World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:7, No:5, 2013

OSGi in Cloud Environments

Irina Astrova, Arne Koschel, Björn Siekmann, Mark Starrach, Christopher Tebbe, Stefan Wolf, and Marc Schaaf

***Abstract*—**This paper deals with the combination of OSGi and cloud computing. Both technologies are mainly placed in the field of distributed computing. Therefore, it is discussed how different approaches from different institutions work. In addition, the approaches are compared to each other.

***Keywords*—**Cloud computing, OSGi, distributed environments.

I. Introduction

THERE are a few reasons why it sounds practical to bring two technologies – OSGi and cloud computing – together. On the one hand, OSGi [16] works well in distributed environments. It is possible to have a repository that gives access to all components (called bundles) and their offered services. On the other hand, cloud computing is a service that offers computing power, storage or platforms in a network connected field. Therefore, it is obvious to get both technologies together as a working platform.

The OSGi framework provides functionalities to build applications consisting of different modules. Such modules are called bundles. A bundle contains its own address space and class loader. The several bundles communicate over the services they import from other bundles or offer to other bundles. This behavior is defined in a manifest file for each bundle. A special service registry is needed to register and get services. The communication between bundles could be compared to the approach the service oriented architecture provides.

As already mentioned a cloud contains many connected computers (nodes) which are used to run distributed applications. To develop applications in a distributed way often outcomes in a very complex planning, implementation and testing phase. Even more the software developer is not able to get a complete overview of the complexity of the whole software system in most cases. OSGi can provide simple and well known mechanisms to handle such problems. The OSGi framework itself is limited on using bundles on only one node. But there are different approaches to circumvent these limitations to enable distribution of bundles to arbitrary nodes, which leads to completely distributed applications. Some of these approaches are presented in this paper. When using OSGi the developer has no longer to take care of the later distribution of the system, because he can develop and test the application on one pc. The formally mentioned approaches could then be used to distribute the bundles on arbitrary nodes transparently.

Irina Astrova is with the Institute of Cybernetics, Tallinn University of Technology, Estonia (e-mail: irina@cs.ioc.ee).

Arne Koschel, Björn Siekmann, Mark Starrach, Christopher Tebbe, and Stefan Wolf are with the Faculty IV, Department for Computer Science, University of Applied Sciences and Arts Hannover, Hannover Germany (e-mail: arne.koschel@hs-hannover.de).

Marc Schaaf is with the Institute for Information Systems, University of Applied Sciences Northwestern Switzerland, Olten Switzerland (e-mail: marc.schaaf@fhnw.ch).

The first approach presented in Section II is a draft by the OSGi Alliance that was produced while a workshop with the theme of combining both techniques. Section III presents the Cloud Computing API and the following Section IV deals with OSGi Remote Services, which extend OSGi with remote exporting and importing of services. Section V describes the concept of OSGi*4C*, which was developed at the universities of Ulm and Erlangen. The last approach is called R-OSGi, which was developed at the ETH Zurich in Switzerland is discussed in Section VI. Finally, Section VII compares the different approaches with each other.

II. The OSGi Alliance Approach

In this section an approach of the OSGi Alliance is discussed in detail. In March 2010 the OSGi Alliance held a workshop with the topic of discussing how OSGi and Cloud Computing can be combined. The result of this workshop is a draft [7]. A lot of recommendations and assumptions about the combination of both techniques are made within this paper. In fact the paper is only a theoretical approach to the topic. But it handles the main sources of problems and gives the according answers. The paper was in a real early state when this document was written. But the main aspects are mostly clear defined. It can also be seen, that the paper is continuously refined, due to the version history.

A. Guidelines

Within the paper there are several guidelines that the authors mention as good rules to bring Cloud Computing to the OSGi world. It is mentioned that the core aspect is based on the dependencies. This is caused by the fact that the OSGi framework is based on the combination of single components or rather bundles that can use the services of other bundles that they are connected with. This enables the modular and service oriented programming style with the OSGi framework. One concerning fact on the dependencies is the USE/REUSE Paradox [4]. It describes the fact that the reusing ability of software components depends on how the components and the resulting dependencies are modeled. If the components are too coarse-grained they are easier to use but the factor on how reusable they are gets smaller. This is caused by the fact that

International Science Index, Computer and Information Engineering Vol:7, No:5, 2013 waset.org/Publication/5362

International Science Index, Computer and Information Engineering Vol:7, No:5, 2013 waset.org/Publication/5362

the coarse-grained components are clearly more inflexible and static. On the other hand if the components are more fine-grained they would be more complicated to use but they are more reusable because they fit better into more different problems. The other point is the weight of a component. If the component is of heavy weight it can be easier to use it, because the most things are capsule in it. But if the component is more lightweight it is easier to reuse, because it can be set easier into another context. But this would cause more configuration work. For the dependencies between the bundles the draft includes three aspects that should be used: sized just right, avoiding vendor lock-in and reactive runtime.

It is important to get the sizes of the bundles in the right way. If the cloud environment would be built with the help of virtual machines the deployment would be static and ponderous. Bandwidth and calculating power is needed for every deployment. If it would be possible to deploy just single bundles on a cloud and the need for a redeploy only depends on the changed bundles, it could spare some resources. The effect is heavier if the infrastructure gets greater. Another fundamental point is to avoid a vendor lock-in. As today's cloud computing solutions are often based on proprietary virtual platforms. If the user stays with one company the problem is not as big. But if he wants to port his virtually built up systems to a platform by another vendor, it could cause trouble. In the worst case the user has to copy all data from one vendor, build up a complete new environment at the second vendor and then merge the data into the new system. Because of this massive lack of interoperability it is necessary to get way to port between vendors in an easier way.

