyst Software GmbH. mbedded server. Website, Aug 2007. [Retrieved 8.8.2007] From: http://www.prosyst.com/products/osgi.html.

[28] Ryan, Caspar and Rossi, Pablo. Software, performance and resource utilisation metrics for context-aware mobile applications. In: METRICS '05: Proceedings *of the 11th IEEE International Software Metrics Symposium (METRICS' 05)*, page 12, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Weyuker, Elaine J. and Vokolos, Filippos I. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Transactions on Software Engineering,* 26(12):1147–1156, 2000.

[30] Binder, Walter and Hulaas, Jarle. A portable cpu-management framework for java. *IEEE Internet Computing*, 8(5):74–83, 2004.

[31] Braddock, Robert L., Claunch, Michael R. and Rainbolt, J. Walter. Operational performance metrics in a distributed system. Part ii. Metrics and interpretation. In: *SAC '92: Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing,* pages 873–882, New York, NY, USA, 1992. ACM Press.

[32] Chen, G., Kandemir, M., Vijaykrishnan, N. Irwin, M. J., Mathiske, B. and Wolczko, M. Heap compression for memory-constrained java environments. In: *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 282–301, New York, NY, USA, 2003. ACM Press.

[33] Raeder Clausen, Lars, Pagh Schultz, Ulrik, Consel, Charles and Muller, Gilles. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):471–489, 2000.

[34] Kim, Jin-Soo and Hsu, Yarsun. Memory system behavior of java programs: methodology and analysis. In: *SIGMETRICS '00: Proceedings of the 2000*

ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pages 264–274, New York, NY, USA, 2000. ACM Press.

[35] Cahoon, B. and McKinley, K. Tolerating latency by prefetching java objects, October 1999. In: Workshop on Hardware Support for Objects and Microarchitectures for Java.

[36] Vijaykrishnan, N., Kandemir, M., Irwin, M. J., Kim, H. S. and Ye, W. Energydriven integrated hardware-software optimizations using simplepower. In: *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture,* pages 95–106, New York, NY, USA, 2000. ACM Press.

[37] Catthoor, F., Franssen, F., Wuytack, S., Nachtergaele, L. and De Man, H. Global communication and memory optimizing transformations for lowpower signal processing systems. In: *VLSI Signal Processing Workshop,* pages 178–187, Oct 1994.

[38] Lindholm, Tim and Yellin, Frank. *The JavaTM Virtual Machine Specification*, Second edition. Addison-Wesley, 1999.

[39] Hartikainen, Vesa-Matti, Liimatainen, Pasi P. and Mikkonen, Tommi. On mobile java memory consumption. In: *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06),* pages 333–339, Washington, DC, USA, 2006. IEEE Computer Society.

[40] Choi, Yoonseo and Han, Hwansoo. Protected heap sharing for memory-constrained java environments. In: *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems,* pages 212–222, New York, NY, USA, 2006. ACM Press.

[41] Smith, Connie U. *Performance Engineering of Software Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[42] Metz, Edu, Lencevicius, Raimondas and Gonzalez, Teofilo F. Performance data collection using a hybrid approach. I: *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering,* pages 126–135, New York, NY, USA, 2005. ACM Press.

[43] Mansouri-Samani, Masoud and Sloman, Morris. A configurable event service for distributed systems. In: *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, page 210, Washington, DC, USA, 1996. IEEE Computer Society.

[44] Sun Microsystems Inc. Java management extensions (JMX) specification, version 1.4. Website, Nov 2006. [Retrieved 17.7.2007] From: http://java.sun.com/ javase/6/docs/technotes/guides/ jmx/JMX_1_4_specification.pdf.

[45] Sun Microsystems Inc. Jvm ti reference. Website, Aug 2006. [Retrieved 2.7.2007] From: http://java.sun.com/javase/6/docs/platform/ jvmti/jvmti.html.

[46] Whaley, John. A portable sampling-based profiler for java virtual machines. In: *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 78–87, New York, NY, USA, 2000. ACM Press.

[47] Czajkowski, Grzegorz and von Eicken, Thorsten. Jres: a resource accounting interface for java. In: *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 21–35, New York, NY, USA, 1998. ACM Press.

[48] Binder, Walter. Portable profiling of memory allocation in java. In: *NODe/GSEM*, pages 110–128, 2005.

[49] Binder, Walter. Portable and accurate sampling profiling for java. *Software–Practice and Experience,* 36(6):615–650, 2006.

[50] Binder, Walter, Hulaas, Jarle and Villazón, Alex. Portable resource control in java. In: *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications,* pages 139–155, New York, NY, USA, 2001. ACM Press.

