OSGi$4C$: Enabling OSGi for the Cloud

Holger Schmidt, Jan-Patrick Elsholz, Vladimir Nikolov, Franz J. Hauck
Institute of Distributed Systems
Ulm University
Germany
{holger.schmidt,jan-patrick.elsholz, vladimir.nikolov,franz.hauck}@uni-ulm.de

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

ABSTRACT

OSGi is an industry standard for a lean Java-based component system with focus on local applications following the service-oriented architecture. Initially developed for dedicated application domains such as gateways and set-top boxes it is meanwhile used in many other areas, with Eclipse as a prominent example. OSGi allows dynamic deployment of components called bundles with automatic local dependency resolution on basis of exported Java packages. In this regard, OSGi builds an ideal basis for demand-driven deployment of complex Java applications, such as needed in the context of emerging cloud computing infrastructures. However, it lacks distributed code deployment and resolving.

We present *OSGi for the Cloud* (OSGi$4C$), a novel OSGi service allowing seamless deployment of locally non-existent OSGi bundles and services on demand without requiring any changes to the OSGi platform. In OSGi$4C$, we use an underlying peer-to-peer infrastructure to provide, share and load OSGi bundles at runtime. Unlike related work, OSGi$4C$ automatically resolves OSGi bundle and service dependencies. Therefore, dependent bundles that are not yet locally installed are also deployed. OSGi$4C$ considers platform-specific implementations (e.g., native code) and non-functional requirements (e.g., performance and resource demand) while automatically discovering and selecting the best of multiple available bundles for download.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
COMSWARE'09, June 16-19, Dublin, Ireland
Copyright 2009 ACM 978-1-60558-353-2/09/06 ...$10.00.

Keywords

OSGi, Cloud Computing, Dynamic Loading of Code, Dependency Resolution, Non-functional Properties

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. In the meantime, OSGi emerged to a de-facto standard for modularising and managing all kinds of complex Java-based software, such as integrated development environments (IDEs, e.g., Eclipse) and application servers (e.g., WebSphere). OSGi enjoys wide industry acceptance as it facilitates lightweight techniques for dynamic component updates and automatic code dependency resolution during system runtime. Furthermore, it allows the seamless combination of code provided by different vendors and the parallel usage of multiple versions of the same code. Taking these facts into account, we consider OSGi as a key-technology for emerging cloud computing [2] infrastructures to support on-demand deployment of complex Java-based applications. One of the main ideas of cloud computing is the dynamic allocation and deallocation of computing resources within a service-based infrastructure. Whereas protagonists of the field (e.g., Amazon EC2 [3]) and early research initiatives (e.g., XenServer [4]) consider disk images comprising operating system, middleware and applications as a deployment unit, we argue that disk images can only provide a base infrastructure for a more fine-grained and flexible management of distributed applications. For that purpose, OSGi can be leveraged as overlying technology.

Despite all the outlined benefits and the fact that instantiation and updating of OSGi components is well-supported if the components and the code location is known, this is not the case if components have to be dynamically discovered, loaded and deployed as this is not covered by the OSGi specification. These issues were partly solved by providing a bundle repository, such as the OSGi Bundle Repository (OBR) [5] or an update site (e.g., for Eclipse plugins). However, both approaches suffer from typical issues of server-based solutions, such as single point of failure and limited scalability. A peer-to-peer (P2P) approach can solve these issues and even reduce administrative efforts as each peer could provide bundles without the need of server administration. If necessary, even dedicated server nodes can

be integrated in order to provide components for download. Current solutions for bundle provisioning assume that the user deploys bundles explicitly, and they lack support for dynamic selection among multiple functionally equivalent bundles on the basis of non-functional properties, such as resource demand and performance. This is not an issue if a developer selects an IDE extension bundle that has only one available implementation. However, support for automatic selection is needed to install bundles composing complex services that are distributed over a large set of nodes providing different resources. This issue becomes even more severe if OSGi is used to implement large distributed applications in a heterogeneous environment. For example, if client applications base upon a rich client platform and require updates of components on demand. In this case, dynamic deployment equally affects server and client systems. They might even require the same type of a service, for instance a Web server, but with different non-functional properties. While a mobile client (e.g., a smartphone) needs an implementation that provides the required functionality with a low memory footprint, a server demands for a scalable implementation but memory is not a pressing issue.

Accounting the lack of dynamic discovery, selection and deployment in OSGi, this paper makes the following three main contributions:

- Centralised and decentralised dynamic discovery of bundles and services
- Automatic bundle selection by functional and non-functional properties
- Transparent support for discovery, selection and deployment of bundles

The proposed OSGi*4C* platform supports the generic and automatic selection among functionally equivalent bundles. In addition to our recent workshop publication [6] where we outlined only preliminary results, we present an in-depth discussion of the necessity of non-functional properties and a detailed example how to handle them. Furthermore, we outline a transparent integration of OSGi*4C* into OSGi (without requiring any changes to the OSGi platform) and provide an extensive performance evaluation of our JXTA-based prototype by a comparison with the OBR and the use of SOAP-based Web services (remote invocations in contrast to dynamic loading and deployment) with our implementation.

The paper is structured as follows. Section 2 gives a brief introduction to OSGi and JXTA. Then, Section 3 sketches a heterogeneous cloud computing scenario regarding deployment. In Section 4, we outline the necessity of non-functional property support for dynamic deployment. Section 5 presents our requirements and the following section describes our OSGi*4C* platform. We evaluate the performance of OSGi*4C* in Section 7, being followed up by Section 8 discussing related work. Finally, we conclude and show possible future work in Section 9.

2. BASIC TECHNOLOGIES

In the following, we present the underlying basic technologies of OSGi*4C*.

2.1 OSGi

OSGi is an open and standardised service platform defined by the OSGi Alliance [1]. The platform provides a lean Java-based component framework, which allows the installation, update and uninstallation of components at runtime. Components are standard Java archives with a special manifest containing metadata. Such components are called *bundles* and can contain libraries and applications in terms of OSGi *services*. These services should implement a regular Java interface and are registered with this interface and optional metadata at an OSGi platform *service registry*. Bundles are able to share functionality on the basis of Java packages that can be exported and imported. The coordination of such bundle dependencies is part of the OSGi framework. Following the idea of the service-oriented architecture [7], OSGi services can be built on basis of other bundles' services.

OSGi was designed with focus on local applications, such as gateways and set-top boxes. The OSGi framework provides functions for local service discovery and instantiation. Thus, services are able to collaborate only within the OSGi system's frontiers. Automatic service dependency resolution with remote service repositories is not part of the OSGi specification, but services have to be installed locally by an administrator. Dependencies can in fact be described with the bundle manifest header `Import-Service` but the actual resolution of service dependencies is left to bundle developers even in the local case. Therefore, OSGi provides support for monitoring the availability of local services.