To get more interoperability into the cloud for OSGi applications, the OSGi Alliance wants to build up a repository. This can be compared to the service registry that the OSGi framework has included. The OSGi based cloud environment would have a central registry with all bundles stored in it. This registry is called OBR (**O**SGi **B**undle **R**epository). Within the OBR all available bundles of one cloud are held. Due to the fact that a bundle is self-describing and contains all bundles that it depends on, the dependencies can be easily dynamically solved. For the case that a user wants to change the vendor of his cloud computing environment, it is easier. If the user uses OSGi to build up his software and he wants to change the vendor. He just ports the bundles to the other vendor and all dependencies are resolved by the OBR. This behavior can enable an easier vendor change and moving from one platform of one vendor to one of another vendor. For this behavior it is needed to have uniform OBR that is available on every OSGi supporting cloud environment. A similar approach to a bundle repository for the cloud is the Oscar Bundle Repository [6]. But this approach is more general and has its roots in a public distribution of bundles over a central point in the Internet. It is not specialized for the use in a cloud environment.

The OSGi Alliance mentions as a third point that a reactive runtime should also be applicable. That means that if a bundle is changed, the OBR manages the dependencies and it enables updates and rollbacks in an easier manner as if using static virtual machines that have to be replaced with a new version. This would lead to a loss of agility and more workload for the developers or users. In addition to the reactivity a virtual machine has to be uploaded over the Internet or at least over the network. This could cause long upload times on the one hand. On the other hand it occupies bandwidth. If only single bundles are uploaded and the dependencies are resolved, it would save a lot of the network bandwidth. The design of bundles abets the reactive runtime. With the meta data that is stored within the manifest file the dependencies are included. With this foundation it is easy to resolve the dependencies with the help of the OBR.

With this behavior it would also be possible to let the deployed system grow with the needs of the user. If e.g. a Software developing company wants to expand and let their provided systems grow. They just need more computation power bought from the cloud provider. It would also be possible to stay at a certain level of agility. Because they still only have to redeploy the bundles for special needs of the customer.

B. Conclusion

Overall the draft can be seen as a good collection of guidelines on how to develop with the OSGi framework in a cloud environment. Due to the theoretical approach it will be supposably stay as advises on how to build up such a system. Even this early version of the paper has good ideas, use cases and advises to build up upon OSGi in the Cloud. This is mostly caused by the fact that the workshop members and also the OSGi Alliance consist of professionals from the industry and well known institutions. If the combination of both techniques is working on a PaaS-Platform with OSGi libraries and an implementation of an ORB, it could lead to an agile and vendor unspecific solution. The reaction on changing bundles can be more reactive and the reusability can be enhanced.

III. Cloud Computing API

A paper of the university Minho (Universidade do Minho) [2] from Portugal describes CAPI (Cloud Computing API). This new concept targets to an abstract API for the cloud. This general API, the Cloud API, shall help to prevent the vendor lock-in when creating an application for the cloud. Additionally the benefits of OSGi shall improve the portability of created cloud applications.

In the actual situation every vendor specializes on one or more layers of the cloud stack (IaaS, PaaS or SaaS). Additionally the vendors offer mostly only a few services which are specialized on a certain sector. The most important disadvantage is that every vendor offers his own proprietary API for his services. That forces the developer to build their applications upon a certain specialized proprietary API, since there is no standardized interface to access a service in the cloud. So every developer has to deal with a new proprietary API before he can build applications. Changing the cloud

International Science Index, Computer and Information Engineering Vol:7, No:5, 2013 waset.org/Publication/5362

vendor is very difficult to realize, since the whole application needs to get refactored to another proprietary API. This results in a very poor portability of cloud applications. The Cloud API tries to avoid all this deficits with an abstract API that uses OSGi for dynamic loading of single bundles.

A. Architecture

The fundamental architecture of the Cloud API, as shown in Fig. 1, is based on an OSGi environment and further bundles to connect all layers of the cloud stack. On this OSGi environment the other ingredients are put on. The modular properties of OSGi help to define modules and to control the module life-cycles. OSGi supports also the dynamic loading and unloading of modules at the runtime. This is very helpfully for the Cloud API.

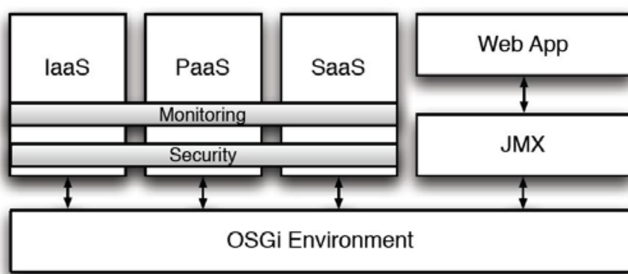

Fig. 1 CAPI Architecture

B. Cloud API Modules

The target of the Cloud API is to abstract the cloud stack to single modules. The new defined interfaces are very general to be mostly universally applicable. To reach this abstraction, everything should be seen as an entity. Additionally there are monitoring and security capabilities integrated in these modules.

In the Infrastructure as a Service (IaaS) layer, everything is seen as a resource. This can be a hard disk, a CPU, storage and so on. In this layer, three new interfaces are defined. The Resource Pack consists of pre-build images. When a Resource Pack is running, it is handled as a Resource Container with the possibility to add or remove resources at runtime. The interface called Resource Manager defines access to the modules and offers the ability of monitoring.

The Platform as a Service (PaaS) layer module abstracts running applications. They are defined as Managed Applications and through this the applications get a well defined life-cycle. The Managed Applications can get started or stopped and also the status of the application can get monitored.

On the Software as a Service (SaaS) layer ready applications are offered. Here the interface Monitored Services is introduced which adds the possibility to monitor the application. Since this layer is very inflexible only the monitoring can be added through the Cloud API.

The Cloud API uses standard JMX (Java Management eXtensions) to export the CAPI interface and therefore it offers access for a web interface.

C. Conclusion

The Cloud API describes a very useful approach of an abstract and general API to be used for the cloud. This abstract API could combine all proprietary APIs to support the consumer and developer of cloud applications. The Cloud API would also enhance the portability of developed applications. However this paper is very short and can therefore only considered as a rough draft. Furthermore the cloud vendors have to unify on this standard abstract API. This leads to more problems and delays the standardization.

