

D-OSGi: Supporting the OSGi Middleware with Service Distribution

(Working Paper)

Abdelgadir E. Ibrahim , Liping Zhao ^{*†‡}

07 March 2007

Abstract

The Open Services Gateway Initiative (OSGi) defines an open service platform standard for service delivery, composition and execution in networked environments. The current OSGi application model is limited to a single Java Virtual Machine; it does not support the distribution of services across OSGi frameworks and devices. The OSGi Enterprise Experts Group (EEG) recently identified a number of desired extensions to the OSGi application model including a distribution extension but has not yet defined a specification for such extensions. This paper clarifies and defines a terminology for OSGi distribution. It then proposes the publish-subscribe interaction model as basis for OSGi distribution extensions. The paper describes the D-OSGi bundle which uses the publish-subscribe communication paradigm for supporting the OSGi platform with mobility and distribution of services. The paper concludes with a specification outline for OSGi distribution extensions.

1 Introduction

The Open Services Gateway Initiative (OSGi) defines an open service platform standard [1] for service delivery, composition, and execution in networked environments. The OSGi service platform provides a dynamic environment where new components (bundles) can be dynamically inserted into and removed from this environment without the need to restart the OSGi platform or the application system. The OSGi service platform current application areas include: home services gateways, connected homes, industrial automation, mobile environments, automotive, desktop, and server applications.

The OSGi application model is restricted to a single Java Virtual Machine (JVM); it supports the dynamic download, deployment, and management of bundles within a *single* JVM. The OSGi Enterprise Experts Group (EEG) has recently identified a number of desired extensions to the OSGi application model which include a *distribution* extension. The EEG states there is a huge demand for the ability to access remote functionality. Support for distributed services will scale the OSGi platform to multi-process and multi-container environments. Although release 3 of the specification had already defined two standard services for interaction with UPnP and Jini networks, these have some limitations as described below. The standard OSGi Jini service was later withdrawn from the latest release 4 of the OSGi specification.

^{*}This work is supported by the Engineering and Physical Sciences Research Council (EPSRC) and NXP Semiconductors (formerly Philips Semiconductors).

[†]Abdelgadir E. Ibrahim and Liping Zhao are with the School of Computer Science, University of Manchester, Kilburn Building, Oxford Road, Manchester, M13 9PL, United Kingdom. E-mails: (abdelgadir.ibrahim, liping.zhao)@manchester.ac.uk. +44(0)1613063744.

[‡]©Abdelgadir E. Ibrahim and Liping Zhao 2007

To summarize, the aim of an OSGi distribution extension is to facilitate the construction of OSGi applications that span multiple OSGi frameworks, multiple JVMs, and multiple devices. However, the OSGi Alliance has not yet defined a specification for OSGi distribution.

In this paper, we discuss the use of the publish-subscribe communication paradigm for OSGi service distribution. Publish-subscribe is a communication and interaction model that has become very popular recently because of its loosely coupled nature and other benefits it brings. The publish-subscribe interaction style has been also used in many different application domains including monitoring, awareness, enterprise integration, and groupware. It is hoped the work described in this paper will contribute to a specification for OSGi service distribution.

The paper contributions can be summarized as follows:

- It clarifies and defines a terminology for OSGi distribution.
- It proposes the publish-subscribe interaction model for OSGi distribution. To this end, the paper describes the D-OSGi bundle which adopts a publish-subscribe interaction model for supporting the OSGi platform with mobility and distribution of services.

The rest of the paper is organized as follows. Section 2 provides an overview of the OSGi application model. Section 3 describes related work. Section 4 defines a terminology for OSGi distribution. The R-OSGi extension is then evaluated in Section 5. R-OSGi is a related research project that aims to support the OSGi platform with capabilities for distributed services. Section 6 introduces and describes the D-OSGi distribution extension while Section 7 provides an evaluation of D-OSGi and describes the research findings. Finally, Section 8 concludes the paper.

2 Overview of the OSGi Application Model

The OSGi service platform consists of two elements: the OSGi framework and a set of *standard* service definitions. The OSGi framework is a lightweight Java-based container for deploying and executing *service-oriented* applications. The OSGi framework defines a component model, a services registry and provides the runtime environment for handling the interactions between services and between components. In addition, the OSGi framework supports the remote management of the entire application lifecycle including dynamic on-the-device software deployment and extension. The OSGi standard service definitions are optional and can be implemented by different vendors for inclusion in any given solution. These standard services include among many others: an HTTP service, a Logging service, an XML Parsing service, a UPnP Device service, and a Configuration Admin service.

The OSGi application model combines the two approaches of component-orientation and service-orientation resulting in what is known as a Service-Oriented Component Model. In this model, a software application is viewed as a set of collaborating components that provide and use services to and from each other respectively. Component collaboration follows the service-oriented interaction pattern where services provided by components are published into a registry. Client components can then dynamically discover and bind to those published services.

A component in the OSGi service platform is known as a *bundle*. Bundles are the unit of delivery and deployment. The OSGi platform supports the dynamic download, deployment, and management of bundles within a *single* Java virtual machine. Within the service platform, a service is represented as a Java interface which provides for many implementations of the same service to exist simultaneously. A unique and innovative characteristic of bundles is that they can share Java packages i.e. code, between each other. The service platform defines the mechanism for sharing code and manages it automatically; a process that is known as package dependency resolution. The OSGi component model is a powerful model that overcomes many of the limitations of other existing component models such as lack of support for

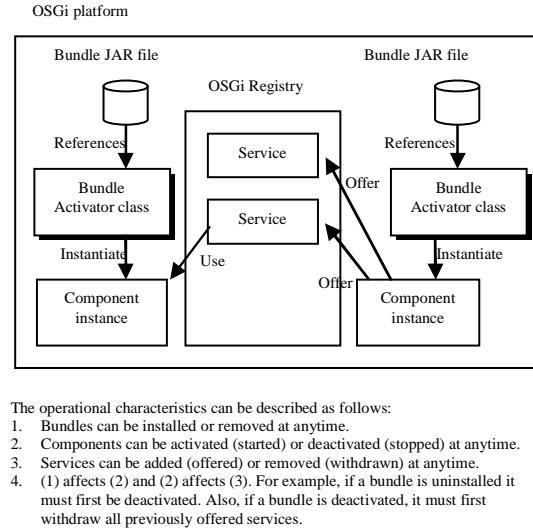


Figure 1: OSGi Application Model

explicit dependencies, type safety and module safety [2]. In addition, the OSGi platform provides support for installing, uninstalling, activating (starting), deactivating (stopping), updating and refreshing of bundles. All these activities can be performed either locally or remotely. Together, they also define the bundle's life cycle.

