Decentralised Dynamic Code Management for OSGi

Holger Schmidt, Jon H. Yip, Franz J. Hauck
Institute of Distributed Systems
Ulm University
Germany
{holger.schmidt,franz.hauck}@uni-ulm.de

Rüdiger Kapitza
Dept. of Comp. Sciences
Informatik 4
University of Erlangen-Nürnberg
Germany
rrkapitz@cs.fau.de

ABSTRACT

Originally designed for the management of network-attached devices OSGi builds a de-facto standard to modularise all kinds of complex Java applications. It enables deployment and updating of components, which are called bundles, by supporting automatic resolution of inter-component dependencies. Despite these benefits the OSGi specification omits dedicated support for discovery, selection and loading of locally unavailable bundles. However, this is a key requirement for large distributed applications especially in dynamic and heterogeneous environments. Current solutions are server-based and provide a central bundle repository thereby representing a single point of failure. Furthermore, these approaches lack support for automatic bundle selection based on non-functional properties such as resource demand or performance.

We introduce the D^2CM infrastructure accounting these issues and enabling automatic discovery, selection and loading of bundles in a distributed system on basis of the peer-to-peer platform JXTA. By providing extended bundle descriptions, non-functional properties can be automatically evaluated for bundle selection and dependency resolution.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Design

Keywords

OSGi, Dynamic Loading of Code, Dependency Resolution, Automatic Code Selection, Non-functional Properties

MiNEMA'08, April 1, 2008 Glasgow, Scotland

1. INTRODUCTION

Originally, OSGi [1] was designed to provide a lean Java-based component system for the management of network-attached, resource-restricted devices, such as gateways and set-top boxes. Until now, OSGi emerged to a de-facto standard for modularising and managing Java-based software, such as car infotainment systems, integrated development environments (IDEs, e.g., Eclipse) and application servers (e.g., WebSphere). OSGi is successful because of its dynamic component updating and dependency resolution support. It makes few assumptions on components enabling almost all-purpose use.

Whereas the instantiation and updating of OSGi components is easy and well-supported, dynamic loading and deployment of locally unavailable components is not. This is not part of the OSGi specification. Recent implementations try to solve this issue by providing custom services that connect to a bundle repository, such as the OSGi Bundle Repository (OBR), or an update site (e.g., for Eclipse plug-ins). However, this is a very restrictive server-based approach as it relies on central services representing a single point of failure. Furthermore, such solutions lack support for dynamic selection among multiple functionally equivalent bundles on the basis of non-functional properties such as resource demand or performance. This is not an issue if a developer selects an IDE extension bundle that has only one available implementation. In contrast to this, support for automatic selection is needed to install and update bundles composing an infotainment system that runs on different hardware and that can be connected to a variety of different devices. This issue becomes even more serious if OSGi is used to implement large distributed applications in a heterogeneous environment, which is the case for our own research work on self-adaptive mobile processes [2]. There, complex tasks are dynamically distributed as applications over multiple nodes.

Accounting the lack for dynamic selection and deployment in OSGi, this paper proposes support for on-demand bundle selection and integration using the peer-to-peer platform JXTA. The proposed decentralised dynamic code management (D^2CM) platform supports the generic and automatic selection among functionally-equivalent components. In this work, we present tailored support for OSGi. Additionally, our extended discovery service for JXTA provides optimised discovery and selection.

The paper is structured as follows: Section 2 gives a brief introduction in OSGi and JXTA. The following sec-

tion shows related work and Section 4 describes our D^2CM platform with an accompanying basic example application. We conclude and present future work in Section 5.

2. FUNDAMENTALS

In this section, we present technologies that are the basis for our D^2CM service. First, we give an introduction to OSGi and then we provide information on JXTA.

2.1 OSGi

OSGi is an open, standardised and component-based service platform defined by the OSGi Alliance [1]. It allows the installation, update and uninstallation of components at runtime. Such components, which are plain Java archives (JAR) with a special manifest with metadata, are called bundle and can contain libraries or applications. Bundles are able to share functionality on the basis of Java packages that can be exported and imported. The OSGi framework manages the coordination of such bundle dependencies.

2.2 JXTA

JXTA is an open platform providing a general basis for peer-to-peer (P2P) applications [3]. Therefore, JXTA specifies protocols to implement fundamental P2P functions [4].

JXTA nodes are called *peers*. There exist different types of peers for implementing a super peer infrastructure; standard peers are called *edge peers* and super peers are called *rendezvous peers* managing a set of edge peers. Peers form *peer groups* for restricting message propagation. Each JXTA resource (e.g., peers and peer groups) is uniquely identified and represented by an *advertisement*. Advertisements are an XML metadata structure describing resources that are used for resource publication and discovery. In order to provide an abstraction from the underlying network infrastructure, JXTA introduces *pipes* as communication channels.