IV. OSGi Remote Services

The OSGi framework offers the possibility of communication between bundles inside of one OSGi framework (local). The bundles will use services for the communication with each other. Services can get registered on the Service Registry by the bundles itself. So every bundle can offer services. Other bundles can use the Service Registry to find and get the offered services. OSGi remote services extend the local communication between two bundles to a communication between two bundles on different OSGi frameworks. This communication will then be done with endpoints which provide access to services in other OSGi frameworks.

A. Architecture

The general architecture of OSGi remote services is shown in Fig. 2.

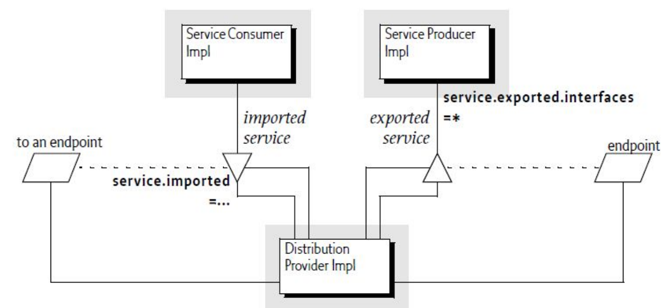

Fig. 2 OSGi Remote Services Architecture

A bundle can export a service and therefore registering it to the service registry. The distribution provider will then create one or more endpoints for this exported service, if some important conditions are set. Other distribution providers are able to import the exported services. This also allows other OSGi frameworks to import remote services. Importing a remote service is generally the same as importing a local service. There are only a few capabilities or properties that need to be fulfilled. Next to the importing of a remote service a bundle has the possibility to search for services. These remote services are then added automatically if they are available.

Invoking a remote service is slightly different from invoking a local service. There are fewer opportunities on

International Science Index, Computer and Information Engineering Vol:7, No:5, 2013 waset.org/Publication/5362

invoking a remote service. To compensate this weakness the Properties for remote services are defined. These Properties try to map the whole opportunities of invoking local services, also to remote services. The Properties are passed to the remote service while registering it to the Service Registry.

Security is an important topic. Since with remote services the calls are no longer only local, the security gets a more important status. The distribution provider needs to prevent that bundles get more permissions through the fact that the distribution provider is calling the imported service. The distribution provider should therefore limit the available permissions. Additionally the distribution provider is a worthwhile target so it should be secured very well.

B. Conclusion

OSGi Remote Services are a good approach to enable exporting and importing services to and from other OSGi frameworks. This is a good deal to integrate OSGi into the cloud. And this can also be done with nearly the same usage of OSGi services. This enables to distribute the services remotely and use it somehow like local services. The current OSGi Enterprise Specification [16] currently already contains the OSGi Remote Services. However the OSGi Remote Services will get improved and are therefore still in development.

V. OSGi*4C*: Enabling OSGi for the Cloud

A. Introduction

This section will explain the general properties of OSGi*4C*. OSGi*4C* was developed at the universities of Ulm and Erlangen. The results were published in an ACM paper [15] in 2009 which is the basis of the following preparation. The mean focus of the OSGi*4C* platform are described by the following three headwords: centralized and decentralized discovery of bundles, automatic selection of bundles with the help of functional and non functional properties and transparent support of discovery, selection and deployment of bundles.

The following section will give a brief overview about the concepts, the architecture and the central functions of OSGi*4C*. The first part of this section will deal with the technologies of OSGi*4C*. After that the basic architecture and the main components of the platform will be presented. In today's companies fast work flows always play a big deal, so this section also will investigate the performance of OSGi*4C* in contrast to other possibilities to share functionalities over a network. At the end of this section the benefits and disadvantage will be discussed.

B. Technology

To decentralize all bundles in a network, OSGi*4C* uses a peer to peer approach. The basic idea is to decouple the whole architecture from the underlying peer to peer implementation. Since this is only a prototype, the developer focuses on JXTA [17] as peer to peer platform. JXTA is completely free and was initially developed by Sun Microsystems in 2001. JXTA defines a lot of platform and implementation independent protocols which are based on XML. JXTA is available for the most popular programming languages (like C, C++, Java and C#). JXTA resources are organized in peers and more peers in a peer group. Peers in a peer group are sharing a specified context and work together for the group. Each resource (peer and peer groups) has a network wide unique id.

For bundles in a network with shared resources it is very important to communicate and to share such resources. For this approach the developers' analysis different possibilities to implement an OSGi HTTP service. Different implementations of the OSGi HTTP service are fundamental for environments which consist of clients with different hardware capacities. The paper provides three HTTP implementations which are applicable for different platforms. For platforms with low hardware capacities the developers use NanoHTTPD which only consists of a single Java class. It has a very low memory footprint but has cut backs in performance because of no multi threading. The other approaches are the Knopflerfish implementation with and without server side caching. Both supporting multi threading and in both implementations this results in higher memory footprint and the need of more CPU power.

C. Architecture

1) Architecture Layers

As shown in Fig. 3 the complete OSGi*4C* architecture consists of four different layers which are independent of each other. At the top there is the application bundle which describes the individual application of each client. If the application bundle needs bundles from somewhere in the peer to peer infrastructure it starts to communicate with the OSGi*4C* layer. The layer consists of two mean components. At first there are three OSGi*4C* Services, which will be described in the following

- **Loading Service:** This service is responsible for automatic selection and automatic loading of needed bundles. the loading service enables dynamic integration of bundles which are loaded from the peer to peer infrastructure
- **Repository Service:** The repository service manages and loads locally available bundles
- **Resolver Service:** With the input of the already mentioned services the resolver services automatically resolve bundles and service dependencies. It communicates directly with the services provided by the JXTA platform to load bundles from the network. The JXTA services will be describes later in this section.