Physically, a bundle is represented as a Java JAR file that contains all resources required for the operation of the bundle as well as specifications of its dependencies. In addition, this JAR file also contains a *manifest* file that describes the content of the JAR and provides information about the bundle. The manifest file is comprised of a set of standard manifest headers that are processed by the platform to correctly install and activate the bundle. One of these headers is the *Bundle-Activator* header. This describes the name of a Java class file referred to as the *bundle activator class* which is used to activate the bundle after it had been installed.

After installation, a bundle must be first resolved before it can be activated (started). The resolving process is automatically performed by the framework to ensure that package and environment dependencies are satisfied. This involves processing the bundle's manifest for declared dependencies on the environment or on Java packages (i.e., code) that are provided by other bundles. If such dependencies exist, the framework verifies whether these are available and accessible to the declaring bundle. Once resolved, a bundle can then be activated. Activation involves creating an instance of the bundle activator class and calling its start method. Once activated and through the framework registry, a bundle can then publish, discover and bind to available services. If deactivated, the framework calls the stop method in the activator class. The bundle must then withdraw any services it previously published and release any services it is currently using. Fig. 1 illustrates this OSGi application model.

3 Related Work

3.1 OSGi-Jini and OSGi-UPnP Interactions

Jini is a technology for devices interaction that adopts a service-oriented paradigm. A Jini service provider advertises a service (a normal java object implementing a user defined service interface) with a lookup server. A Jini client searches the lookup server for a required service either by its API, attributes or ID. Jini discovery is the process of locating the lookup server. Once a service is obtained, the Jini infrastructure automatically downloads all the code for interacting with the service. The actual communication protocol with the offered service can use any type of networking technology including, for example, Java Remote Method Invocation (RMI) and SOAP messages. Although the communication protocol with services is flexible, the Sun Jini implementation, as pointed by [3], does require RMI for communication with the lookup server.

UPnP technology allows a device to discover other devices in an UPnP network and to configure them and control their operations using open internet standards like HTTP and XML. It therefore enables data communication between any two devices under the command of any control device on the network i.e., it facilitates device interoperability.

Release 3 of the OSGi specification defined a standard service for OSGi-Jini interaction. The basic operation of this service adopts an import/export mechanism [1] where a Jini service is imported into the OSGi platform thus making it accessible to OSGi services and bundles. Conversely, an OSGi service can be exported to a Jini network thus making it accessible to other Jini peers within the network. According to [3], there are three major limitations with the Jini approach to OSGi cross-device interaction:

- Incompatibility of the service attributes data types complicates the process of searching and selecting among imported Jini services and prevents the use of the powerful OSGi filtering support. Mappings could eventually be supported but their implementations proved to be non-straightforward and invasive [3].
- If a service implements multiple interfaces and once it is registered with the lookup server, it becomes accessible under all those interfaces. This is in contrast to OSGi services which can be published under a subset of their implemented interfaces making it possible to limit service accessibility to those published interfaces.
- The requirement for the lookup server limits its applicability in ad hoc networks and mobile environments.
- RMI should be avoided because it is not supported by all devices. Also, RMI violates OSGi generality where an OSGi platform is expected to work on many different devices as defined by the OSGi supported execution environments. Furthermore, OSGi bundles and services have a well defined life cycle and subsequently RMI features such as distributed garbage collection add unnecessary overhead.

The OSGi-UPnP interaction follows the same import/export approach described above. The author in [3] identifies the following limitations to the OSGi-UPnP device interactions:

- Interaction with imported UPnP services can only be done using a subset of the standard SOAP data types (a limitation of the UPnP standard itself). Subsequently, this prevents interaction between devices using complex java objects as invocation parameters.
- According to the UPnP specification, you can either subscribe to all events emitted by an UPnP device or none. In other words, it does not support selective subscriptions for receiving particular events.
- The SOAP communication protocol significantly degrades performance especially in comparison with other binary protocols.

3.2 R-OSGi

R-OSGi is a research project at the Swiss Federal Institute of Technology (ETH) that aims to provide a lightweight solution to distributed OSGi services. R-OSGi is sub-component of the FlowSGi project (<http://flowsgi.inf.ethz.ch/>) which aims to support the concept of *fluid computing* as an approach to manage the synchronization of data and state between collaborating federations of devices [4]. In effect, this makes it possible to move applications and data between devices as required. FlowSGi uses the OSGi platform as the underlying middleware. R-OSGi proposes to extend the OSGi platform so that multiple framework instances can behave as a single one. R-OSGi extends the OSGi platform with support for distribution by introducing *remote services* and *remote events*.

3.2.1 Philosophy for OSGi Distribution

In order to avoid the drawbacks and overcome the limitations of the Jini and UPnP interaction approaches, the key requirements for an OSGi distribution solution are [3]:

- *Transparency.* Remote services must be accessible as if they were local; individual framework instances should not be aware of peers or of the federation.
- *Non-invasiveness.* The solution should be non-invasive; it should be possible to enable existing OSGi services for remote access without requiring any modifications.
- *Consistent Behaviour.* Interaction with remote services should not be different from interaction with local services.
- *Spontaneous Interaction.* It should be possible to federate a set of OSGi framework instances spontaneously. No pre-configuration or special provisions should be needed.
- *Statements of Supply and Demand.* To allow for large scale networks, the solution should support *selectivity* with respect to federation of framework instances. In particular, “individual framework instances should not be overwhelmed by the number of available services [3]”.
- *Generality.* The OSGi platform is being used in a variety of domains and devices ranging from mobile devices to enterprise servers. Therefore, the solution should be lightweight so that it can be used in all areas and devices supported by the OSGi platform.