JXTA provides a generic module framework for supporting dynamic service integration using code loading. For this purpose, the following advertisements are introduced. A *module class advertisement* announces the existence of a module and thus provides an abstraction for the class of provided functionality. This advertisement is referenced by a *module specification advertisement* specifying different module versions following a specific protocol, which is itself referenced by a *module implementation advertisement* providing implementation-specific details such as the code location. However, support for dynamic loading of code in JXTA is insufficient, because real implementations require further conventions [5]. These lead to platform-specific implementations that are not interoperable with each other.

3. RELATED WORK

There are a number of systems that are server-based and rely on a central repository exposing a single point of failure.

Java Web Start [6] is a software deployment system using the Java Network Launching Protocol and describing the code and the requirements of a Java application in XML. This results in a self-contained application that is installed over the Internet via a special Java Web Start client. However, there is no support for non-functional properties or automatic dependency resolution as it is required for OSGi.

In previous work [7] we investigated dynamic loading of code in context of CORBA by proposing the Dynamic Loading Service (DLS). The DLS allows applications to request locally unavailable functionality by just passing the fully-qualified name of an IDL interface. The local DLS instance requests information from a remote repository that preselects suitable implementations. Finally, the best fitting implementation of the demanded functionality is loaded and instantiated. In contrary to the approach proposed in this paper the DLS uses a classic client-server model and does not provide support for automatic implementation selection based on non-functional properties. These cases have to be manually resolved by an application-specific handler.

Paal et al. [8, 9] developed a proprietary code loading infrastructure on basis of multiple application repositories. These are queried at runtime by a custom application loader. In contrast to our D^2CM platform, their infrastructure does not provide compatibility to OSGi, is not able to consider non-functional properties and application repositories have to be preconfigured before runtime.

The OSGi Bundle Repository (OBR) offers a central repository for OSGi bundles. This is a straightforward solution that is neither scalable nor fault-tolerant. Furthermore, it misses support for automatic bundle implementation selection based on non-functional properties.

Frenot et al. [10] partially solved these issues by distributing the OBR in a peer-to-peer network. However, they only support selection and loading of bundles based on Java package dependencies but omit support for the selection based on service dependencies with non-functional properties.

The issues of a client-server solution, i.e. fault-tolerance and scalability, have also been solved by our JXTA based code loading service [5, 11]. However, in addition to this work, the proposed system provides support for automatic code selection on basis of non-functional properties and a special JXTA discovery service that is optimised for searching and selecting suitable implementations. Additionally, D^2CM provides means for automatic dependency resolution and is integrated into the OSGi framework.

A JXTA-based infrastructure for remote loading of Java classes is presented in [12]. The approach is implemented as an alternative to the standard Java class loader mechanism. However, in contrast to our approach it does not provide means to automatically describe, search and select code fulfilling particular non-functional properties.

R-OSGi [13] enables addressing remote OSGi services in a transparent way comparable to Java RMI. Instead of loading code on demand into the local OSGi framework, remote services are used. Thus, the support of R-OSGi is orthogonal to D^2CM and both frameworks can be used in conjunction.

4. DECENTRALISED DYNAMIC CODE MANAGEMENT PLATFORM

In this section we present the design of our D^2CM platform for OSGi using JXTA as underlying P2P technology. As a basic accompanying example application we introduce a printer driver service for OSGi-enabled devices. Whenever a printer is connected to such a device, an appropriate printer driver bundle can be automatically searched, selected, loaded and finally installed with our D^2CM platform.

4.1 Requirements

OSGi bundles represent the loadable units within D^2CM. For being manageable these have to be describable (see Figure 1). *Interfaces* define the component's functionality and

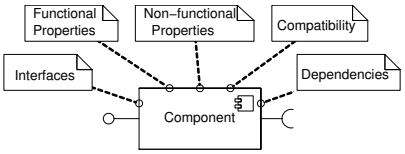

Figure 1: Description of OSGi bundles

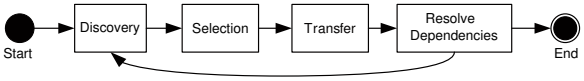

Figure 2: OSGi bundle discovery and loading

can be represented by a platform- and language-independent description, such as IDL or WSDL. For OSGi, it is sufficient to use plain Java interfaces for that purpose. *Compatibility* information describes the runtime environment like the needed platform, operating system and programming language. *Functional properties* represent actual component functionality (e.g., printer driver bundle providing colour printouts), while *non-functional properties* describe qualitative aspects such as performance and resource demand (e.g., fully-fledged or restricted printer driver bundle). Components may have *dependencies* on other components.