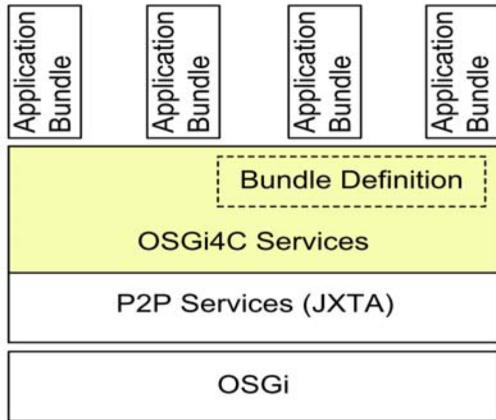

Fig. 3 OSGi*4C* Architecture

The next layer in the architecture is the JXTA layer it provides all necessary functionalities to share resources and thus bundles over the peer to peer infrastructure. JXTA consists of two different services which take care of the dissemination and retrieval of resources in the peer to peer network. The *code sharing service* enables loading and sharing of bundles while the *code discovery service* provides discovery of bundles and publishing the meta data of a JXTA resource. The bottom layer is standard OSGi. It is not necessary to change anything on the respective OSGi implementation.

2) Meta Data

As mentioned in the previous section some meta data is needed to give informations about resources and bundles in the peer to peer infrastructure. As shown in Fig. 4, a bundle has to be enhanced by some properties. The *Interface* gives a detailed description about the functionality a bundle provides. *Functional properties* gives an overview about additional functionalities the bundle provides. An example for this may be the HTTP services each bundle implements.

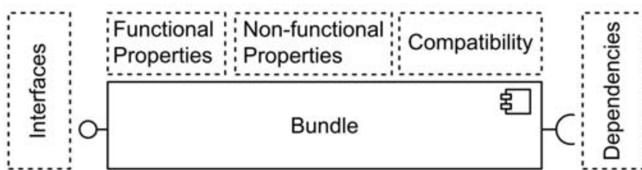

Fig. 4 OSGi Bundle Description

Non functional properties meanly define quality of service attributes like security or performance. *Compatibilities* are necessary for additional system properties a bundle needs like the JDK version or something like that. *Dependencies* may be the most important part of a bundles meta data. The dependencies part gives a detailed overview about all additional bundles and services a bundle needs to provide complete functionality. If more bundles in the cloud provide a needed functionality an internal rating systems rates all found bundles. For the internal rating non functional properties will be used to calculate an average rating sum.

To publish OSGi bundles as a JXTA resource in a peer group some advertisements are needed. The *interface description advertisement* (IDA) publicizes the existence of an interface in the network. The IDA can be searched by using the full interface name of the searched bundle. The next one is the *resource advertisement* (RA); it gives a detailed description about the functionality of an interface. The last one is the *code description advertisement* (CDA) it describes how an interface is implemented for a specific platform. Platform depended implementations are useful.

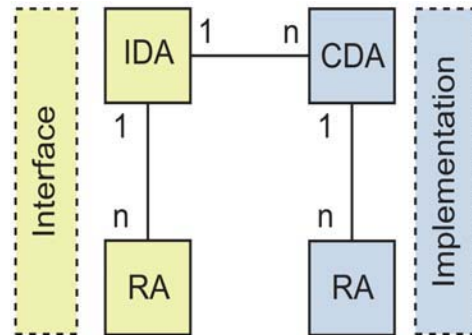

Fig. 5 JXTA advertisements

Since a cloud infrastructure is often used by very different terminals, a solution is needed for a powerful desktop pc as same as for a limited mobile device. Code description advertisements also provide some more meta data to publicize how an interface can be loaded. All just introduced advertisements can refer to each other by using a unique id. Fig. 5 illustrates how advertisements refer to each other. It also shows that the IDA can be seen as the mean advertisement which can refer all other advertisements. It publicizes the existence of an interface and points to the provided functionality (RA) and also links to the real implementation (CDA). The CDA also links to one or more RA's which contains meta data and implementation details to load a resource.

D. Performance

As mentioned in the introduction to this chapter performance plays a big deal in modern companies. The developer of OSGi*4C* also realizes that it has to be evaluated if the new solution can obtain performance results as currently available technologies. The paper focus on comparisons between OSGi*4C* and OBR respectively SOAP based Web Services. Table I shows an excerpt of the results provided in the paper.

International Science Index, Computer and Information Engineering Vol:7, No:5, 2013 waset.org/Publication/5362

TABLE I
COMPARISON OF OSGi*4C* VS. OBR TO LOAD A BUNDLE (IN SECONDS)

Bundle Size	LAN	Internet
0.1 MB	OSGi*4C* (1.55)	OSGi*4C* (1.55)
	OBR (0.42)	OBR (0.49)
1.0 MB	OSGi*4C* (1.90)	OSGi*4C* (4.80)
	OBR (0.56)	OBR (3.01)
10 MB	OSGi*4C* (6.81)	OBR (22.74)
	OBR (3.31)	OSGi*4C* (33.80)

As seen the result of the comparison is that an centralized OBR approach is must faster in allocating a bundle. Especially a bundle with a size of 10 MB visualizes the extreme differences. While OSGi*4C* is much slower than an OBR approach in is nearly as fast as a Web Service solution. It really depends on how often a service is invoked. Nevertheless OSGi*4C* offers some benefits beside performance which will be discussed in the following section.

E. Conclusion

The previous sections gave a brief overview about the architecture and functionality of OSGi*4C*. In conclusion it can be said that OSGi*4C* has some very good approaches. The decentralized peer to peer architecture makes sense in matter of independent bundles. Each bundle has its own peer in the infrastructure. For an developer OSGi*4C* might be easy to use because OSGi remains intact. As a matter of this the developer does not need to rethink his/her development steps when he/she is already familiar with OSGi. The implementation also provides integration in the consoles of Eclipse Equinox and Apache Felix. A problem might be the performance but in fact of a decentralized approach and the lost of a single point of failure this may be neglected for the first time. Some more problematic may be the focus on pure meta data. Meta data often leads to failures because often the developer has to set them on his/her own. The rating system shortly mentioned in the section above also leads to more problems because a good and useful rating is very complex. Ratings may differ from developer to developer and not functional properties which are important for OSGi*4C* might be less important for the developer. Its also problematic that meta data are the basis of the rating system which are not really checked. The disadvantages of meta data also results in another problem named security. Meta data can be faked, a hacker or something like that can use meta data of another bundle to spread the new malicious bundle in the cloud. The peer to peer cloud of OSGi*4C* also works *unchecked* as a matter of this everyone can add new peers to the cloud, even when they are malicious. A solution for both problems might be a signature based approach which is also offered in the paper. All bundles have to be signed and if not the client reject the bundle. Another security problem is that all JXTA resources communicate via plain XML messages. As a result of this a hacker is able to read all messages send via the network. He is also able to change message or work as a man in the middle. It follows that XML security is highly required for a further commercial use.