[51] Hulaas, Jarle and Binder, Walter. Program transformations for portable cpu accounting and control in java. In: *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation,* pages 169–177, New York, NY, USA, 2004. ACM Press.

[52] Reiss, Steven P. Visualizing java in action. In: *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 57–65, New York, NY, USA, 2003. ACM Press.

[53] Reiss, Steven P. and Renieris, Manos. Jove: java as it happens. In: *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization,* pages 115–124, New York, NY, USA, 2005. ACM Press.

[54] De Pauw, Wim, Jensen, Erik, Mitchell, Nick, Sevitsky, Gary, Vlissides, John M. and Yang, Jeaha. Visualizing the execution of java programs. In: *Revised Lectures on Software Visualization, International Seminar,* pages 151–162, London, UK, 2002. Springer-Verlag.

[55] Back, Godmar and Hsieh, Wilson C. The kaffeos java runtime system. *ACM Transactions on Programming Languages and Systems,* 27(4):583–630, 2005.

[56] Yrjönen, Anton. Performance analysis of software run-time behaviour using 3-d visualization. Master's thesis, Lappeenranta University of Technology, Department of Information Technology, Lappeenranta, 2007.

[57] Pakkala, Daniel, Pääkkönen, Pekka and Sihvonen, Markus. A generic communication middleware architecture for distributed application and service messaging. In: *ICAS/ICNS*, page 22. IEEE Computer Society.

[58] Pakkala, Daniel, Koivukoski, Aki and Latvakoski, Juhani. Midgate: Middleware platform for service gateway based distributed systems. In: *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05),* pages 682–688, Washington, DC, USA, 2005. IEEE Computer Society.

[59] Pakkala, Daniel and Latvakoski, Juhani. Distributed service platformfor adaptive mobile services. *International Journal of Pervasive Computing and Communications (JPCC),* 2(2):175–187, 2006.

[60] Pakkala, Daniel, Perälä, Juho and Niemelä, Eila. A component model for adaptive middleware services and applications. In: *EUROMICRO '07: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007),* pages 21–30, Washington, DC, USA, 2007. IEEE Computer Society.

Appendix 1 Configuration file of the monitor agent

```
#                       Monitor agent configuration file
#
#                       File format:
# help                  Prints some usage info
# include=item          Classes which will be instrumented with new bytecodes
# exclude=item          Classes which won't be instrumented with new bytecodes
# cpu_interval=time     Defines the cpu polling interval in ms
# mem_interval=times    Defines the memory dump interval in times*cpu_interval [ms]
# garbage=times         Defines garbage collector interval in times*mem_interval*cpu_interval [ms]
#                       With this interval garbage collection is forced to start
# file=name             Filename for the monitor log
# monitor=bname         Defines bundle that will be monitored
#
#                       Attribute examples:
# item                  org/osgi/*
# time                  150
# times                 100
# name                  monitor.log.txt
# bname                 bundlea.jar
#
#                       If some attribute is not defined then defaults will be used
#                       Defaults:
# include               All of the packages
# exclude               None of the packages
# cpu_interval          100
# mem_interval          100
# garbage               0 (Disabled)
# file                  monitor_log.txt
# monitor               No bundles
#
garbage=100
cpu_interval=20
mem_interval=30
file=config_example.log
monitor=BundleA.jar
monitor=BundleB.jar
```