2.2 JXTA

JXTA is an open and generic P2P platform [8]. It specifies programming–language-independent protocols for fundamental P2P functions [9]. JXTA nodes are called *peers* and implement a super peer infrastructure: standard peers are called *edge peers* and super peers are called *rendezvous peers* managing a set of edge peers. Independent from the peer type, peers are organised in *peer groups*. All JXTA resources (e.g., peers and peer groups) are uniquely identified and represented by *advertisements*, XML metadata structures for describing resources. Advertisements are used for resource publication and discovery.

JXTA provides a *generic module framework* for supporting dynamic integration of JXTA services on basis of code loading. For this purpose, a set of advertisements is 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, 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 [10]. These lead to platform-specific implementations that are not interoperable with one another. Additionally, JXTA does not support taking non-functional requirements into account during the selection process.

3. OSGI AND THE CLOUD

Cloud computing aims at providing on-demand computing resources as a service in a scalable way. Thus, it should be possible to instantly deploy and undeploy applications within a large set of nodes. Typically, nodes do not map to physical machines but are provided as virtual machines as they can be easily managed and collocated on fewer physical hosts. Thus, the typical deployment unit for cloud computing is a disk image comprising the operating system, middleware and application. Although disk images are a convenient and robust way to deploy basic infrastructure, we think that this is not flexible enough to handle dynamic demands of large-scale Java-based applications. For instance, if certain parts of a distributed application have to be updated due to security concerns this should not lead to the redeployment of a whole disk image. The same issue applies if additional components have to be added. We advocate that a middleware on the corresponding nodes should provide services to handle these tasks and that OSGi will play a major role for Java-based applications as it facilitates updating of components and handling of code of multiple vendors, even in multiple versions. However, OSGi falls short of distributed deployment support, an essential requirement for cloud computing.

In the following, we identify requirements for such distributed deployment building the basis for a cloud computing infrastructure, which allows the management of large dynamically distributed Java applications. At server-side, we consider a standard multi-tier application and at client-side a rich client platform (RCP), such as provided by Eclipse. This facilitates deployment of complex services beyond typical Web-based solutions, interaction with local devices and storage, and off-line working. For instance, such a scenario applies for an insurance company with stationary staff and mobile insurance agents. Typically, the server-side infrastructure provides services related to account management and information about rates and contracts. Here, the stationary and mobile employees use an RCP that can be dynamically extended by plug-ins according to the current task.

Beside the fact that software deployment in such scenarios is complex and a multitude of components are required, the deployment system should be reactive to peak loads. For instance, peaks can be triggered by software updates or flash crowds requiring the allocation of more computing resources (this requires deployment at provider-side). Furthermore, heterogeneity has to be addressed as it should be possible to deploy software on server and on client machines. Client machines range from standard desktop PCs to smart phones used by mobile insurance agents. If components have a dedicated interface determining the target execution environment, the selection of an appropriate component—for the target machine—is an easy task. However, we assume multiple instances of a component with the same interface but having different non-functional properties. An example is an HTTP service component being available as a standard OSGi service [11]. At server-side, such a service is used to provide the aforementioned services with performance as key requirement. However, mobile staff might also need an HTTP service component to offer services to stationary staff, such as location information. Here, performance is of minor interest but memory footprint is a severe issue. We figured out that such a scenario can already be supported as various HTTP OSGi services are available (each having different performance and memory demand; see Section 4). The key challenge is to make these properties visible to the selection process during dynamic deployment.

4. NON-FUNCTIONAL PROPERTIES

Before deployment it is necessary to evaluate if a certain component suits the needs of an application, which should be deployed. This applies to all candidates with the required functional support. For improved resource usage a dynamic loading platform should neither require the loading nor the deployment of components for verifying if a certain component suits application needs. Instead, it should support this process by representing all kinds of non-functional properties in the same way as functional and compatibility properties. It is necessary to represent non-functional properties as values that can be compared and rated. Thus, there has to be an evaluation function (e.g., a benchmark) that provides a value for rating and comparing implementations of the same functionality. The evaluation function and the conditions for performing the evaluation have to be agreed among all implementation providers. This is a non-trivial requirement but it enables to reason about non-functional properties. In general, we assume a document outlining the procedure to measure the required non-functional properties. If for some reasons (e.g., an open system) different measurement settings were used, an ontology, for instance on the basis of OWL [12], could be used to gain a common understanding of the general evaluation conditions.

4.1 Analysis

We analysed the support for the non-functional properties *performance* and *resource demand*. Therefore, we took three different implementations of the OSGi HTTP service [11], which enables bundles to share resources, and measured their performance and resource demand. Providing information via HTTP is needed for various kinds of applications. Consequently, an HTTP bundle is needed for different general conditions, such as to provide a location service on an embedded device (see Section 3). On a server system the service can be used to serve all kinds of interactive applications that are concurrently accessed by a large number of users. In both use cases, an HTTP service has to be provided but there are different non-functional requirements for the implementations. In the first case, the service implementation should be lean and less resource consuming. In the second case, it should provide good performance and scale up to a large number of users.

For the following analysis we consider three different HTTP service bundles, each having different advantages under certain conditions and demands:

- A simple HTTP service based on NanoHTTPD[1]. NanoHTTPD comprises only a single Java class. It has a very low footprint in terms of memory and code size. The drawback of NanoHTTPD is the comparable low performance due to the lack of multi-threading support.

- Knopflerfish HTTP as an average OSGi HTTP service implementation

[1]http://elonen.iki.fi/code/nanohttpd/

- Knopflerfish HTTP extended by an additional server-side caching proxy. The idea is to offload the transfer of static files from the JVM. This improves performance but the drawback of this implementation is its dependency on native code for the caching proxy (e.g., to the Linux OS).

4.2 Analysing Performance

Like most non-functional properties, performance is complex to measure. Measurements highly depend on the configuration and on the benchmarks and their settings. A dynamic loading platform should not make any assumptions about how or what kind of values are measured to characterise the performance of a component. It should just require information to determine these values. Thus, an implementation provider should be able to gain the necessary information to describe a component in a complete way. For instance, there are approaches to model the evaluation conditions for software performance using OWL, such as proposed by Lera et al. [13]. In the context of our analysis we use the httperf tool [14] to measure the performance of the HTTP service bundles in terms of requests per second. Two machines with a 3.2GHz CPU and 1024Mb RAM running Sun JDK 1.5_09 served as evaluation platform. Table 1 shows the results. As expected, the NanoHTTPD bundle provides the slowest implementation due to its missing support for multi-threading. The standard Knopflerfish bundle is slightly faster but not as fast as the proxy-extended variant.