Also fault tolerance is not described by the paper. It is quite obviously that this could be managed by multiple peers providing one bundle. Also how load distribution works is not considered.

VI.R-OSGI

With R-OSGi [14] it acts as likewise with OSGi*4C* around an almost finished implementation, for the distributed use of OSGi Bundles, which is likewise based on its own concept. The concept of distributing Bundles on multiple nodes differs strongly in the two approaches. The communication between nodes differs too, although both use some kind of peer to peer communication. For this reason R-OSGi will be presented in this section as an additional possibility, to distribute OSGi applications. R-OSGi could be used to distribute an application in the cloud over an arbitrary number of nodes (instances), to keep the application available for the user, without fearing a loss of performance.

R-OSGi offers the advantage of developing and testing applications on a single computer and distributing single bundles of the application at deploy time to arbitrary nodes (with OSGi-Framework and R-OSGi). For this reason the developer is discharged, because he has no longer to deal with the distribution of the application in the cloud. R-OSGi supports the distribution of existent OSGi-Applications, with no need to change the application itself, how this works is described later on.

R-OSGi can be seen as a kind of middleware layer, which is attached on OSGi. Hence all benefits of OSGi are available when using R-OSGi, too. Additionally it is possible to offer all valid OSGi services as remote services. Between nodes there is no hierarchy (no Client/Server), but a symmetric relation, which can be regarded as a kind of peer to peer connection. To keep the transparency for the local OSGi frameworks R-OSGi makes sure that remote calls look like a local call of a service. An advantage of R-OSGi is, that no stubs or skeletons have to be created, to enable this. Proxy bundles are used to ensure the transparency instead. If a node respectively a bundle on a node retrieves a remote service, a proxy bundle is installed on that node by R-OSGi, which the local OSGi framework uses to call the service. For the local framework the proxy bundle is the original service provider, so it does not know that it calls a remote service. With this technique bundles could be distributed to arbitrary nodes in the cloud.

These Properties of R-OSGi have been presented in [10] and partly in [11], where six requirements were presented, which R-OSGi has to fulfill (something similar was presented in [8]). Some of them have already been mentioned:

- **Seamless embedding in OSGi:** It is necessary that remote and local services are indistinguishable for the local OSGi framework to provide a transparent

International Science Index, Computer and Information Engineering Vol:7, No:5, 2013 waset.org/Publication/5362

distribution of bundles. Additionally R-OSGi must be able to distribute existent OSGi applications, without changing the application itself.

- **Reliability:** Through the distribution of bundles no new failures are added to the OSGi framework. New appearing failures, such as network problems, are mapped to existence failures of the framework. Generality: The R-OSGi middleware does not limit the number of possible services. Every valid OSGi service has to be potentially distributable.
- **Portability:** R-OSGi has to be very portable, because its origin lies in embedded systems. Hence it is compatible since Java 1.2 and Java ME CDC.
- **Adaptivity:** There are no roles between nodes (no Client/Server), the relationship between two nodes is a symmetric one.
- **Efficiency:** R-OSGi has to be efficient in communication between nodes. Therefore a own binary protocol was implemented, which is slightly faster than the highly optimized Java RMI and two times faster than UPnP.

In R-OSGi there are two different possibilities to distribute an OSGi application respectively offering services to other remote bundles. The first possibility is the transparent way, which should be used when distributing existent applications. Otherwise the applications code has to be changed. To transparently offer remote services to other bundles a so called distributed service registry is used by R-OSGi. When a service, which should be accessible by remote nodes, is registered in an OSGi framework the distributed service registry is informed too. Now other bundles can request for those services over R-OSGi, which routes the request transparently for the framework to the distributed service registry. How that works in detail is presented in the previous subsection. Making the application aware of its distribution is the second possibility. Therefore a bundle which offers a remote service has to add a property, saying that this is a remote service, to the registration of the bundle with the local service registry. If such a property is set R-OSGi realizes it and perceives this service as a remote service. If now a remote consuming bundle wants to use the service, R-OSGi sends the service-reference to the consumer. To get a (known) remote service a consuming bundle has a lot more to do, than a service offering bundle. The primary condition is that the consumer knows which service is offered by which node (IP address and port). To build up a connection to another node and to get a list of all available services of that node the method *connect()* from R-OSGi can be used. It returns a list of the remote services of this node. The consumer can then search the list for the needed service and get the corresponding reference. Using this reference the consumer can call the remote service.

The second possibility should be used if a service discovery described in possibility one is not possible or available. But there is one problem; the location of the needed service has to be known before runtime [8].

A. Techniques

To keep the remote services completely transparent to the OSGi framework on a computer, different techniques are used. In this section these techniques are presented. These techniques are: proxy bundles, distributed service registry, type injection and failure transparent distribution [10].

Before going into details the type of communication channels used is shown. R-OSGi connects nodes with network channels which are persistent TCP connections. To make them persistent TCP keep alives are used. This minimizes the needed traffic to make a remote call; otherwise for every call a new TCP handshake is required. The used protocol is a self implemented binary protocol, which is quite fast as already mentioned. During the establishment of the connection leases are exchanged by the nodes. The leases contain the name of all offered remote services and a list of events a node is interested in. If a remote service changes new leases are exchanged.