Appendix 2 Example output of the monitor agent

2
20134456761962"JVMProcessTime"2835468750"BundleB.jar"750000000"BundleA.jar"203437500000"
20134480927603"JVMProcessTime"2835781250"BundleB.jar"765625000"BundleA.jar"203453125000"
20134505273994"JVMProcessTime"2836406250"BundleB.jar"796875000"BundleA.jar"203484375000"
20134532046112"JVMProcessTime"2836718750"BundleB.jar"812500000"BundleA.jar"203500000000"
20134557025823"JVMProcessTime"2837343750"BundleB.jar"843750000"BundleA.jar"203515625000"
20134581466918"JVMProcessTime"2837968750"BundleB.jar"875000000"BundleA.jar"203546875000"
20134607152864"JVMProcessTime"2838281250"BundleB.jar"890625000"BundleA.jar"203562500000"
20134631281629"JVMProcessTime"2838906250"BundleB.jar"921875000"BundleA.jar"203593750000"
20134654702762"JVMProcessTime"2839218750"BundleB.jar"937500000"BundleA.jar"203609375000"
20134679501165"JVMProcessTime"2839843750"BundleB.jar"968750000"BundleA.jar"203625000000"
20134705833283"JVMProcessTime"2840468750"BundleB.jar"1000000000"BundleA.jar"203656250000"
20134731903356"JVMProcessTime"2840781250"BundleB.jar"1015625000"BundleA.jar"203671875000"
20134756440273"JVMProcessTime"2841406250"BundleB.jar"1046875000"BundleA.jar"203703125000"
20134779804416"JVMProcessTime"2841718750"BundleB.jar"1062500000"BundleA.jar"203718750000"
20134805061537"JVMProcessTime"2842187500"BundleB.jar"1093750000"BundleA.jar"203718750000"
20135007540343"JVMProcessTime"2845000000"BundleB.jar"1156250000"BundleA.jar"203812500000"
20135031652626"JVMProcessTime"2845156250"BundleB.jar"1171875000"BundleA.jar"203812500000";"BundleB.jar"20894392"BundleA.jar"191456
20135368922307"JVMProcessTime"2849531250"BundleB.jar"1250000000"BundleA.jar"203906250000"
20135392977040"JVMProcessTime"2850156250"BundleB.jar"1281250000"BundleA.jar"203937500000"
20135416457957"JVMProcessTime"2850468750"BundleB.jar"1296875000"BundleA.jar"203953125000"
20135441082595"JVMProcessTime"2851093750"BundleB.jar"1312500000"BundleA.jar"204000000000"
20135467199322"JVMProcessTime"2851406250"BundleB.jar"1328125000"BundleA.jar"204015625000"

Appendix 3 Example output of the visualization tool

```
Input file: Example_OSGi_resource_cosumption_model.log
Statistics timeframe: from 861615 to 1242764, length 381149, duration:
    ~190 sec

Cpu usage statistics:
Cpu utilization of the whole JVM: 0.749434 %
Bundle: BundleA Cpu utilization: 0.0484561 % and utilization compared to
    JVM: 0.064657 %
Bundle: BundleB Cpu utilization: 0.155435 % and utilization compared to
    JVM: 0.207404 %
Bundle: BundleC Cpu utilization: 0.302068 % and utilization compared to
    JVM: 0.403062 %
Bundle: BundleD Cpu utilization: 0.00674802 % and utilization compared to
    JVM: 0.00900415 %
Monitoring overhead: 0.0451477 % and overhead compared to JVM: 0.0602424 %

Memory consumption statistics:
Bundle: BundleA largest memory consumption: 31.7013 MB smallest: 0 MB and
    average: 12.7312 MB
Bundle: BundleB largest memory consumption: 17.6072 MB smallest: 6.77072
    MB and average: 11.3382 MB
Bundle: BundleC largest memory consumption: 0.00244531 MB smallest:
    0.00103125 MB and average: 0.00109417 MB
Bundle: BundleD largest memory consumption: 3.43541 MB smallest: 0.761734
    MB and average: 1.44036 MB



Input file: Example_OSGi_resource_boundary_model.log
Statistics:

Bundle:  BundleA
CPU util: 37.8049 %, boundary: 45 %, resource underspend by 7.19512 %
Mem alloc: 1.33638 MB, boundary: 5 MB, resource underspend by 3.66362 MB
Net util: 2233.4 KB/sec, boundary: 1500 KB/sec, bound exceeded by 733.398
    KB/sec

Bundle:  BundleB
CPU util: 60.7591 %, boundary: 65 %, resource underspend by 4.2409 %
Mem alloc: 10.8249 MB, boundary: 13 MB, resource underspend by 2.17512 MB
Net util: 0 KB/sec, boundary: 250 KB/sec, resource underspend by 250 KB/
    sec
```

Series title, number and
report code of publication

VTT Publications 685
VTT-PUBS-685

Author(s)
Miettinen, Tuukka

Title

Resource monitoring and visualization of OSGi-based software components

Abstract

This work introduces a novel approach for the resources consumption analysis of OSGi-based software components. OSGi Service Platform provides a component based and service-oriented Java environment that is especially emerging in environments with constrained computational resources. OSGi Service Platform enables the cooperation of multiple Java based components within a single Java Virtual Machine. Existing JVM analyzing tools typically monitor the resource consumption of the whole Java environment, which is not sufficient in an OSGi environment since the JVM conceals the resource consumption information of separate OSGi components. This emphasizes the need for monitoring solutions that are able to provide a detailed view of the resource consumption of the Java environment.