4.3 Analysing Resource Demand

Resource demand has various dimensions, such as memory, disk space, CPU and bandwidth. Logical resources, such as open files and sockets and even software licences can be accounted. In this work, we focused on memory demand as it is an important criterion to determine if a certain implementation is executable on a given resource-restricted device, such as a mobile phone or a PDA. We measured the memory usage of the deployed bundle. Table 1 outlines the results. The NanoHTTPD bundle needs only 19.6 kB as only 43 classes from the bundle and the Java standard library have to be loaded and instantiated. Due to providing more features, support for multi-threading and object pool support, the Knopflerfish bundle demands 33.2 kB memory. Finally, the proxy-extended bundle uses the same amount of memory inside the JVM as the Knopflerfish bundle but it needs 1580 kB additional memory for the proxy as a native process.

4.4 Cloud Scenario

Considering the aforementioned three implementations, a dynamic loading platform should be able to select a suitable implementation depending on performance requirements and resource demand (i.e., memory demand). Therefore, it is necessary to rate these non-functional properties. There should be default priorities that are defined by the developer. For instance, at server-side in the scenario of Section 3, performance is rated higher than resource consumption. This leads to an implementation being first selected on the basis of requests per second; then, resource consumption is taken into account.

[2] 33.2 kB (JVM) + 1580 kB (ext.)

[3] 126 kB (Bundle) + 29 kB (ext.)

On an average machine (desktop or notebook) the proxy-extended bundle is selected as it provides the best performance. This implementation has specific system requirements as it includes native code that is only executable on a Linux system. If the requesting system runs a different operating system the standard Knopflerfish HTTP bundle is loaded. If our platform runs on resource-limited devices, such as mobile phones and PDAs, resource demand is rated higher than performance. This leads to the NanoHTTPD implementation being loaded.

4.5 Discussion

To sum it up, support for non-functional properties is an essential part of a sophisticated dynamic deployment platform. Especially in the case that multiple functionally-equivalent bundles are available, taking non-functional requirements into account can improve system performance and allows resource-aware implementation selection.

We propose the use of a standardised procedure (e.g., a benchmark) for making certain kinds of non-functional properties (e.g., performance and resource demand) measurable and therefore accountable during implementation selection. Our outlined example assumes that all implementations are evaluated in context of the same environment setting. This is applicable for the sketched mobile insurance agent scenario (see Section 3) but might be complicated in open distributed systems, in which all kinds of companies and open source projects are able to provide implementations. Today, for instance, a web server is selected based on features and performance. In most cases, performance is either advertised by the project itself or if available by using standard benchmarks. This way, our approach integrates what could be called current-best-practice in an automated deployment process.

Additionally, one has to note that all kind of benchmarks account certain scenarios, which might not apply for the current deployment situation. For example, if memory footprint is measured under low load but the deployed component is actually under high load most of the time.

However, we assume that considering non-functional properties during deployment improves the selection process in contrast to a selection without these information. For certain corner cases the evaluation function has to be adapted and extended but our approach automatically improves the general case.

5. REQUIREMENTS

OSGi*4C* aims at providing automatic and dynamic loading of OSGi bundles and services. Therefore, it has to provide several functions (see Figure 1). OSGi resources have to be automatically *discovered* according to a given bundle description (see below). Appropriate bundles have to be automatically *selected* and *transferred* to the local OSGi environment. *Dependencies* on other OSGi resources have to be automatically resolved. If dependencies cannot be resolved a backtracking mechanism should load an alternative bundle of the previous selection if possible.

To automate the entire loading process it should run without user interaction. Thus, for dynamic loading of appropriate bundle code at runtime, bundles have to be describable with metadata characterising the OSGi bundle/service requirements and dependencies (see Figure 2). In prior work [10], we already suggested general categories of meta-

Analysed Bundle	Requests/s	Memory Demand (kB)	Code Size (kB)
NanoHTTPD	289	19.6	22
Knopflerfish HTTP	334	33.2	126
Knopflerfish HTTP with Native Proxy	399	1613.2[2]	155[3]

Table 1: Performance and memory consumption of HTTP bundles

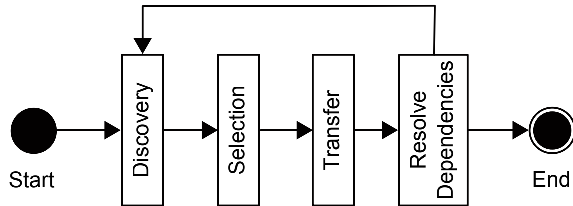

Figure 1: OSGi4C bundle discovery and loading

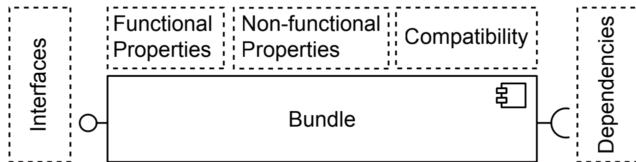

Figure 2: Description of OSGi bundles

data and properties for a generic classification of code. In this work, we have adapted our former approach for particularly describing OSGi bundles, services and their dependencies. *Interfaces* define the bundle functionality in terms of provided services. *Compatibility* information describes the runtime environment, such as the needed platform and operating system. *Functional properties* represent component functionality that is not represented by the pure interface (e.g., an HTTP bundle providing dynamic content), while *non-functional properties* describe qualitative aspects such as performance and resource demand (e.g., an HTTP bundle with or without native proxy). During the selection process (see Figure 1), functional properties usually represent mandatory criteria while non-functional properties represent optional criteria, which are necessary for obtaining a bundle ranking. Bundles can have *dependencies* on other components.

OSGi*4C* should rely on a P2P infrastructure as this overcomes the issues of server-based solutions, such as exposing a single point of failure, scalability, and administrative effort. Furthermore, in a P2P infrastructure server nodes can be integrated if appropriate. Our approach should allow each peer to publish bundle code to reduce administrative efforts. This enables convenient deployment, especially in mobile and ad-hoc networks. Conceptually, the P2P part of our OSGi*4C* platform should be exchangeable.

6. OSGi*4C* PLATFORM

In this section, we provide an overview of our OSGi*4C* architecture (see Figure 3). The architecture is divided into two main parts: P2P services building the underlying infrastructure and OSGi*4C* services providing the OSGi interface to our OSGi*4C* platform. Our concept is independent from a specific P2P mechanism in use; in this work we focus on JXTA providing a generic P2P infrastructure. As JXTA supports arbitrary topologies it allows integrating server nodes as well. In scope of our prototype we focussed on the pure P2P approach. The next section provides a more detailed overview of these JXTA services. The rest of this section will provide information on the OSGi integration of OSGi*4C* with our enhanced bundle definition.