The operations of R-OSGi are detailed in [4] [3]. In the following section, we highlight its approach.

3.2.2 R-OSGi Overview

In order to facilitate spontaneous interactions, R-OSGi makes use of the Service Location Protocol (SLP) to discover and advertise remote OSGi services. Using SLP, a device is able to search its environment for advertised services. The motivation behind using the SLP protocol for service advertising and discovery is described in [3].

On the service offering device, and in order to avoid scalability and security issues, remote OSGi services have to be explicitly marked for remoting (by setting a remote service property). This defines the statement of supply.

On the client device, interest on a remote service is expressed by registering a listener with the OSGi registry. This listener defines the type and properties of the required service. R-OSGi is then responsible for searching the environment and calling back the listener when a matching service is found. Registration of discovery listeners forms the statement of demand. Once a match is established, the client at its discretion can decide whether a network connection to the service offering device should be established.

Connections are manifested as one-to-one network connections between offering and client devices. By default R-OSGi uses TCP channels but this is customizable. For example, R-OSGi can be configured to use HTTP channels. Once a connection is established, peers exchange their statements of supply (offered services) and the list of event topics in which they are interested. Furthermore, it is possible to bypass the discovery mechanism and connect directly to known service providers. This is useful in large scale distribution where service discovery is not feasible.

After the connection is established, a client can obtain (fetch) a remote service. The R-OSGi runtime automatically determines and transmits all types that are required to make the service work at the client side. This is analogous to Jini code downloading. At minimum, only the service interface and its attributes are transferred to the client peer. Once transferred, The R-OSGi runtime automatically generates and installs a proxy bundle on the fly for integrating the remote service with the local client's framework. Once installed and started, the proxy bundle offers a proxy service with the client's local framework registry. Behind the scenes, every call to the local proxy service is translated into a remote procedure call to the remote service on the offering peer.

For efficiency, these remote calls are modelled using R-OSGi specific messaging protocol although they behave just like any call to a local service. This is an intentional design decision in order to avoid using RMI for the reasons described above (see 3.1).

So far, the described approach adopts a client-server interaction style where all calls to the proxy service are propagated to the remote offering peer, invoked there, and results returned to the client peer. This is not always required or necessary. Therefore, R-OSGi also allows partial shipping of code so that some of the remote service's functionality can be executed on the client side (concept of smart proxies). A smart proxy is a normal abstract java class that is transferred to the client peer as part of the exchanged service properties. All abstract methods of this abstract class are treated as remote method calls and invoked according to above description. All non-abstract methods are treated as local and invoked on the client peer.

Remote events are also supported by R-OSGi. It forwards events between peer devices.

3.3 Shared Spaces and the Lime Middleware

This paper proposes the publish-subscribe interaction style as basis for OSGi distribution. The current D-OSGi bundle implementation described in Section 6 uses the Lime (Linda in Mobile Environment) shared space coordination middleware to enable the transparent sharing of services across OSGi enabled devices. 'Shared spaces' is a variation of the publish-subscribe communication paradigm in which multiple processes running in multiple hosts communicate through operations on a shared data space [5]. The same concept also applies in single host scenarios where multiple processes in a single host are able to concurrently access the shared space.

The Linda coordination model [6] was the first work to introduce the concept of a *tuple space* as an abstraction for accessing the shared space. A tuple space is a multi set of tuples which are basic data structures consisting of a sequence of typed fields, for example ("Adam Smith", "01/01/1980", 7, 45.5) is a tuple [7]. Some modern implementations also support named fields where it is possible to associate each field with a unique name that can be later used to retrieve the field. The tuple space is equally accessible, possibly concurrently, by all hosts or processes.

In the next section, we describe the Lime middleware which adapts the Linda shared space coordination model to mobile environments that are characterized by both physical and logical mobility. In physical mobility it is the hosts (devices) that move around while in logical mobility it is the processes (mobile agents) that exhibit mobility across different hosts. This description is based on the discussion in [7].

3.3.1 Lime: A High Level Overview

The problem to be tackled is that especially in mobile settings characterized by physical mobility, there is no stable, always accessible, place to store the shared data [7]. For example, if a centralized architecture is adopted, devices will lose connectivity to the central server as they move in and out of range. The problem is compound in ad hock networks because connectivity itself comes and goes.

The core and innovative idea behind Lime can be stated as transparent context maintenance [7]. The shared data represent the context for the accessing processes or hosts. Lime adapts this fixed context to the mobile settings by partitioning the shared tuple space into a number of tuple spaces each permanently and exclusively associated with a single process [7]. Lime refers to this individual tuple space as the interface tuple space (ITS). It represents the single point of interaction for the owning process. Processes use normal tuple space operations to add and retrieve tuples to/from this tuple space. The data in this local tuple space is the only data available to the owning process while it is alone [7].

When multiple processes (or hosts) are involved, they form a Lime group. Groups can be either host level (when multiple processes are running in a single host) or federated (when multiple hosts are able to communicate with each other). In this latter case, all the processes in all the hosts involved, form a single logical group spanning the multiple hosts. Group formation is handled and managed transparently by the lime middleware. Lime transiently shares the ITS content of each individual process to form a single shared tuple space. Each process is able to interact with the shared space using its own ITS. As group membership changes, Lime automatically adjusts the content of the shared tuple space to accommodate the changes. These changes are then transparently reflected by the individual ITS so that the owning process is kept up to date with the status of the shared space.