D^2CM should provide automatic dynamic bundle loading. For this purpose, our platform has to provide several functions (see Figure 2). Bundles should be automatically *discovered* according to our bundle descriptions. Then, appropriate bundles should be automatically *selected* and *transferred* to the local machine. Last, *dependencies* on other bundles should be automatically resolved.

In order to avoid the shortcomings of server-based systems, D^2CM should rely on a P2P infrastructure potentially providing bundle code replication. Such an approach allows each peer to publish bundles and additionally enables easy deployment, especially in mobile and ad-hoc networks.

In general, the dynamic loading support of D^2CM should be generic and portable. Thus, it should work for any standard component system but in this work we focus on OSGi.

4.2 Architecture

This section describes the D^2CM architecture implementing the specified requirements (see Figure 3).

4.2.1 JXTA Services

D^2CM provides special JXTA Services for efficient bundle discovery. A *code sharing service* supports loading and sharing resources, while a *code discovery service* supports discovery and publishing resources.

To support these services, bundle resources are published with three specific advertisements in a JXTA *CodePeer-Group* (see Figure 4). These advertisements have a globally unique identifier and therefore can reference each other. A *resource advertisement* (RA) represents an arbitrary resource such as the code of a printer driver bundle. It contains metadata on loading the resource as well as implementation properties, such as size, filename and checksum. The RA references a *code description advertisement* (CDA) identifying an implementation of an interface for a specific programming language[1]. It allows the specification of non-functional properties of the implementation, such as the needed platform and implementation's resource demand (e.g., restricted printer driver bundle for Linux with low resource demand), and references an *interface description advertisement* (IDA), which is used to announce only the existence of an interface (e.g., printer driver interface). The interface description is not part of the IDA; it is rather part of a RA that is bound to the IDA. The IDA can be searched using its fully-qualified interface name or based on keywords.

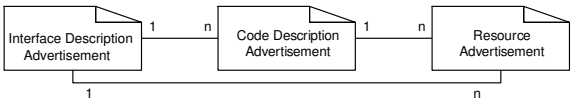

Figure 3: D^2CM architecture

Figure 4: D^2CM advertisements

The code discovery service is an extension of the JXTA discovery service, which allows searching for arbitrary advertisements. The JXTA discovery service supports only simple key-value search requests for one attribute. It is not possible to search for advertisements with multiple functional and non-functional properties. This leads to higher processing load at the requesting peer as well as to higher resource usage within the network due to the fact that even advertisements not fulfilling all of the requesting peer's functional requirements are returned and have to be sorted out. Thus, we provide an extended discovery mechanism considering multiple properties, which already sorts out incompatible advertisements on the resource-providing peers.

The code sharing service is responsible for resource loading as well as for resource provisioning. For resource provisioning the service automatically creates the corresponding RA with a concrete loading address, which is published using the code discovery service. Peers pass discovered RAs to the code sharing service for loading the required bundle code. The service supports dynamic selection of the transfer method by providing a diverse set of transfer handlers (e.g., HTTP-based), which can be loaded on demand.

4.2.2 Bundle Definition

Beside the standard OSGi bundle manifest (see Section 2.1), we added manifests to describe OSGi bundles with respect to our D^2CM platform. These manifests provide a basis for automatic D^2CM advertisement generation (i.e., IDA and CDA) and bundle dependency specification.

An *interface description manifest* describes the interface and is used for IDA generation and a *code description manifest* contains information for CDA generation. Figure 5 shows an exemplary CDA that describes a printer service bundle for Linux version greater equal 2.4 and smaller 2.6 providing duplex colour printouts. For specifying dependen-

[1]OSGi uses only Java, but our service design is generic and thus can be transferred to other platforms as well.

```
1  Name: de.uulm.impl.printer
2  Version: 1.3
3  Interface : de.uulm.Printer
4  InterfaceVersion : 1.0
5  InternalName: uulmp_1_3
6
7  compatibility {
8    os.name::linux
9    os.version:version :[2.4;2.6)
10 }
11 properties {
12   mode::duplex
13   color:boolean:true
14 }
```

Figure 5: Exemplary code description manifest

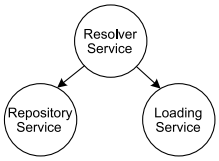

Figure 6: *D²CM* OSGi services

cies on other bundles we introduce a *resolve descriptor* (see Figure 9 and Figure 10). Dependencies can be described either on basis of IDAs or CDAs. While an IDA dependency defines a dependency to an interface, a CDA dependency describes a dependency to a specific interface implementation. For this purpose the resolve descriptor specifies the requirements to the CDA and IDA, respectively.