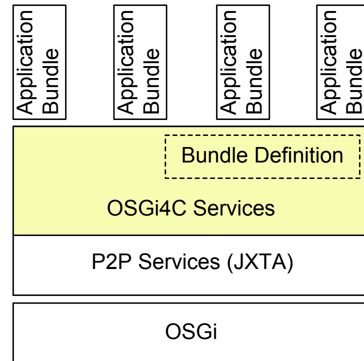

Figure 3: OSGi4C architecture

6.1 JXTA Services

A key requirement for dynamic deployment is efficient discovery. In OSGi*4C*, a *code sharing service* is responsible for loading and sharing resources, while a *code discovery service* supports discovering and publishing resource metadata.

OSGi resources are published by three advertisements in a specific JXTA *code peer group* (see Figure 4). In contrast to our generic approach [10], we propose an optimised partition of resource metadata into advertisements being more suitable for OSGi. JXTA advertisements have globally unique identifiers and are able to reference each other using these identifiers. An *interface description advertisement* (IDA) announces the existence of an interface (e.g., HTTP service interface) and can be searched using a fully-qualified interface name or based on keywords. The interface description is not part of the IDA. It is described in one or more referenced *resource advertisements* (RA), which represent an arbitrary resource. IDA-referencing *code description advertisements* (CDA) describe implementations of an interface for a specific platform with non-functional properties, such as the resource demand (e.g., an HTTP service with low resource demand). The IDA contains a section with advised criteria, which specify optional and mandatory functional and non-functional properties for describing a specific implementation (see Section 4). The CDA references resource advertisements (RA), which contain metadata for loading the resource as well as implementation properties, such as

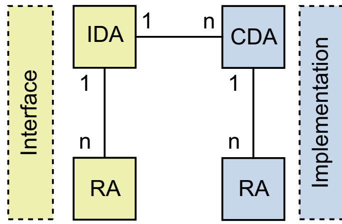

Figure 4: JXTA advertisements for describing OSGi resources in OSGi$4C$

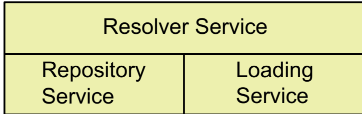

Figure 5: OSGi$4C$ services

size, filename and checksum.

JXTA provides a standard discovery service, which enables searching/publishing of arbitrary advertisements. It supports only search requests with one key-value as search criteria. Thus, it is impossible to search for advertisements with multiple functional and non-functional properties at the same time. To overcome this issue a query could be executed with only one key-value and incompatible advertisements could be filtered out on the requesting peer side. This would lead to higher processing load on the requesting side as well as higher network load. Hence, we developed a special JXTA code discovery service. Our service allows discovering advertisements with arbitrary criteria. Thus, incompatible advertisements are already sorted out at the resource-providing peers.

For loading and provisioning of OSGi resources we developed a code sharing service. Our service is able to automatically create an RA with the concrete loading address for a given resource. The service manages physical resource provisioning as well. In our prototype we implemented a simple HTTP-based resource provider; our service automatically starts a Web server serving the provided resource. Our service supports resource loading as well. Therefore, an RA has to be passed to the service. The transfer method is dynamically selected on the basis of the RA data. Specific *transfer handlers* implement the particular mechanism and can be loaded on demand. OSGi$4C$ provides a JXTA-based handler as a basis. This enables seamless data transfer across firewalls and using a Bluetooth connection for data transfer due to JXTA-provided transparency.

6.2 OSGi$4C$ Services

Our OSGi$4C$ platform consists of three OSGi services (see Figure 5). The *loading service* supports automatic selection and loading of OSGi resources and the *repository service* manages and provides already loaded and locally available resources. With the help of these services, the *resolver service* allows automatic OSGi bundle and service dependency resolution.

The loading service enables dynamic integration of OSGi resources, i.e., either required bundles or services. Figure 6 shows an exemplary selection process for an HTTP bundle with optional support for dynamic content and maximal throughput with respect to the available bundles. Our loading service expects either an interface or a bundle name as input. Additionally, functional and non-functional properties specify the required resource. There are mandatory and optional properties. Mandatory properties build the basis for the selection of compatible bundle code. Optional properties are used for an evaluation of all compatible bundles that have been found. Searching for OSGi resources is implemented with our JXTA code discovery service (see Section 6.1). An important factor is the time period the loading service waits for incoming advertisements. Due to the fact that this is highly application and environment dependent, we allow the specification of a timeout as well as a threshold for incoming advertisements after which the selection process can start. The evaluation of found resources is implemented by a comparison of the metadata of corresponding CDAs and finally results in a bundle ranking on the basis of scores. Therefore, optional properties have a quantifier that is defined by the bundle developer and represents a relative importance of the requirement (throughput has a quantifier of 40 and support for dynamic content has a quantifier of 50 in Figure 6). These quantifiers can also be overwritten by runtime parameters in order to allow user-defined discovery. We developed a set of *type handlers*, which are used for comparing particular non-functional demands with non-functional properties of a specific CDA. These properties have specific types, for instance, *string* or *version*. Thus, the corresponding type handler allows a comparison of the typed properties. Whenever a specific type handler is unavailable it can be loaded using our OSGi$4C$ infrastructure (type handlers are OSGi services). If a CDA fulfils an optional demand, the corresponding score is assigned to it. Scores are added if further optional properties are met (in Figure 6 CDA 3 is ranked highest because it supports maximal throughput and dynamic content). For an improved evaluation we also developed *min/max* type handlers, which credit the full score to the best-fitting CDA, a null-score to the worst-fitting one and a continuous value for those in between (e.g., required for the maximal throughput property). Finally, the best ranked resource is loaded using the corresponding RA.

The repository service manages already loaded and locally available resources. In order to save time and network resources applications are able to search for bundles at the local repository service first. Subsequently, the loading service can be used to search for remote bundles using JXTA.