1. Proxy Bundles

To make the OSGi framework believe that only local services exist proxy bundles are used, which have been mentioned earlier. For the OSGi framework the proxy bundles behave exactly like normal bundles, but in reality they just forward the service call to a remote node which offers the service. When a remote service is requested a proxy bundle is created dynamically on the requester framework by R-OSGi. First the service interface and the service properties of the remote service are send to the requesting bundle on the local node. Then R-OSGi uses the ASM library [1] to create the proxy bundle from the received service interface and properties and the local BundleContext of the OSGi-Framework through bytecode manipulation. The BundleContext is needed to register the remote services as local services to the service registry of the OSGi framework. Internally all service calls are mapped to the method *invokeMethod(final String serviceURL, final String methodSignature, final Object[] args)*, which takes the address of the remote node, the method signature of the remote service to call and the parameters of the service as parameters. The address of the remote node is hard coded into the proxy, because for every remote bundle a new proxy is generated. The method signature of the remote service is used to identify the service on the remote side. Because the proxy bundle is registered under the same name like the remote bundle (which is offering the remote service) and the remote services are registered as local services the OSGi framework can not differentiate between a local and a remote service. Another advantage of this technique is that services can communicate spontaneously with each other and the amount of data, which has to be stored on the offering node and send through the network, is reduced to a minimum, due to the fact that the proxy code is generated dynamically at runtime. Fig. 6 shows an example of a proxy bundle which offers the remote service of the left node to the OSGi framework of the right node.

International Science Index, Computer and Information Engineering Vol:7, No:5, 2013 waset.org/Publication/5362

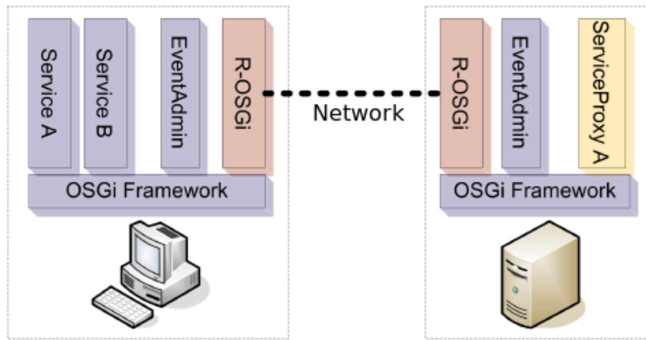

Fig. 6 Example for usage of R-OSGi [10]

2. Distributed Service Registry

To offer remote services transparently without changing the OSGi framework the distributed service registry is used by R-OSGi to inform other nodes about remote services. The distributed service registry uses the so called Whiteboard pattern [5], like the normal OSGi service registry. An interested node can register its interest on specific services directly at the distributed service registry. If an event concerning one of the services is available the node is informed.

To find a registering service transparently R-OSGi can install an additional bundle, which is called surrogate. This bundle listens for events concerning services in a local service registry of a node. If a service is registered which should be a remote service (specified in the surrogate bundle) the surrogate bundle informs the distributed service registry about the service, which then can inform the listeners. On the listener side R-OSGi is informed about the services. When a bundle requests a service R-OSGi generates the proxy bundle and the remote service can be used. If R-OSGi is informed about the disappearance of the service (due to a failure or unregistration of the service), it sends a bundle unload event to the local service registry. The service registry then informs the consuming bundle about the problem and the proxy bundle is removed by R-OSGi. With this technique the OSGi application must not be changed, because everything is managed by R-OSGi.

3. Transparent Distribution

The name of this technique is a little bit confusing (chosen by [10]). In the end it just means wrapping failures, which are unknown to the OSGi framework, to failures that are known to it. This is needed to make the remote calls transparent to the framework. Without doing this the framework has to be extended and aware of distribution. That means more exactly above all the network connections between nodes and failures in R-OSGi respectively on the remote side. If a failure occurs in one of the previous mentioned areas it is forwarded to the calling OSGi framework. If this concerns an unknown failure it is wrapped to a known failure by R-OSGi before forwarding it to the OSGi framework. A remote failure during a remote service call, which occurred in the network communication, is wrapped by R-OSGi into an unload event of the remote bundle which offers the called service. So the local OSGi framework is only informed, that the proxy bundle does no longer offer the service.

4. Type Injection

The solution for one big problem is still missing. Because of the distribution of applications it is possible, that a bundle on one node which offers a remote service has access to classes respectively data types that are missing on another bundle which wants to use the service. To solve this problem type injection is used. It can be seen as a kind of distributed data type system.

If the remote service expects parameters or returns a result of type(s) which are unknown to the caller, the type injection is used by R-OSGi during the generation of the proxy bundle. When the service interface and properties are sent to the requester of a remote service additionally a list of all types which are not available on the requester side. This”injection list” is created during the registration of the remote service using a static code analysis. The analysis checks only data types that are contained in the bundle, which offers the service. Data types of other bundles are ignored like all classes from java.* and org.osgi.*, which are supposed to be on every node. All found data types are saved in the”injection list”. If now another node requests the remote service the list is sent to it with the service interface and properties needed for the proxy generation on the requesting node. During the generation of the proxy all needed data types are set as exports of the proxy bundle to make them available to the bundles of the requesting node. The ignored data types from other bundles are added to the imports of the proxy bundle. This technique makes the proxy bundles self contained and offers type consistency for an OSGi framework requesting remote services.

B. Remote Service Call

To give a deeper view into calling remote services with R-OSGi the flow of an remote service call is described in this subsection. As you can see in Fig. 7, which visualizes the flow of a remote call, the starting call comes from a bundle inside the local node. Because the local OSGi framework is not aware of the distribution the proxy bundle, on which the service is called, seems to be a normal local bundle. But in reality the proxy bundle just forwards the call to the remote node which contains the bundle offering the called remote service. Therefore it uses the *invokeMethod()* method which was already explained in the previous subsection. After the remote node received the service call through the network channel between the two nodes, it looks up in a HashTable which method has to be called using the received method signature and then extracts the needed parameters for the service out of the received packet. Afterwards it calls the service with the extracted parameters using reflection [10]. The result of the service call is packed into a packet and then sends back to the proxy bundle on the local framework. In the last step the proxy bundle sends the result to the initial service

International Science Index, Computer and Information Engineering Vol:7, No:5, 2013 waset.org/Publication/5362

caller.