Tools implemented in this work enable the effective resource consumption analysis of individual software components executed on a OSGi platform. A monitoring tool that is able to identify the resource consuming component was developed to extract both component and environment specific data from the Java environment. An existing visualization tool was extended in order to provide an easy to understand view of the resource consumption behaviour of both single component and component compositions. Two novel visualizations were introduced to facilitate the analysis of software resource usage. The tool produces 3D visualization that simultaneously illustrates the time related CPU utilizations and memory consumptions of all desired components executed on the OSGi platform. The other novel visualization presents the amount of resources required by a component to operate normally. In addition, it enables the comparison of resource consumption information to desired usage boundaries. The OSGi-based resource monitoring service was also developed in order to provide runtime resource consumption information for components that are able to adapt their behaviour according to available computing resources.

The applicability of the tools was demonstrated with two use cases. Firstly, an OSGi component's resource usage boundaries were detected and validated. Secondly, multiple components were monitored and use of the resource monitoring service was demonstrated with an adaptive OSGi component. It was proved that implemented tools effectively reveal how the components behave inside the OSGi environment from a resource consumption perspective.

ISBN
978-951-38-7104-8 (soft back ed.)
978-951-38-7105-5 (URL: http://www.vtt.fi/publications/index.jsp)

Series title and ISSN	Project number
VTT Publications 1235-0621 (soft back ed.) 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)	21427

Date	Language	Pages
June 2008	English, Finnish abstr.	107 p. + app. 3 p.

Name of project	Commissioned by
ITEA2-CAM4Home	Tekes, VTT

Keywords	Publisher
resource consumption, resource monitoring, software visualization, performance analysis	VTT Technical Research Centre of Finland P.O. Box 1000, FI-02044 VTT, Finland Phone internat. +358 20 722 4520 Fax +358 20 722 4374

Julkaisun sarja, numero ja raporttikoodi

VTT Publications 685
VTT-PUBS-685

Tekijä(t)
Miettinen, Tuukka

Nimeke

OSGi-pohjaisten ohjelmistokomponenttien resurssien kulutuksen monitorointi ja visualisointi

Tiivistelmä

Tässä työssä luotiin uudenlainen lähestymistapa OSGi-pohjaisten ohjelmistokomponenttien laskentaresurssien käytön analysointiin. OSGi-palvelualusta tarjoaa komponenttipohjaisen ja palvelusuuntautuneen Java-alustan, joka on herättänyt kasvavaa kiinnostusta erityisesti resurssirajoitteisten tietokoneympäristöjen markkinoilla. OSGi-ohjelmistoalusta mahdollistaa useiden Java-pohjaisten ohjelmistokomponenttien yhteistoiminnan samassa Java-virtuaalikoneessa. Olemassa olevat Javan analysointityökalut tarkkailevat koko Java-ympäristön resurssien kulutusta, mikä ei ole riittävää OSGi-ympäristössä, koska virtuaalikone kätkee yksittäisten ohjelmistokomponenttien resurssienkulutuksen. Tämän vuoksi tarvitaan uusia resurssien käytön seurantaratkaisuja, joiden avulla saadaan yksityiskohtaisempi kuva Java-ympäristön resurssien käytöstä.

Tässä työssä kehitetyt työkalut mahdollistavat yksittäisten ohjelmistokomponenttien laskentaresurssien käytön tehokkaan analysoinnin. Kehitetty resurssien monitorointityökalu tarkkailee koko Java-ympäristöä ja pystyy erottelemaan laskentaresurssien käytön komponenttikohtaisesti. Olemassa olevaa visualisointityökalua laajennettiin, jotta kerätty tieto voidaan esittää helposti ymmärrettävässä muodossa. Työssä esitellään kaksi uudenlaista visualisointia, jotka helpottavat ohjelmiston resurssien käytön analysointia. Visualisointityökalu tuottaa kolmiulotteisen näkymän, joka yhtäaikaisesti esittää haluttujen OSGi-komponenttien tuottaman prosessorikuorman ja muistinkulutuksen. Toinen uusi visualisointi esittää laskentaresurssien määrän, jotka ohjelmistokomponentti vaatii toimiakseen. Tämä myös mahdollistaa komponentin resurssienkulutuksen vertailun haluttuihin käyttörajoihin. Työssä kehitettiin myös OSGi-pohjainen laskentaresurssien monitorointipalvelu, joka mahdollistaa resurssien käyttötiedon ajonaikaisen hyödyntämisen. Tämä taas mahdollistaa vapaana oleviin laskentaresursseihin mukautuvat ohjelmistokomponentit.

Työkalujen hyödyllisyys osoitettiin kahdella erilaisella käyttötapauksella. Ensimmäisessä etsittiin ja vahvistettiin erään OSGi-komponentin resurssien kulutusrajat. Toisessa tapauksessa useita komponentteja monitoroitiin ja havainnollistettiin resurssienmonitorointipalvelun käyttöä mukautuvan komponentin avulla. Näin pystyttiin osoittamaan, että kehitetyt työkalut paljastavat tehokkaasti komponenttien käyttäytymisen OSGi-ympäristössä resurssien kulutuksen näkökulmasta.