The loading service and the repository service build the basis for providing the resolver service, which enables automatic bundle dependency resolution of resources that were loaded. Therefore, the resolver service reads dependencies from the OSGi bundle manifest headers. Service dependencies are specified using the bundle manifest `Import-Service` header. Non-functional demands are described in a *resolve descriptor*, which is part of OSGi$4C$-enabled bundles. If such a descriptor is not available non-functional properties are not considered during selection. The resolve descriptor contains the interface name (prefixed by `byIDA`) as well as mandatory and optional functional and non-functional demands (see Figure 7). These demands can be separated into sections, which implement a namespace. Optional demands are prefixed with 'o' and have a score, while mandatory demands are prefixed by 'm'. A `Require-Bundle` header describes a bundle dependency and thus results in searching

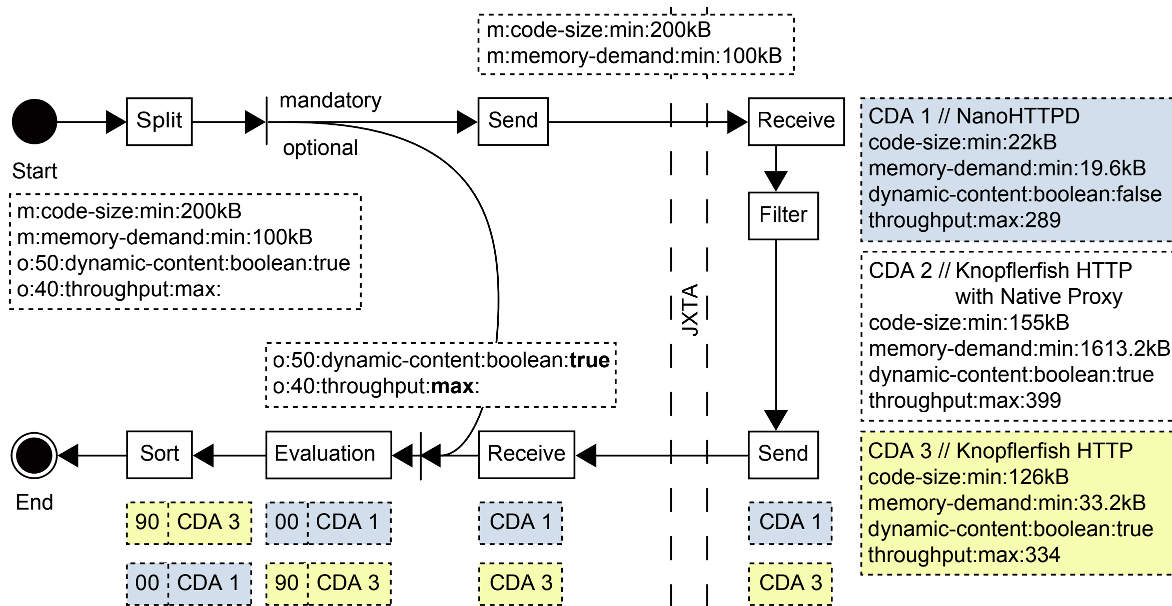

Figure 6: OSGi4C selection process

```
1  byIDA:org.osgi.service.http.HttpService: {
2    compatibility {
3      m:os.name:String:windows
4      m:lang.name:String:java
5      m:code−size:min:200kB
6      m:memory−demand:min:100kB
7    }
8    properties {
9      o:50:dynamic−content:boolean:true
10     o:40:throughput:max:
11   }
12 }
```

Figure 7: Resolve descriptor with IDA dependency

for a specific implementation described by a CDA (see Figure 8 for a bundle manifest describing a dependency on an HTTPBase bundle providing shared HTTP functionality). A specific `byCDA` section within the resolve descriptor defines functional and non-functional demands. Dependency resolution is achieved by first[4] querying the repository service for bundles being already locally available. If no appropriate bundle is found, the loading service is used to search for best-fitting bundles. In case of service dependencies the loading service starts with searching for IDAs with the required interface name, whereas for bundle dependencies the service starts with searching for CDAs with the needed bundle name.

6.3 OSGi Integration

Beside the standard bundle manifest [1], we added further manifests to describe bundles with respect to OSGi4C. These manifests allow automatic OSGi4C advertisement generation in the loading service (i.e., IDA and CDA) and bundle dependency specification. They are ignored by non-OSGi4C-capable platforms.

[4]The search method order is configurable in our prototype (see Section 6.3)

```
1  Bundle−SymbolicName:
2                 org. knopflerfish .bundle.http.HttpServiceImpl
3  Bundle−Version: 1.3
4  Bundle−Name: HTTP service bundle
5  Bundle−Activator: org.knopflerfish.bundle.http.Activator
6  Require−Bundle: org.knopflerfish.bundle.http.impl.HTTPBase
7  Import−Package: org.knopflerfish.bundle.http.impl. util ,
8                 org. knopflerfish .bundle.http.impl. tools
```

Figure 8: Bundle manifest with dependency on HTTPBase bundle

Interface description manifests describe the service interfaces within a bundle and are used for IDA generation and respective *code description manifests* contain information for CDA generation. Figure 9 shows an exemplary code description manifest that describes an HTTP bundle for Linux version equal to or greater 2.4 and less than 2.6 providing an HTTP service with support for dynamic content. As already mentioned, we introduce a *resolve descriptor* for specifying dependencies on other bundles (see Figure 7).

6.3.1 Manual start of loading process: OSGi console

Most OSGi frameworks provide a management console (e.g., for manual bundle installation and starting). For a seamless integration, we integrated OSGi4C into the OSGi console of Apache Felix [15] and Eclipse Equinox [16]. For resolving the dependencies of an installed bundle the command `OSGi4C_resolve <bundle_id>` is used (`bundle_id` identifies the bundle to be resolved). Internally, our resolver service manages dependency resolution (see Section 6.2). We implemented a basic backtracking mechanism: if dependencies of dependent bundles cannot be resolved, an alternative from the previous selection is tried. For installation with automatic resolving of bundle dependencies `OSGi4C_install <url>` is used (`url` specifies the bundle file location).

```
1   Name: org.knopflerfish.bundle.http.HttpServiceImpl
2   Version: 1.3
3   Interface: org.osgi.service.http.HttpService
4   InterfaceVersion: 1.0
5   InternalName: knopf_http_1_3
6   compatibility {
7     os.name:String:linux
8     os.version:version :[2.4;2.6)
9   }
10  properties {
11
12    code-size:min:155kB
13    memory-demand:min:1613.2kB
14    dynamic-content:boolean:true
15    throughput:max:399
16  }
```

Figure 9: Exemplary code description manifest

6.3.2 Automatic start of loading process: OSGi4C Service Tracker

For a seamless deployment of locally unavailable services we used a standard OSGi service tracker [11]. In addition to our prior workshop publication [6], this enables a completely automated process of selection, loading and installation of the best-fitting bundle and services without user interaction.

In general, a service tracker decouples OSGi bundles from directly monitoring whether required services are available at runtime. Therefore, a service tracker autonomously observes service registration and deregistration activities within the OSGi framework. This capability allows resolving required services with the service tracker. Therefore, the service tracker expects filter criteria, such as the service name and service vendor in terms of an LDAP string [1]. These filter criteria enable determining a specific implementation in case of multiple suitable ones. Some filter criteria are proposed by the OSGi specification but user-defined criteria are left open to developers. Parameters within the LDAP strings are matched against *service properties*, which can be specified at service registration. A timeout can be specified for the case that a required service is unavailable.