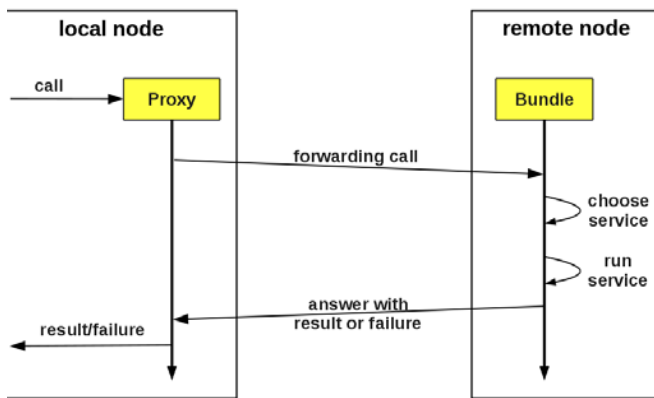

Fig. 7 Example for a remote service call
(based on description of [10])

If a failure occurs on the remote node a failure message is packet into the packet instead of the result. If a failure outside the OSGi framework occurred it is wrapped into an OSGi failure like described in the previous subsection. For this reason the remote service behaves like a local service in the view of the calling bundle.

C. Conclusion

The concept of R-OSGi is well elaborated and covers different use cases. Its usage is slightly similar to the remote services of OSGi (although it's older) but different approaches for the implementation have been used.

As already mentioned creating applications for the cloud is quite easy, due to the fact, that the developer has not to deal with the distribution. The coupling between R-OSGi and the OSGi framework is quite easy to accomplish and the usage of R-OSGi is quite easy, too. The usability is simplified by R-OSGi itself, because it pays attention to the details without interaction, like the transparency of remote services for the OSGi framework. But some people reported problems using remote services, so it seems, that there are still some bugs to fix.

The performance of R-OSGi, as already mentioned, is quite well. The network communication is slightly faster than the highly optimized Java RMI and two times faster than UPnP [10]. During the benchmark the binding time of a remote service and service invocation have been measured. Another advantage is the small footprint of R-OSGi which is just about 120kB, so it is really lightweight. ROSGi offers a high downward compatibility; it is usable since Java 1.2 or Java ME CDC.

R-OSGi is in a beta phase since January 2009, which can mean that the development of R-OSGi has been discontinued. Another problem is, that there are nearly no examples, how to other facilities available which describe or how to use R-OSGi in detail. At least there is an API online [12] and some websites like [3] offer a really short introduction to use remote services with the OSGi framework being aware of its distribution, like described at the beginning of this section. Additionally some very important features for using R-OSGi in the cloud are missing. R-OSGi itself offers neither load balancing nor transparent failover [10], what limits the application spectrum in the cloud dramatically. However the developer of R-OSGi advertised and presented a tool as an Eclipse plugin in [9], which is called R-OSGi Deployment Tool. This tool monitors a distributed application in Eclipse. It shows all nodes with their installed bundles and enables the developer to move bundles from one node to another, during the runtime of the application. In order to make this possible the tool installs an “agent bundle” on every node to get the information needed to monitor it. Additionally it adds transparent failover and load balancing to distributed applications using R-OSGi. This would be useful for using R-OSGi in the cloud, but there is a big problem. The tool is not available. It might be that the tool was never finished and therefore never published, but this is unknown. May be it will be published in the future. If this happens R-OSGi can be fully used in a cloud environment at last.

Altogether R-OSGi could be a usable approach for distributed applications in a cloud environment, if fault tolerance and load balancing are added. Because R-OSGi is no standard it is likely to be a niche product in the future. Only with leaving the beta phase and adding support to R-OSGi, which is missing so far, it could leave its niche and become a serious competition to other approaches.

VII. Comparison and Final Conclusion

The previous sections gave a detailed overview of the different approaches to enable OSGi for the cloud. Now this approach is compared to each other as far as possible using the following aspects: Similarities, usability, performance, security, current state.

- **Similarities:** The different approaches are hard to compare because they all base on different concepts and have very different stages of development (some are drafts and others are already implementations). There is an obvious similarity between OSGi*4C* and R-OSGi. Both act as a kind of middleware between the OSGi framework and the distributed bundles. OSGi*4C*, R-OSGi and OSGi Remote Services, all use a peer to peer or a peer to peer-like approach to import or export services. Additionally all concepts try to distribute the bundles transparently to the OSGi framework except OSGi Remote Services which are integrated into the OSGi framework.
- **Usability:** The guidelines that the OSGi Alliance is working on can be best practices for future implementations of both technologies. Since the Cloud API is only a conceptual paper, the usability cannot be compared in this content. OSGi Remote Services is easy to use since it is directly implemented into OSGi. This leads to only a few modifications to export or import a service remotely. OSGi*4C* and R-OSGi both don't change

International Science Index, Computer and Information Engineering Vol:7, No:5, 2013 waset.org/Publication/5362

International Science Index, Computer and Information Engineering Vol:7, No:5, 2013 waset.org/Publication/5362

the OSGi Framework itself, so they have to be added additionally to the environment of a node. One disadvantage for the usability of OSGi*4C* is the meta data. Meta data are susceptible for errors in case of wrong or not complete entries. As a result of this some dependencies can not be resolved which results in errors and not working bundles. R-OSGi instead is quite easy to use (similar to OSGi remote services). On the one hand it is possible to make normal OSGi applications distributed without changing the code. On the other hand an application can be developed knowing about its distribution, to make direct remote service usage possible.