ISBN
978-951-38-7104-8 (nid.)
978-951-38-7105-5 (URL: http://www.vtt.fi/publications/index.jsp)

Avainnimeke ja ISSN	Projektinumero
VTT Publications 1235-0621 (nid.) 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)	21427

Julkaisuaika	Kieli	Sivuja
Kesäkuu 2008	Englanti, suom. tiiv.	107 s. + liitt. 3 s.

Projektin nimi	Toimeksiantaja(t)
ITEA2-CAM4Home	Tekes, VTT

Avainsanat	Julkaisija
resource consumption, resource monitoring, software visualization, performance analysis	VTT PL 1000, 02044 VTT Puh. 020 722 4520 Faksi 020 722 4374

VTT PUBLICATIONS

667 Ahlqvist, Toni, Uotila, Tuomo & Harmaakorpi, Vesa. Kohti alueellisesti juurrutettua teknologiaennakointia. Päijät-Hämeen klusteristrategiaan sovitettu ennakointiprosessi. 2007. 107 s. + liitt. 7 s.

668 Ranta-Maunus, Alpo. Strength of Finnish grown timber. 2007. 60 p. + app. 3 p.

669 Aarnisalo, Kaarina. Equipment hygiene and risk assessment measures as tools in the prevention of *Listeria monocytogenes* -contamination in food processes. 2007. 101 p. + app. 65 p.

670 Kolari, Kai. Fabrication of silicon and glass devices for microfluidic bioanalytical applications. 2007. 100 p. + app. 72 p.

671 Helaakoski, Heli. Adopting agent technology in information sharing and networking. 2007. 102 p. + app. 97 p.

672 Järnström, Helena. Reference values for building material emissions and indoor air quality in residential buildings. 2007. 73 p. + app. 63 p.

673 Alkio, Martti. Purification of pharmaceuticals and nutraceutical compounds by sub- and supercritical chromatography and extraction. 2008. 84 p. + app. 42 p.

674 Mäkelä, Tapio. Towards printed electronic devices. Large-scale processing methods for conducting polyaniline. 2008. 61 p. + app. 28 p.

675 Amundsen, Lotta K. Use of non-specific and specific interactions in the analysis of testosterone and related compounds by capillary electromigration techniques. 2008. 109 p. + app. 56 p.

677 Hanhijärvi, Antti & Kevarinmäki, Ari. Timber failure mechanisms in high-capacity dowelled connections of timber to steel. Experimental results and design. 2008. 53 p. + app. 37 p.

678 FUSION Yearbook. Association Euratom-Tekes. Annual Report 2007. Eds. by Seppo Karttunen & Markus Nora. 2008. 136 p. + app. 14 p.

679 Salusjärvi, Laura. Transcriptome and proteome analysis of xylose-metabolising *Saccharomyces cerevisiae*. 2008. 103 p. + app. 164 p.

680 Sivonen, Sanna. Domain-specific modelling language and code generator for developing repository-based Eclipse plug-ins. 2008. 89 p.

681 Kallio, Katri. Tutkimusorganisaation oppiminen kehittävän vaikuttavuusarvioinnin prosessissa. Osallistujien, johdon ja menetelmän kehittäjän käsityksiä prosessin aikaansaamasta oppimisesta. 2008. 149 s. + liitt. 8 s.

682 Kurkela, Esa, Simell, Pekka, McKeough, Paterson & Kurkela, Minna. Synteesikaasun ja puhtaan polttokaasun valmistus. 2008. 54 s. + liitt. 5 s.

683 Hostikka, Simo. Development of fire simulation models for radiative heat transfer and probabilistic risk assessment. 2008. 103 p. + app. 82 p.

685 Miettinen, Tuukka. Resource monitoring and visualization of OSGi-based software components. 2008. 107 p. + app. 3 p.

Julkaisu on saatavana	Publikationen distribueras av	This publication is available from
VTT PL 1000 02044 VTT Puh. 020 722 4520 http://www.vtt.fi	VTT PB 1000 02044 VTT Tel. 020 722 4520 http://www.vtt.fi	VTT P.O. Box 1000 FI-02044 VTT, Finland Phone internat. + 358 20 722 4520 http://www.vtt.fi

ISBN 978-951-38-7104-8 (soft back ed.)
ISSN 1235-0621 (soft back ed.)
ISBN 978-951-38-7105-5 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)