We extended an existing service tracker implementation to automatically deploy bundles, which implement a required service according to user-defined non-functional requirements. In general, this enables bundle developers who use a traditional service tracker to implement a blueprint of a service-based application (i.e., an application construction plan on the basis of an initial service with all required bundle and service dependencies that build-up the whole application) without caring about local availability of the needed services; node-tailored services are automatically deployed by our OSGi4C service tracker. Our OSGi-compliant OSGi4C service tracker enables a seamless integration of service-containing bundles without any OSGi framework modifications. Therefore, it uses the OSGi4C services (see Section 6.2). The preferred method order (use local bundle or OSGi4C) is specified as an ordered tuple within the LDAP string. If no such policy is specified, first locally available bundles are used. Then, if appropriate local bundles are unavailable OSGi4C is used. For seamless integration of OSGi4C within the OSGi4C service tracker, we automatically map given LDAP filter criteria specifying non-functional requirements to our OSGi4C platform for performing queries for bundles providing the needed services. Therefore, the LDAP string is parsed and non-functional requirements, which are intended for the OSGi4C platform, are automatically extracted (these are prefixed by OSGi4C). Then, an OSGi-local query is performed for determining whether an appropriate implementation for the requested service is already available. Potential service candidates are matched against the given non-functional requirements and the best-fitting one is returned. If no adequate service implementation is found, the OSGi4C service tracker uses our OSGi4C platform (see Section 6.2). Therefore, a resolve descriptor is generated for the service interface and the given non-functional requirements are inserted as mandatory and optional items. For avoiding network traffic, the repository service queries its local cache for bundles containing appropriate service implementations. If no such bundles are available the loading service is used. A suitable bundle fulfilling at least the mandatory requirements is automatically discovered, selected, downloaded, and installed. In case of any service and bundle dependencies, the resolver service attempts to resolve these dependencies with the OSGi4C services as already described. A basic backtracking mechanism is started if dependent bundles cannot be resolved (see previous section). When all dependencies are successfully resolved the bundles can be started. When starting bundles, their new services are registered and the OSGi framework propagates *service events*. Our OSGi4C service tracker listens for these events and manages references of all appropriate service candidates matching the non-functional requirements.

For preventing the OSGi framework from locking during OSGi4C network operations, the OSGi4C service tracker performs remote queries, downloading and bundle installation within a multithreaded environment. Consequently, service queries can be repeated continuously by the OSGi4C service tracker until an appropriate service implementation is available without affecting OSGi framework operation.

6.4 Security Considerations

In this section, we discuss security risks that may arise when using OSGi4C. We split these risks into network and local risks and provide solutions.

6.4.1 Network Risks

In general, JXTA is an open P2P infrastructure, in which any JXTA peer may join and leave the network. However, peers may behave maliciously and thus compromise the network. JXTA provides the concept of secure peer groups for this purpose: only peers that provide pre-negotiated credentials are able to join such a group. This establishes some kind of trust between peers [9].

JXTA messages are plain XML files. This gives peers—especially rendezvous peers—the possibility to modify these messages without notice of any other peer. A solution to this would be using standard XML signatures [17]. Additionally, peers may intercept bundle code transfer. Especially when transferring confidential data such as bundles that have to be payed, this is a serious concern. A solution would be the development of a transfer handler using transport layer security (TLS). JXTA XML messages could be secured using XML encryption [18].

6.4.2 *Local Risks*

A serious problem with dynamic code loading is malicious code. Due to the fact that OSGi is Java-based it provides standard Java security mechanisms such as sandboxing [19]. OSGi provides additional support for digital code signatures [20] and a special optional *Permission Admin* service for a fine-grained security-related framework configuration since release 4 [1].

7. EVALUATION

In this section, we present an evaluation of our OSGi*4C* prototype. We show a comparison of our system building on JXTA 2.4.1 with the pure server-based OSGi bundle repository. Additionally, for estimating the use of dynamic bundle loading in contrast to remote access, we provide performance measurements of OSGi*4C* in comparison to SOAP-based Web service invocations.

7.1 OSGi*4C* vs. OSGi Bundle Repository

We evaluated OSGi*4C* in comparison to the OSGi bundle repository (OBR) for determining the overhead of our decentralised approach in contrast to a pure server-based solution. The OBR provides a central Web server with an XML file describing bundles that can be downloaded on demand. A typical scenario of our measurement case would be the dynamic loading of an application on demand. Therefore, the application developer has to specify some kind of application construction plan in terms of bundle and service dependencies. These dependencies can be resolved either automatically using our OSGi*4C* platform or manually with the OBR.

For the evaluation we implemented an OSGi bundle, which requests a bundle at startup time. We measure the time to load, install and start the requested bundle within the local OSGi framework. For the loading process either OSGi*4C* or the OBR is used. Due to the fact that the OBR automatically loads the first available bundle implementation, we also configured OSGi*4C* to select the first-fitting bundle in order to obtain comparable results. The OBR parses an XML description of the repository at startup time and thus does not support repository changes at runtime. To allow dynamic changes at runtime, this XML file has to be parsed each time before the discovery process. Thus, we take this time into account for the OBR performance cases to make it comparable to our approach. Both systems, an OSGi*4C*-providing node as well as the OBR, are running on the same machine during the measurements. We diversified the bundle size to estimate the overhead of our JXTA-based loading infrastructure and evaluated the loading process for different network configurations. Table 2 shows the loading time of a particular bundle with the loading-client and the particular infrastructure being in the same *local network* (switched 100 Mbit LAN), in different but *inter–university*-connected networks (fast WAN connection between University of Erlangen and Ulm University) and in different *Internet-connected* networks (6 Mbit DSL WAN). We repeated the experiment 20 times and calculated the average deployment time and the standard deviation. As evaluation platform we used Apache Felix 1.0.0 running on Sun Java 1.6.0_03. An AMD Athlon 64 2.20GHz processor with 1GB RAM acted as client machine, while an Intel XEON 2.40GHz processor with 2GB RAM acted as server machine.

For the performance evaluation in the local network, the OBR outperforms our OSGi*4C* platform. However, while for a 0.1 MB bundle OBR is factor 3.69 faster than OSGi*4C*, the factor shrinks[5] to 2.05 for a 10 MB sized bundle. Performance in the inter-university WAN scenario is similar to the local network scenario. Although the OSGi*4C* results show that the JXTA data transfer does not scale well, bundle sizes greater than 1 MB are rather an exception for OSGi. The Internet WAN scenario is similar to the previous performance cases. However, it shows the great opportunity of OSGi*4C*: while the probability of a local–network-connected OBR repository is quite low, the probability of another OSGi*4C* system within the same local network having already loaded a needed bundle is higher (e.g., a team working in the same department probably needs the same software; thus, the probability is high that a local team member already shares a required update). This can drastically decrease deployment time due to loading from a local–network-connected node instead of an Internet-connected one (OBR Internet WAN deployment times vs. OSGi*4C* LAN deployment time).