4.2.3 Platform Services

For providing our *D²CM* platform for OSGi, we developed three OSGi services (see Figure 6). The *loading service* provides automatic code selection and loading for the OSGi platform and the *repository service* unloads the loading service by managing already loaded and locally available bundles. The *resolver service* builds up on these services and is responsible for automatic resolution of bundle dependencies.

The loading service expects an interface name and non-functional properties as input. These properties specify the requirements of the bundle to be loaded. We differentiate mandatory (e.g., system requirements) and optional properties (e.g., performance and resource demand). While mandatory properties have to be accounted for selection, optional properties build the basis for evaluation in order to finally obtain a bundle ranking. Figure 7 shows an exemplary loading process for a printer bundle running in Java and Windows with optional colour and duplex printouts. Only bundle descriptions containing the mandatory requirements are searched for (1) and finally returned to the requesting peer (2, 3). For evaluation, optional properties have a quantifier (defined by the bundle developer). If a bundle description contains an optional property it gets a score according to the quantifier. Scores are added if further optional properties are met (4). All appropriate bundles are ranked and the best-ranked advertisement is selected for loading (5). Bundle discovery is realised by searching for advertisements with the code discovery service (see Section 4.2.1). An issue with automatic bundle selection is to determine the duration of the time period the loading service waits for incoming advertisements. This is highly application- and environment-dependent. Thus, we allow the specification of a timeout as well as a threshold for incoming advertisements after which the selection process can start. Last, code is transferred using the selected bundle's RA metadata.

```
1  Bundle−SymbolicName: de.uulm.impl.printer
2  Bundle−Version: 1.0.2
3  Bundle−Name: printer driver bundle
4  Bundle−Activator: de.uulm.impl.printer.Activatr
5  Require−Bundle: org.eclipse.swt
6  Import−Package: org.eclipse.swt,org. eclipse .swt
       .events
```

Figure 8: Bundle manifest with SWT dependency

```
1  byCDA:org.eclipse.swt: {
2    compatibility {
3      m:lang.name::java
4    }
5    properties {
6      o:50: native.lang :: german
7    }
8  }
```

Figure 9: Resolve descriptor with CDA dependency

The repository service is a local service managing already loaded and locally available bundles. This allows applications to first search for bundles at the repository service in order to save time and network resources. Then, the loading service is used for searching for remote bundles using JXTA.

The resolver service uses the loading service and the repository service to resolve bundle dependencies. Therefore, it reads the dependencies from the bundle manifest headers `Require-Bundle` and `Import-Service`. A specific `Require-Bundle` header results in searching for a specific implementation described by a CDA (see Figure 8 for a bundle manifest describing a dependency on an SWT bundle). The requirements for the selection process are described within a specific *resolve descriptor*, which contains the implementation name (prefixed by `byCDA`) as well as mandatory and optional properties (see Figure 9). A dependent service is specified in the bundle manifest's `Import-Service` header, which maps to the service interface name in the resolve descriptor's `byIDA` section (see Figure 10). On basis of the given properties in the resolve descriptor, first the resolver service is queried for appropriate local bundles. If none are found, the loading service is able to automatically search for the best-fitting bundles according to the resolve descriptor's requirements. In case of a `byCDA` description, the loading service starts with searching for CDAs with an appropriate implementation name, in case of a `byIDA` description, IDAs with an appropriate interface name are searched.

4.3 Integration into OSGi Console

OSGi framework implementations such as Apache Felix [14] and Eclipse Equinox [15] provide an OSGi management console (e.g., for bundle installation, starting and stopping). For a seamless integration of our *D²CM* platform we integrated it into the OSGi console of Equinox and Felix. In order to automatically resolve an installed bundle's dependencies the command `dcm_resolve <bundle_id>` is used (`bundle_id` is the identifier of the bundle to be resolved). Internally, our resolver service is used for bundle dependency