In Lime, engagement and disengagement refer to the effects of joining and leaving a group respectively. Engagement triggers the automatic sharing (inclusion) of the tuples present in the joining process's ITS. Disengagement triggers the automatic un-sharing (removal) of the tuples belonging to the departing process's ITS. Lime also permits a process to own multiple ITS as long as they are given different names. In this case, only ITS with identical names are able to form a group.

The transient sharing of tuple spaces is determined by connectivity; only tuple spaces of connected processes are shared. Lime considers two processes as connected if they reside in the same host or if there is an available communication link between the process's hosts [7]. Availability may depend on many different factors including: the quality of connection, security issues, and cost of connection as few examples [7].

Lime supports a style of coordination that “reduces the details of distribution and mobility to content changes in what is perceived to be a local tuple space ... this has the potential to greatly simplify the application design in many scenarios by relieving the designer from the chore of explicitly maintaining a view of the context that is consistent with changes in the configuration of the system” [7].

4 OSGi Distribution: Back to Basics

The process of distributing OSGi services so that they become accessible from remote locations can be considered in its roots as an exercise in *code mobility*. This can be defined informally as “The capability to dynamically change the bindings between code fragments and the location where they are executed [8]”.

The authors in [8] introduce the concepts of: *Computational Environment (CE)*, *Execution Unit (EU)*, and *State*. CE is an abstraction for entities that provide the relocation capabilities and which maintain the identity of the host in which they reside. EUs represent the entities which are hosted by the CEs. Resources are the entities that are needed for EUs to perform their functions. State is comprised of a *data space* and an *execution state*. The former represents the configuration data required by the EU while the latter represents the runtime data of the EU such as its runtime execution stack. In mobile code systems, all of the code, resources, execution state, and the data space of an EU can be relocated to a different CE across the network.

To this end, the authors in [8] distinguish between strong mobility and weak mobility. In strong mobility, all the code, data space, and execution state of the EU can be relocated. In weak mobility, only code can be transferred. Weak mobility may also involve the transfer of the EU's initialization or configuration data but the execution state is not transferred.

Migration and remote cloning are the two mechanisms for realizing strong mobility [8]. When migrating an EU, it is first suspended, if applicable, physically disconnected and transferred before execution is started/resumed at the destination CE. Remote cloning involves the spawning of a copy of the EU on the remote CE. When an EU is being transferred to a remote CE, its code can either be fetched by the destination CE from the source CE, or alternatively it can be shipped by the source CE to the destination CE (direction of transfer) [8]. Furthermore, the transferred code can be either self contained or a code fragment [8]. Self contained code can be used to independently instantiate and execute the EU at destination. Code fragments must be linked with other code residing at the destination CE before they can be used.

In [8], the authors model a resource as a triple (I, V, T), where 'I' is the resource identifier, 'V' is its value, and 'T' is its type. Resource types determine the transferability of the resource. For example, a resource of type *Hardware Device* is usually not transferable while a resource of type *Information* is transferable. Even if a resource is transferable, it can be designated as either fixed (disabled transferability) or free (enabled transferability). The transferability status (fixed/free) of transferable resources is determined by the developer on the basis of application requirements such as performance considerations and the nature of the resource. For example, data resources deemed sensitive or confidential may be marked as fixed.

When an EU is transferred (either by migration or remote cloning), the resources it uses may also need to be transferred along with it. This depends on the nature of the resource and the type of link (binding) between the EU and the resource [8]. In addition, different applications may have different requirements with respect to resource transferring. There are essentially three possible types of bindings between an EU and a resource: *by identifier*, *by value*, and *by type* [8]. Binding by value indicates that the EU always requires the same resource instance for its execution. Binding by value indicates that only the type and value of the resource matters. Binding by type indicates that only the type of the resource matters (irrespective of the value).

The authors in [8] summarize the problems of EU migration as follows:

- *Resource relocation.* The question is whether and how the required resources are migrated?
- *Binding reconfiguration.* The question is how can the binding between the EU and the resource be re-established after the EU is migrated?

Depending on the nature of the binding, the authors in [8] identify the following strategies:

- *Migration by move.* Here, the resource is transferred along with the EU. In other words, the binding is not modified. However, this solution can only be employed when the resource is both transferable and free.
- *Use of network references.* Here the resource is not migrated along with the EU. At destination, the EU establishes a remote reference back to the resource residing in the source CE. This solution is useful when the resource is either not transferable or is fixed. However, performance issues and potential network failures may affect the operations of the EU.
- *Migration by copy.* In this case, a copy of the resource is packaged along with the EU. At destination, the EU rebinds to this copy.
- *Rebinding.* Here, when the EU is migrated, it simply searches for a resource of the same type. If found, the EU rebinds to this found resource.

At the application design level, the authors in [8] distinguish between: *code components* (the know-how), *computational components*, and *resource components* (data or devices). Based on these concepts, the same authors identify the following paradigms of mobile applications:

- Client-server. Clients request the services offered by a potentially remote server. The server hosts both the know-how and the resources required for providing the service.
- Remote evaluation. A computational component may know how to perform a particular task but lacks the resources. In this case, it may choose to migrate to a remote location where the resources are thought to be available. Once there, the computational component performs the task and then delivers (or brings back) the results to the original location.
- Code on demand. A computational component may have access to resources required for a specific task but lacks the know how to manipulate these resources. In this case, it may request such know how (the code) from a remote location which is then delivered.
- Mobile agents. A computational component may possess the know-how and some of the required resources. It starts execution on the source location. When additional resources are required, it migrates, along with the code, data space, execution state and the intermediate results, to a remote location where the extra resources are available. Once there, the computational component simply resumes executions.

It must be noted that these different paradigms should be evaluated by developers on application-by-application basis according to application requirements since no single paradigm is optimal for all applications [8]. Furthermore, the actual technology to implement the chosen design paradigm must be evaluated on individual basis because some technologies are more suited for certain design paradigms [8]. See [8] for further details.