Additionally, OSGi*4C* solves some issues of the OBR. For instance, OBR does not support non-functional properties during the discovery and selection process. Moreover, OBR automatically selects the first only functionally fitting bundle. OBR is a pure server-based solution with its well-known limitations, e.g., scalability (the performance evaluation was made with a very small bundle repository providing only 10 bundles; in case of a bigger one, a longer XML file would have to be loaded and parsed). Our approach enables each OSGi system running OSGi*4C* to share bundles, which reduces administrative efforts to a minimum. Moreover, OSGi*4C* is independent from the data transfer mechanism (OBR is restricted to HTTP). For the measurements, we used our JXTA-based data handler, which provides the ability of data transfer across firewalls and via Bluetooth. Last but not least, using the OBR forces manual bundle selection, while OSGi*4C* allows transparent service instantiation (see Section 6.3). Thus, considering the solved issues of the OBR the overhead of our OSGi*4C* platform is adequate. Especially in the case that a local OSGi*4C* system already loaded a needed bundle the overhead of our approach is low or it performs even better than the OBR.

7.2 OSGi*4C* vs. SOAP-based Web services

In this section we provide measurements for scenarios, in which it might be useful to allow the configuration of the decision to deploy a bundle on demand or to use it remotely as a Web service, such as in typical cloud scenarios (see Section 3). On the one hand, an application running on a PDA would prefer to use a computation-intensive bundle remotely by using a remote Web service due to the resource restrictions. On the other hand, the same application running on a desktop machine with a slow network connection would deploy the bundle with our OSGi*4C* platform due to the benefits of local access.

We measured the different invocation times for a remote SOAP-based Web service and a local bundle that is deployed with OSGi*4C*. For gaining comparable results we

[5]Discovery time is independent from the bundle size, whereas the transfer time increases accordingly. Thus, with an increasing bundle size, discovery time becomes insignificant.

Bundle Size	Platform	Deployment Time (s)		
		LAN	WAN	
		Local Network	Inter-University	Internet
0.1 MB	OSGi4C	1.55 ± 0.10	1.55 ± 0.07	1.55 ± 0.06
	OBR	0.42 ± 0.10	0.48 ± 0.05	0.49 ± 0.05
1 MB	OSGi4C	1.90 ± 0.05	2.31 ± 0.10	4.80 ± 0.24
	OBR	0.56 ± 0.05	0.64 ± 0.06	3.01 ± 0.17
10 MB	OSGi4C	6.81 ± 0.08	11.42 ± 0.22	33.80 ± 0.26
	OBR	3.31 ± 0.09	3.32 ± 0.12	24.74 ± 0.21

Table 2: Evaluation of bundle loading time with OSGi4C vs. OBR

Test	Latency (μs)		OSGi4C beats SOAP
	Remote	Local	Invocation Number
ping void	6722.857 ± 98.954	0.003	241
ping int	6935.714 ± 200.845	0.003	234
return int	8335.714 ± 513.054	0.003	195
ping (int,int)	6804.286 ± 104.315	0.003	239
ping byte[100]	6932.857 ± 187.899	0.002	234
ping byte[200]	6804.286 ± 108.477	0.002	239
ping byte[800]	6764.286 ± 59.966	0.002	240
ping byte[102400]	23591.250 ± 90.199	0.002	69
ping int[100]	6800.000 ± 55.032	0.002	239
ping int[200]	8334.286 ± 234.207	0.002	195
ping int[25600]	215144.286 ± 803.881	0.002	8
ping float[100]	6666.250 ± 23.419	0.002	244
ping float[200]	8318.571 ± 95.831	0.002	195
ping float[25600]	225048.571 ± 422.152	0.002	8
ping double[100]	6652.857 ± 34.107	0.002	244
ping double[12800]	115395.714 ± 180.385	0.002	15

Table 3: JavaParty benchmark results for local OSGi4C and remote SOAP Web service access

used the well-known standard JavaParty Benchmark[6]. Table 3 shows the results. The tests were performed on Intel Core2 Duo machine(s) with 2.33GHz and 3GB RAM. We used the Equinox OSGi framework implementation running with Sun Java 1.6.0_03 and a 100 Mbit switched Ethernet LAN. Table 3 shows only the pure call latency. We restricted the test bed to a LAN environment due to the fact that the JavaParty benchmark bundle has only about 100kB size. Thus, the LAN scenario represents the worst-case scenario for OSGi4C because the loading time is equal in a WAN setting (see Table 2) but SOAP-based Web service invocation times increase.

As expected, local invocations of loaded bundles perform much better (the accuracy of Java-internal timing does not allow to measure the standard deviation). The deployment of the JavaParty benchmark bundle with OSGi4C takes in each case only 1.62±0.07s. In the worst case of the JavaParty benchmark (`ping float[100]`) OSGi4C outperforms the SOAP-based Web service invocation after 244 invocations. In the best case (e.g., `ping int[25600]`) we outperform the SOAP-based benchmark Web service already after only 8 invocations.

To sum it up, the benefit of OSGi4C highly depends on the expected invocation count. However, we observed that further system parameters such as the CPU, memory, current load, network usage and bundle size influence the JavaParty benchmark in our evaluation. Thus, it is impossible to certainly determine the best alternative in advance. However, our measurements show that in general it is feasible to deploy bundles on demand instead of using them remotely.

[6]http://www.ipd.uka.de/JavaParty

8. RELATED WORK

Many code deployment systems are server-based and rely on a central repository. The following systems suffer from well-known limitations of server-based solutions, e.g., additional administrative effort and lacking scalability. A P2P approach can solve these issues and still include server nodes. Unless otherwise noted, *none* of the presented approaches support the automatic selection of the best-fitting software according to non-functional requirements. Section 4 highlighted the necessity of non-functional properties in context of this work.

Java Web Start [21] is a deployment system for Java software on the basis of the Java Network Launching Protocol. Requirements of the application and the needed code are described in XML. A Java Web Start client is able to install the application by evaluating the XML description. Java Web Start targets the deployment of whole applications and is not suited for dynamic and incremental dependency reso-

lution such as needed in the context of OSGi.

We proposed the *Dynamic Loading Service* (DLS) in context of CORBA [22]. It allows the installation of locally unavailable code on the basis of an IDL interface name. A local DLS client queries a central repository and returns available implementations. Unlike the approach in this work, implementation selection is only supported by simple policies and dependent code is not resolved automatically.

A central code server is provided by the *OSGi Bundle Repository* (OBR [5]). OBR is based on a Web server providing an XML file describing available bundles. This XML can be parsed by clients but current clients do not support dynamic changes. Unlike our approach, OBR does not support queries without the fully-qualified bundle name and thus does not support discovery on the basis of keywords or other criteria. A simple distributed OBR was presented by *Frenot et al.* [23]. However, only selection and loading of bundles based on Java package dependencies but no selection based on service dependencies is supported. Just as with the OBR, bundles have to be installed explicitly.