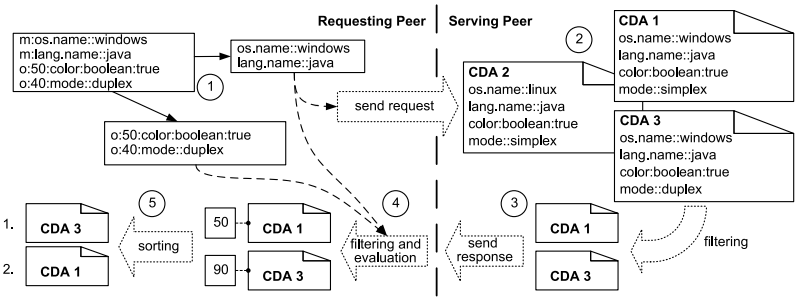

Figure 7: *D²CM* loading process

```
1  byIDA:de.uulm.printer: {
2    compatibility {
3      m:os.name::windows
4      m:lang.name::java
5    }
6    properties {
7      o:50: color : boolean:true
8      o:40: mode::duplex
9    }
10 }
```

Figure 10: Resolve descriptor with IDA dependency

resolution (see Section 4.2.3). For automatically resolving bundle dependencies during installation `dcm_install <url>` is used (`url` specifies the bundle's location).

5. CONCLUSION AND FUTURE WORK

In this paper we presented our novel *D²CM* infrastructure for decentralised dynamic management of code for OSGi. Our platform provides means for describing and publishing bundle code with JXTA, which avoids the shortcomings of server-based systems. It allows automatic discovery, selection and loading of platform-specific code on demand. In contrast to related work, functional as well as non-functional properties are considered during the discovery and selection process with a bundle ranking based on quantifiers. *D²CM* allows automatic resolution of bundle dependencies based on functional and non-functional properties. A basic example application of a printer driver service for OSGi-enabled devices shows the general feasibility of our approach. Our *D²CM* platform is compatible to OSGi R4 frameworks.

At the moment, our platform does not allow the publication of more than one interface per bundle. However, as bundles may provide more than one interface, we plan an extension of our bundle publication mechanism. Additionally, we would like to enhance the evaluation mechanism of our platform. Currently, *D²CM* only supports the evaluation of each non-functional property according to Boolean operators (see Figure 7), but some non-functional properties such as performance may require stepless quantification. Moreover, we will evaluate the OSGi built-in security mechanisms with respect to suitability in our *D²CM* platform.

We are investigating more complex scenarios for our *D²CM* platform. For instance, we would like to support our infrastructure for adaptive Web service migration [2]. This would allow having node-tailored Web service containers that can be loaded on demand. Even parts of the Web service containers, such as the Web server and the SOAP engine can be provided as bundles and configured at runtime. We plan an extensive evaluation of such scenarios.

6. REFERENCES

[1] OSGi Alliance. OSGi service platform: Core specification, release 4. Technical report, 2005.

[2] H. Schmidt and F. J. Hauck. SAMProc: Middleware for Self-adaptive Mobile Processes in Heterogeneous Ubiquitous Environments. In *MDS '07*. ACM Press, 2007. Accepted for publication.

[3] L. Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5(3), 2001.

[4] JXTA Project. JXTA v2.0 protocols specification. Technical report, Sun Microsystems, 2001.

[5] R. Kapitza, H. Schmidt, U. Bartlang, and F. J. Hauck. A Generic Infrastructure for Decentralised Dynamic Loading of Platform-Specific Code. In *DAIS '07*, 2007.

[6] Inc. Sun Microsystems. Java Web Start overview. White paper, Sun Microsystems Inc., 2005.

[7] R. Kapitza and F.J. Hauck. DLS: a CORBA service for dynamic loading of code. In *OTM '03*, 2003.

[8] S. Paal, R. Kammüller, and B. Freisleben. Dynamic software deployment with distributed application repositories. In *KiVS '05*. Springer, 2005.

[9] S. Paal, R. Kammüller, and B. Freisleben. Self-managing application composition for cross-platform operating environments. In *ICAS '06*. IEEE, 2006.

[10] S. Frenot and Y. Royon. Component deployment using a peer-to-peer overlay. In *Component Deployment*, volume 3798 of *LNCS*, 2005.

[11] R. Kapitza, U. Bartlang, H. Schmidt, and F. J. Hauck. Dynamic Integration of Peer-to-Peer Services into a CORBA-Compliant Middleware. In *OTM '06 Workshops*, volume 4277 of *LNCS*, pages 28–29, 2006.

[12] D. Parker and D. Cleary. A P2P approach to classloading in Java. In *AP2PC '03*, 2003.

[13] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed Applications Through Software Modularization. In *Middleware '07*, volume 4834 of *LNCS*. Springer, 2007.

[14] Apache Software Foundation. Apache Felix. http://felix.apache.org/, 2008.

[15] Eclipse Foundation. Equinox. http://www.eclipse.org/equinox/, 2008.