4.1 Terminology for OSGi Distribution

In this section, we map the above mobility terminology to OSGi concepts and entities.

An OSGi framework instance represents the CE whereas bundles are the EUs. A bundle may use a set of resources. These resources include for example: required bundles, imported packages, native libraries, required execution environment and other contextual dependencies. Configuration information can be considered as the bundle's data space. Basically, a resource is any entity required by the bundle in order for it to be successfully deployed and to perform its functions. The aim of a distributed OSGi extension is to support relocation of bundles across OSGi framework instances. Note that although the current OSGi specification defines mechanisms for the download of bundles from remote locations, this can only be done in a single direction (fetching). Furthermore, statefull bundle relocation is not supported.

Migrating a bundle means that it is uninstalled from the current framework instance, physically transferred to a remote framework instance, and then reinstalled and started in that destination. Migration can be either stateless or statefull. Bundle cloning means a *copy* of the bundle is installed and started in, a potentially remote, framework instance. For remote cloning, a bundle can either be replicated or non-replicated. Replication implies that the original bundle remains active in the source framework instance.

At a different level of granularity, it may be required to support the mobility of individual services as opposed to whole bundles. Service migration means the service is withdrawn from the current framework and re-offered in a different framework instance. Remote service cloning means a copy of the service is offered with a different framework instance (either replicated or non-replicated).

Bundles and services can be transferable or non-transferable. For transferable bundles and services, they can be further marked as either free or fixed.

At the application design level, a client-server paradigm means that the bundle or service and all required resources are hosted by the source framework instance. Clients in remote framework instances can access these services using a form of remote procedure calls (irrespective of the actual implementation technology).

When remote evaluation is employed, it means that the bundle or service is physically migrated/cloned in a remote framework instance where execution takes place. Results can then be delivered, or brought back along with the mobile entity, to the source framework instance.

A code on demand design paradigm means that the bundle or service can request the download of required code from a remote framework instance. Once downloaded, it can then be linked with existing code and executed.

A mobile agent design paradigm means that it is possible for a bundle or service to start execution in one framework instance and finish it in a different instance.

As mentioned previously, support for these mobility strategies depends on the nature of the application, its domain, and its requirements. Hence, not all the above mobility strategies may be necessarily supported by an OSGi distribution extension.

5 Evaluating R-OSGi

R-OSGi is a research project that aims to support the OSGi platform with capabilities for distributed services. It represents the first such effort for supporting distributed OSGi services. Therefore, we feel it is important to critically evaluate its approach and identify its strengths and weaknesses. In the following sections we revisit the fundamentals of distributed OSGi services in light of the discussion in section 4, and describe how they are addressed by R-OSGi.

5.1 R-OSGi Mobility Features

This description and evaluation is based on R-OSGi version 0.6.0.

- *Mobility model.* R-OSGi supports weak mobility of bundles and services. For bundle mobility, the bundle's raw bytes are read from the corresponding class files and installed in a remote framework instance. R-OSGi does not distinguish between migration and remote cloning. For service mobility, a modified copy of the service is offered with a remote framework instance.
- *Design paradigm.* At the application design level, R-OSGi supports the client-server paradigm. When the service copy is offered with the remote framework instance, it forwards all calls to the original service in the source framework. A limited form of remote evaluation is supported via the previously mentioned concept of a smart proxy (see section 3.2.2).
- *Resource relocation.* R-OSGi supports a *resource rebinding* strategy. An OSGi service is an object implementing a service interface. In other words, bindings to such services represent a binding by type. Therefore, rebinding seems to be the most appropriate resource relocation strategy. Using rebinding strategy, a mobile entity can simply search for and rebind to a service of the same type that is present at destination. Similarly, imported packages and other contextual dependencies can be simply re-imported at the destination peer. If required resources are missing in the destination, then relocation fails.
- *Interaction style.* R-OSGi adopts point-to-point communication style where it maintains a separate channel between peers for every remote service. This point-to-point and synchronous interaction style results in rigid and static applications and is not suited for dynamic large scale setups [5]. The publish-subscribe interaction style is widely considered as more appropriate communication model especially for mobile environments (see for example, [5] [9] [10]).

In certain applications, resource relocation by move or copy may be a more appropriate strategy. For example, consider a stock analysis service which requires access to a rate data feed resource. In this case, relocation by copy or move may be more appropriate especially if the resource is not available at destination. Alternatively, if a used resource represents a hardware device, then use of a network reference strategy is the only viable option (unless a similar resource is assumed to be available at destination).

In summary, different applications may have different requirements. Therefore, a distribution extension must be flexible enough to support the different mobility models and strategies based on application requirements (customizability). R-OSGi lacks such customizability. For example, it is not possible to vary the design paradigm or the resource relocation strategy on individual application basis.

5.2 R-OSGi: Lessons Learned

5.2.1 Calculating the Transferred Code Fragment

If a bundle distribution policy is specified (i.e., developer wishes to transfer the whole bundle), R-OSGi reads the bundle's bytes which is then transferred. If a service distribution policy is selected (developer wishes to transfer only the specified service), then R-OSGi automatically calculates the transferable code fragment by analyzing the service interface(s). For each implemented interface, R-OSGi automatically determines all the classes that must be transferred in order to register and operate the service at the remote peer under the given interface. This calculation is based on the specified injections, smart proxy, and the bundle's imported packages. Developers can also optionally specify named classes that are considered as *injections* and which will be transferred along with the service.

This automatic calculation of the code fragment is very powerful, useful, and simplifies application development. However, as mentioned before, R-OSGi offers no customizability with respect to resource relocation. For example, it is not possible to include additional serializable objects for use at destination.

We suggest that automatic calculation of the transferable code fragment should be a *mandatory* requirement for a distributed OSGi extension. However, some degree of customizability is also desired as explained above.