In [10, 24], we proposed an extension to the DLS using a P2P-based infrastructure to solve issues of server-based solutions. In contrast to this approach, our current work provides filtering mechanisms in the network, automatically resolves dependencies, and is transparently integrated as a service into OSGi.

Paal et al. proposed a proprietary code loading platform on the basis of multiple repositories [25]. Repositories are queried at runtime by a custom application loader. Unlike our approach, the infrastructure does not provide compatibility to OSGi and application repositories have to be preconfigured before runtime.

Parker and Cleary implemented a JXTA-based infrastructure for remote loading of Java classes as an alternative to the standard Java class loader mechanism [26]. This provides fine-grained code loading on the basis of Java classes but does not support OSGi.

Addressing of remote OSGi services in a way comparable to Java RMI is implemented by *R-OSGi* [27]. This allows using remote services instead of loading code on demand. Thus, the support of R-OSGi is orthogonal to our approach and both frameworks can be used in conjunction.

The *SATIN* component model was developed to support reconfiguration of mobile applications at runtime [28]. It provides discovery mechanisms to search for available components. Components can be integrated into the application by installing or migrating them to the local host. The approach builds on a proprietary component framework and does not support OSGi.

Wu et al. developed abstract state machine models for dynamic service updates in OSGi [29]. This high-level model provides means to determine if given updating constraints are fulfilled. However, this work provides only a model for updating whereas our work provides a concrete platform to actually discover and deploy update bundles. Thus, both approaches can be used in conjunction.

9. CONCLUSION AND FUTURE WORK

In this paper we presented OSGi*4C*, a dynamic code discovery, selection and deployment infrastructure for OSGi. In contrast to related approaches, OSGi*4C* allows centralised and decentralised sharing and discovery of bundles and services at runtime. Resources are automatically selected on the basis of functional and particularly non-functional properties. For local integration of unavailable bundles and services OSGi*4C* transparently supports discovery, selection, loading and deployment of resources that are propagated in a JXTA network. Thus, OSGi*4C* supports emerging cloud computing scenarios as a base platform. We provide an elaborate discussion of the necessity of considering non-functional properties in such a system. Last, we present an extensive performance evaluation of our prototype: we compare our approach with OBR and the use of remote Web services.

We argued for the necessity of supporting non-functional properties in Section 4. For an improved handling of these properties in larger systems, support for ontologies describing non-functional properties could be necessary. This may build the basis for implementing an enhanced matching of dependencies using semantic discovery approaches. So far, we support static non-functional properties. In the future, support for configurable services with variable non-functional properties depending on the configuration might proof to be useful. As already mentioned in Section 4, it is important to agree about the evaluation function as well as the conditions. Having this ensured, configurability of service parameters would provide even more flexibility and choices during the selection process.

We will investigate the use of automatic policies for supporting the decision of dynamic loading versus remote access of services. For instance, such a policy could automatically lead to dynamic loading of a service bundle whenever the number of service calls of a specific remote service exceeds a given limit for a timeslot. This is a non-trivial task because issues, such as the current state of the service have to be taken into consideration.

We would like to integrate the OSGi Declarative Services specification [11] as this becomes the standard for service dependency specification instead of the `Import-Service` manifest header. Until now, only few OSGi framework implementations, such as Eclipse Equinox, provide a implementation.

Our OSGi*4C* infrastructure provides a basis for application deployment with a given blueprint. This blueprint information can be used to select, load and deploy tailored bundles providing the required service functionality with our infrastructure. An enhancement of such a simple blueprint mechanism—a sophisticated distributed deployment platform—is subject to future work.

10. REFERENCES

[1] OSGi Alliance. OSGi Service Platform: Core Specification, Release 4, Version 4.1. Technical report, 2007.

[2] A. Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, 2007.

[3] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2, 2009.

[4] Steven Hand, Tim Harris, and Evangelos Kotsovinos amd Ian Pratt. Controlling the XenoServer open platform. In *Proceedings of the Sixth IEEE Conference on Open Architectures and Network Programming (OPENARCH 2003)*, pages 3–11. IEEE Computer Society, April 2003.

[5] OSGi Alliance. RFC-0112 Bundle Repository, February 2006.

[6] H. Schmidt, J. H. Yip, F. J. Hauck, and R. Kapitza. Decentralised Dynamic Code Management for OSGi. In *6th MiNEMA workshop*. ACM, 2008.

[7] D. K. Barry. *Web Services and Service-Oriented Architectures. The savvy manager's guide. Your road map to emerging IT*. Morgan Kaufmann, 2004.

[8] L. Gong. JXTA: A Network Programming Environment. *IEEE Internet Computing*, 5(3), 2001.

[9] The Internet Society. JXTA v2.0 Protocols Specification. Technical report, Sun Microsystems, 2007.

[10] 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.

[11] OSGi Alliance. OSGi Service Platform: Service Compendium, Release 4, Version 4.1. Technical report, 2007.

[12] W3C. OWL Web Ontology Language Overview. W3C recomm., 2004.

[13] I. Lera, C. Juiz, and R. Puigjaner. Performance-related ontologies for ubiquitous intelligence based on semantic web applications. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06)*, pages 675–682, Washington, DC, USA, 2006. IEEE Computer Society.

[14] David Mosberger and Tai Jin. httperf–a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

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

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

[17] Network Working Group. (Extensible Markup Language) XML-Signature Syntax and Processing. IETF RFC 3275, 2002.

[18] W3C. XML Encryption Syntax and Processing. http://www.w3.org/TR/xmlenc-core/, 2002.

[19] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, 1993.

[20] G. C. Necula. Proof-Carrying Code. In *POPL'97*, pages 106–119, Paris, 1997.

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

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

[23] S. Frenot and Y. Royon. Component Deployment Using a Peer-to-Peer Overlay. In *Component Deployment*, volume 3798 of *LNCS*, 2005.

[24] 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.

[25] S. Paal, R. Kammüller, and B. Freisleben. Self-Managing Application Composition for Cross-Platform Operating Environments. In *ICAS '06*. IEEE, 2006.

[26] D. Parker and D. Cleary. A P2P Approach to ClassLoading in Java. In *AP2PC '03*, 2003.

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

[28] S. Zachariadis, C. Mascolo, and W. Emmerich. The SATIN Component System-A Metamodel for Engineering Adaptable Mobile Systems. *IEEE TOSE*, 32(11), 2006.

[29] J. Wu, L. Huang, and D. Wang. ASM-based model of dynamic service update in OSGi. *SIGSOFT Softw. Eng. Notes*, 33(2):1–8, 2008.