- **Performance:** It is very difficult to make an assumptions about the performance of the conceptual papers because there is no implementation available yet. The OSGi Specifications gives no information about the performance of the OSGi Remote Services and additionally there are no performance tests known that have been performed. OSGi*4C* provides some performance tests comparing Web Services and OBR. The performance tests figured out that the peer to peer approach is very slow in comparison with the other techniques. OSGi*4C* is about two to three times slower than the OBR implementation and as nearly as fast as Web Services (which are also not very fast). But OSGi*4C* provides the advantages that there is no single point of failure like in the OBR. R-OSGi in comparison to OSGi*4C* is much faster, in a performance test it was compared with the high optimized Java RMI and UPnP. The results of the test pointed out that R-OSGi is slightly faster than RMI and two times faster than UPnP. One reason for the high performance is another field of application for R-OSGi which is embedded systems. Also the own binary protocol improves the performance of R-OSGi.
- **Security:** Except of OSGi*4C* all other approaches do not really deal with security. The developers of OSGi*4C* notice, that security is a big problem in a peer to peer infrastructure. It is possible for everyone to add a new node into the environment. OSGi*4C* provides no additional concepts to avoid this problem. A possible solution may be a signature based approach. Each client which uses OSGi applications can import a "trusted developers" list, each bundle must add a signature, if the signature is in the list the bundle is valid, if the bundle has no valid signature it will not be loaded. Over all the developers do not care a lot about security. Because Cloud Computing is a very new technology where new security problem may occur it is necessary to handle such problems.
- **Current state:** The draft of the OSGi Alliance and the Cloud API are conceptual papers. The OSGi Alliance draft is continuously in development and will supposably end up as good guidelines for the future development for OSGi in the Cloud. The Progress of Cloud API is questionable since the last release is from 2009. OSGi Remote Services are usable when using synchronous calls. They are still in current development. Asynchronous calls are not working yet. The development of OSGi*4C* is in an advanced stage. All basic functionalities are completely implemented and the advertisements and meta data is well defined and gives a good solution for automatic resolving of dependencies with some known problems. There are still some disadvantages which are described in the previous section and which have to be resolved before a commercial use is possible. It is not known if OSGi*4C* will leave the development status because the last results are published in 2009. R-OSGi is the furthest implemented approach, it is already in a beta phase. All mentioned features and techniques in the section about R-OSGi have been implemented already. But there are still some bugs to fix, some people reported problems, which sometimes occur using remote services. To help the developers an online API of all classes of R-OSGi [12] is available. Like the other approaches by the other institutions the last update was in January 2009, which means leaving the beta status is questionable.

As is recognizable from the comparison there are serious differences between the different approaches. For that reason a comparison is hard to do. But everything common is that the approaches are all very sophisticated and contain different concepts to make OSGi available in the cloud. At the moment no approach is free of problems.

The draft of the OSGi Alliance contains guidelines which should be followed, but only the time can show if this guidelines, which are still under development, get used in the future. The cloud API (CAPI) has another problem. It is addicted to the companies if they are supporting this approach or not. For this reason it may be doubted, that CAPI will succeed. OSGi remote services in contrast are likely to be used in future, because they are part of the OSGi framework itself. The problem is that the remote services are implemented just partly at the moment; the asynchronous part is missing [13]. The problem of OSGi*4C* is that there are some problems in the architecture. Especially the complete rely on meta data may be a problem for a commercial use. R-OSGi offers as previously mentioned no load balancing and no fault tolerance, for which reason the exertion in the cloud is questionable. Only if R-OSGi is extended with these features it could be used in the cloud seriously.

ACKNOWLEDGMENT

Irina Astrova's work was supported by the Estonian Centre of Excellence in Computer Science (EXCS) funded mainly by the European Regional Development Fund (ERDF). Irina Astrova's work was also supported by the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12.

We would like to thank Sören Appel from the University of Applied Sciences and Arts Hannover, Germany, for his help

in preparing this paper.

REFERENCES

[1] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In In Adaptable and extensible component systems, 2002.
[2] B. Costa, M. Matos, and A. Sousa. Capi: Cloud computing api. inforum.org.pt/INForum2009/docs/short/paper 63.pdf.
[3] S. Diener. Tutorial: How to create a remote osgi service. http://stefan222devel.blogspot.com/2008/11/how-tocreate- remote-osgi-service.html, Nov. 2008. (last visit February 2011).
[4] K. Knoernschild. Patterns of modular architecture get ready today for the platform of tomorrow! http://www.kirkk.com/modularity/chapters/.
[5] P. Kriens and B. Hargrave. Listeners considered harmful: The whiteboard pattern. http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf, Aug. 2004.
[6] OSGi Alliance. Rfc-0112 bundle repository. http://www.osgi.org/download/rfc- 0112 BundleRepository.pdf.
[7] OSGi Alliance. RFP 133 Cloud Computing, March 2010. Revision 0.6.
[8] J. S. Rellermeyer. Services everywhere: Osgi in distributed environments. http://www.eclipsecon.- org/2007/index.php?page=sub/&id=3661, Mar. 2007.
[9] J. S. Rellermeyer, G. Alonso, and T. Roscoe. Building, deploying, and monitoring distributed applications with eclipse and r-osgi. In Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange, eclipse '07, pages 50–54, New York, NY, USA, 2007. ACM.
[10] J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-osgi: distributed applications through software modularization. In Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware, Middleware '07, pages 1–20, New York, NY, USA, 2007. Springer-Verlag New York, Inc.
[11] J. S. Rellermeyer, M. Duller, and G. Alonso. Engineering the cloud from software modules. In Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing, CLOUD '09, pages 32–37, Washington, DC, USA, 2009. IEEE Computer Society.
[12] J. S. Rellermeyer et al. R-osgi remote service 1.0.0.rc4 api. http://r-osgi.sourceforge.net/remote/apidocs/index.html, Jan. 2009. (last visit February 2011).
[13] M. Schaaf. Extending osgi by means of asynchronous messaging. Master's thesis, Fachhochschule Hannover, Sept. 2009.
[14] R-osgi. http://r-osgi.sourceforge.net.
[15] H. Schmidt, J.-P. Elsholz, V. Nikolov, F. J. Hauck, and R. Kapitza. Osgi*4C* enabling osgi for the cloud. New York, NY, USA, 2009. ACM.
[16] The OSGi Alliance. Osgi service platform enterprise specification. http://www.osgi.org/Download/Release4V42, Mar. 2010. Release 4, Version 4.2.
[17] jxta. http://de.wikipedia.org/wiki/JXTA.

International Science Index, Computer and Information Engineering Vol:7, No:5, 2013 waset.org/Publication/5362