5.2.2 Proxy Generation

At the destination framework instance, R-OSGi uses byte code manipulation to automatically generate a *proxy bundle* for integrating the transferred code fragment. Once created, R-OSGi installs and starts this proxy bundle which then offers a *proxy service* with the local framework. All interactions with this local proxy service are forwarded to the original service at the source framework instance (with the exception of the abstract methods of the smart-proxy if specified).

Question. Is this proxy bundle approach necessary? Is it possible to transfer the service, generate the *proxy service* at destination, and offer it with the local OSGi registry without having to create, install, and start a proxy bundle first?

Answer. When a service is transferred, R-OSGi includes all interfaces and classes required for this service to operate at the destination framework. However, the service may be using interfaces and classes from different bundles (imported packages). R-OSGi does not transfer classes and interfaces from these imported packages for performance reasons. In addition, these imported classes may already be available at destination in which case they can be simply re-imported. Furthermore, R-OSGi must ensure that these package and class dependencies are satisfied at destination. The OSGi specification defines a dependency resolution process which ensures that package and other environment dependencies are satisfied. This dependency resolution is automatically performed by the OSGi framework after a bundle is installed but before it is started. This means that at destination, there is a need to integrate the transferred service at the *module layer*. The only mechanism available in OSGi for triggering this resolution process is by installing and starting a bundle

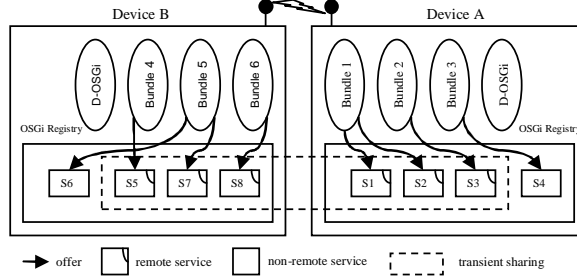


Figure 2: D-OSGi Overview

which results in the resolving of that bundle. Therefore, the proxy bundle approach seems to be the only way available for achieving such integration.

6 The D-OSGi Extension

6.1 D-OSGi Overview

Fig. 2 illustrates the operations of the D-OSGi extension. The D-OSGi bundle runs in the background like any other normal OSGi bundle. Bundles wishing to offer services that are enabled for remote access explicitly set a corresponding `REMOTE_CANDIDATE` property of the service; in effect marking the service for remote access. The D-OSGi bundle automatically and transparently exports the remote candidate service so that it becomes accessible to all OSGi framework instances that are connected i.e., that are within communication range. Conceptually, this results in the formation of a logical federated OSGi registry. This means bundles residing in different framework instances are able to interact as if they were all residing in a single framework. The individual bundles are unaware of the federation; they simply interact with their local registry in the normal way. For example, in Fig. 2, Bundle2 is able to use a remote service 'S7' that is offered by Bundle5 as if Bundle5 was residing in the same 'Device A'. D-OSGi does not trick Bundle2 into thinking that S7 is a local service because certain considerations such as performance and security may be important for Bundle2. Instead, D-OSGi automatically associates S7 with a `REMOTELY_IMPORTED` property before presenting it to Bundle2 which can then decide whether and how to use the service S7. As new bundles and devices join and leave, the federated logical registry is automatically adjusted to reflect the changes. A bundle joins when it is started in its local framework. A device joins when it becomes within communication range. A device join implies the joining of all started bundles within that device. Conversely, a bundle leaves when it is stopped and a device leaves when it becomes out of communication range. Finally, services that are not marked for remoting are unaffected by the D-OSGi extension.

6.2 D-OSGi High Level Design

Fig. 3 illustrates the high level design of the D-OSGi extension. Bundle1 and Bundle2 offer two remote candidate services S2 and S3 respectively. In the exporter device, The D-OSGi bundle runs in the background monitoring the local OSGi registry. As soon as these remote candidate services appear, they are exported by D-OSGi for remote access. This entails the mapping of the service to a tuple that is inserted in a local tuple space. Behind the scene, Lime is used to transiently share the tuple spaces of connected devices. At the client device, the D-OSGi extension continuously monitors the local tuple space to import the previously exported tuples. This entails the mapping of tuples to proxy services that are registered with the local

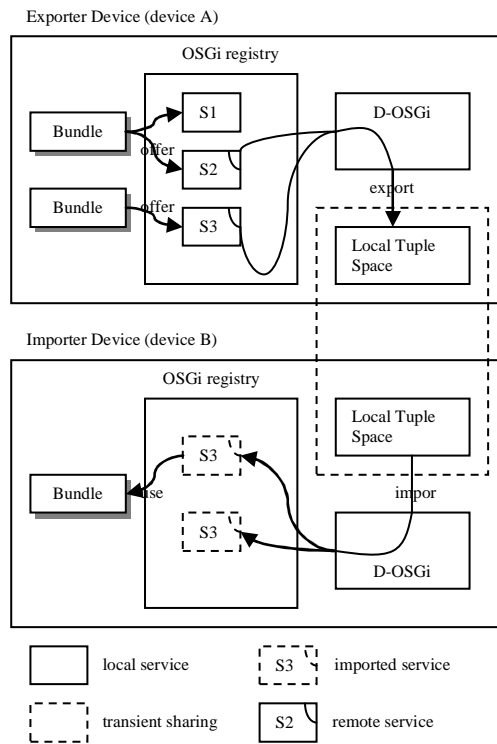


Figure 3: D-OSGi High Level Design

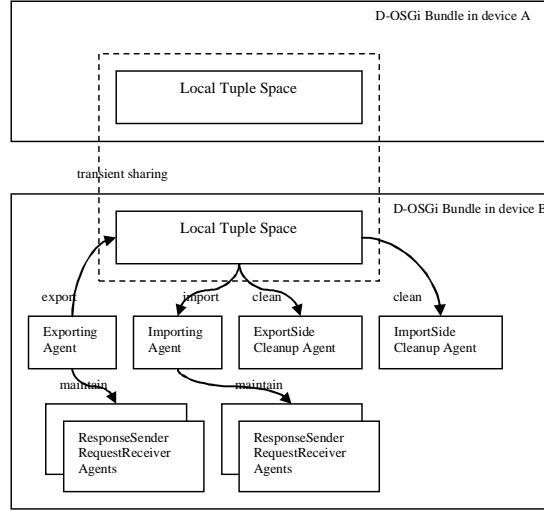


Figure 4: D-OSGi Components

OSGi registry. All calls to these proxy services are forwarded to the original services in the exporting devices. The forwarding mechanism also uses the local tuple space to implement remote procedure calls by means of messages. Method invocations on the proxy services are mapped to tuples that are inserted in the local tuple space. At the corresponding exporter device, these tuples are picked up and translated into method invocations on the corresponding services. Invocation results are then forwarded back using the same mechanism.

6.3 D-OSGi Components

Fig. 4 shows the main components of the D-OSGi extension. A D-OSGi bundle consists of four main *stationary* agents which are described below.

6.3.1 Exporting Agent

This is a singleton (per D-OSGi bundle) agent that is responsible for exporting remote candidate services to reachable peers. It monitors the local OSGi registry and when it detects the registration of a remote candidate service, it automatically maps it into an **ExportTuple** that is inserted in the local tuple space. Similarly, when it detects un-registration of a remote candidate service, it automatically propagates this information to reachable peers. It achieves this by inserting an **UnExportTuple** in the local tuple space. This tuple is then picked up by peers and the appropriate action will be taken (see **ImportSideCleanUp Agent** below). In addition, when this agent detects property updates of a remote candidate service, it automatically inserts a **PropertiesUpdateTuple** with the local tuple space. Again this tuple will be picked up by peers and the appropriate action will be taken. Another responsibility for this agent is to launch and maintain a **ResponseSenderRequestReceiver Agent** for every exported service (see below for details).

6.3.2 Importing Agent

This singleton agent continuously monitors the local tuple space for **ExportTuple(s)** and maps them to proxy services that are registered with the local registry. This mapping uses the same proxy bundle approach

described in section 3.2.2; D-OSGi automatically creates, installs, and starts a proxy bundle that offers the proxy service. In section 5.2.2, we showed that this is the only way available to integrate the imported service with the local OSGi framework. D-OSGi uses the same byte code manipulation technique of R-OSGi for creating the proxy bundle. However, unlike R-OSGi which always creates the proxy bundle on the client side, D-OSGi can be customized to use either import side or export side proxy bundle creation. In the latter case, the proxy bundle is pre-created by D-OSGi before transferring it to peers. The peers then simply install and start this received proxy bundle. Section 6.7 describes the motivation for supporting export side proxy bundle creation. The other responsibility of the Importing Agent is to monitor the local tuple space for `PropertiesUpdateTuple(s)` and update the properties of the corresponding proxy service.

6.3.3 Export Side Cleanup Agent

This singleton agent is responsible for cleaning up expired tuples that were inserted by the parent host. For example, when an exported service properties are updated, a corresponding `PropertiesUpdateTuple` is inserted. After the peers pick up this tuple and update their corresponding proxy services, the tuple is no longer needed. New property updates will result in new tuples being inserted. For performance and efficiency reasons and in order to prevent the tuple space from growing very large, this agent automatically cleans up those expired tuples.

6.3.4 Import Side Cleanup Agent

This is a similar clean up agent but from the perspective of the client. It is responsible for detecting the departure of peers and un-importing all services previously imported from those departing peers. Similarly it detects when previously imported services are withdrawn and automatically un-imports the service. It also cleans up expired tuples that were inserted by the client such as those representing invocation requests.

6.3.5 Response Sender Request Receiver Agents

For each exported service, there is an instance of this agent waiting to receive requests to be invoked on the service and to send back the responses. Note that we must have an instance of this agent for each exported service. This is because method invocation is synchronous in which case an invoker client waits for the result to be returned. If there was a single instance for handling all the exported services, then every single invocation on any of these services will have to potentially compete for attention with all invocations on all the exported services. This is clearly unacceptable. By having a different agent instance for each exported service, each service effectively will have its own thread of control for sending/receiving replies and requests respectively.

6.3.6 Request Sender Response Receiver Agents

For each imported service, there is an instance of this agent waiting to forward requests on behalf of the proxy service to the remote original service. Note that we must have an instance of this agent for each imported service. The justification for this design decision is similar to that given for the `ResponseSender-RequestReceiver` Agents; if we had a single instance for handling all the proxy services, then all requests on any of the proxies will have to potentially compete with each other for attention which is clearly unacceptable. By having a different agent instance for each proxy service, each proxy service effectively will have its own thread of control for sending/receiving requests and replies respectively.

6.4 The Group Abstraction

When exporting (migrating) a service, D-OSGi must ship out all the resources required for operating the service at the receiving peer in accordance with the adopted resource relocation strategy (see Section 4). Hence, the set of shipped resources varies for each service. In addition, the shipped resources also depend on the adopted proxy generation strategy (export side versus import side proxy generation). For export side proxy generation, D-OSGi simply ships out the proxy bundle.

In R-OSGi, the shipped data is associated with the service through its service properties. For example, R-OSGi defines the *policy* property for determining whether to transfer the individual service or the whole bundle. It also defines the *smart-proxy* property for identifying the smart proxy object. Other defined properties include the *injections* property which defines additional set of classes that will be included for transfer to the remote peer. Our concern with this approach is three folds:

1. This approach requires a separate property for each different type of resource which complicates the implementation.
2. Usability is also affected, the developer have to know about a large number of service properties.
3. If new mobility strategies are to be supported (see Section 4), the number of properties is likely to increase. Furthermore, depending on the adopted mobility strategy, not all the properties will be required for all applications. This further complicates usability because developers will have to also know about which properties are relevant for their specific application.

D-OSGi uses and adapts the concept of a *group* from the muCode mobility framework [11]. A group is the unit of code mobility. It can be considered as the briefcase that will contain all the data to be transferred to remote peers. In other words, it provides the developer with a container that can be filled arbitrarily with classes and objects, and shipped altogether to another peer. These classes and objects are equivalent to injections in R-OSGi (although in R-OSGi only classes can be injected). D-OSGi defines several methods for populating and querying the group. The group has some intelligence built-in. For example, when a class is added to the group, it automatically computes and adds the full closure of the class. To prevent certain classes or whole packages from being automatically added, they can be programmatically specified as *ubiquitous*. Ubiquitous classes and packages are assumed to be available in every node and hence will not be transferred. By default, system classes (`java.*` and their sub packages), java extensions (`javax.*` and their sub packages), OSGi runtime (`org.osgi.*` and their sub packages), and D-OSGi packages are all considered ubiquitous. The list of ubiquitous packages contributes to the *Import-Package* header of the proxy bundle at the destination site. The benefits of using the group abstraction can be summarized as follows:

1. It provides the developer with a single API for service mobility.
2. Because a group is the unit of code mobility, it is possible to have different group implementations with different capabilities. For example, we can create a group implementation that only handles bundle mobility. Another implementation may only handle service mobility but one service at a time. A third implementations may be able to handle mobility for a collection of services at a time. Some implementations may support specific resource relocation strategies such as rebinding or remote references.
3. We can change the group implementation, for example by using a different algorithm for calculating class closures, without affecting the using applications.
4. It simplifies the process of mapping the mobile entities to tuples.

6.5 Selective Imports

The default Importing Agent will import all exported services that are available in the transiently shared tuple space. In other words, it will import all services that are exported by any reachable peer. If the number of hosts involved is large, this may result in a large number of unnecessary imports. For example, some devices may be interested in using only a specific type of services. Similarly, some devices may be interested in using services that are provided by specific peers. To allow some degree of controllability with respect to service imports, the Importing Agent can be customized through the use of *import filters*.

An import filter is any object of type `IImportFilter` that implements a single method `public Boolean accept(String serviceURI, Dictionary properties)`. When associated with the Importing Agent, the filter is called whenever a service is about to be imported. Only if the `accept` method returns true then the service is imported. This is a straightforward use of the strategy design pattern [12] to encapsulate the filtering logic. D-OSGi comes with three default filter implementations:

- *UniversalImportFilter*. This is the filter that is used by default which imports all available services.
- *ImportByHostFilter*. This is a filter that selectively imports services from specific hosts which are specified when the filter is instantiated.
- *ImportByServiceTypeFilter*. This is a filter which filters services based on their service types.

We expect developers to provide their own filter implementations based on application requirements. For example, they can create filter implementations that filter out services based on specific service properties in which they are interested.

6.6 Determining the Code Fragment

Similar to R-OSGi, D-OSGi supports the automatic calculation of the transferred code fragment. D-OSGi automatically determines all the classes that must be transferred in order to register and operate the service at the remote peer. D-OSGi uses the following algorithm:

- Include all interfaces under which the service was registered with the OSGi registry and their full class closures i.e., recursively include all reachable classes and interfaces.
- For each explicitly added class (explicit injections), include both the class and its full class closure.
- Classes and interfaces defined by `java.*`, `javax.*`, `org.osgi.*`, and the D-OSGi packages are not included. These are assumed to be available at destination.
- Classes and interfaces that are explicitly specified as ubiquitous are not included. D-OSGi provides several methods for specifying class ubiquity at different levels of granularity. For example, specifying a package as ubiquitous automatically considers all classes within that package as ubiquitous. If the package name ends with `".*"`, then all the sub packages are made ubiquitous too. Alternatively, ubiquity can also be specified on class by class basis.
- For explicitly added serializable objects, their whole object graph is serialized automatically. In order to reconstruct these objects at destination, the involved classes are assumed to be available there. In the future, we will modify this behaviour to automatically include all classes that are required to reconstruct the objects at destination, subject to the ubiquity specifications.
- D-OSGi also automatically calculates the proxy bundle's Import-Package header based on the above steps.

6.7 Export Side Proxy Generation

In R-OSGi a proxy bundle is generated at the *import side* before it is installed and started. Our view is that generating the proxy bundle on the import side is not optimal. In this case, in order to *generate* and *operate* a proxy service under a given interface, R-OSGi must have available the *byte code* of all classes and interfaces reachable from the implemented interface (i.e., the full class closure of the interface). R-OSGi could ensure this data is always available at the receiving end by injecting, at the export side, all that is needed to operate the service at the import side; it can inject the full class closure of the implemented interface. At *minimum*, only the service interface needs to be transferred.

Now assume that the remote candidate service implements an interface that is provided by a 3rd party bundle which we also assume is ubiquitous in all peers. According to R-OSGi, only the interface class (excluding its closure) will be injected for transfer to the remote peer. The justification is that such closure is considered as belonging to the required environment; it should be available at destination where it can be re-imported. This is more efficient and preserves the network bandwidth since less data is being transferred. However, R-OSGi still has to transfer the interface class. We completely agree with this justification for excluding classes from imported packages but we have few other concerns. Firstly, the interface (without its closure) is still injected and transferred even though it is assumed available at destination. Secondly, in certain scenarios it may be more efficient to generate the proxy bundle at the exporting device as opposed to import side generation. For example, in order to preserve memory in importing devices with limited resources.

In D-OSGi we support both import and export side proxy bundle generation. In the latter case, a proxy bundle is pre-generated at the *export side* before it is transferred to peers. This also excludes the need to inject (and transfer) the ubiquitous interface or its closure. At the import side, the received proxy bundle simply re-imports the interface in question from the local 3rd party bundle.

6.8 Method Invocation

When a client calls a method of a service proxy, the call is forwarded to the actual service object on the service providing peer. In order to avoid the use of RMI (see Section 3.1), D-OSGi uses the tuple space to implement this remote invocation by means of message passing. The method call is mapped to a `RemoteMethodInvocationRequest` tuple which is inserted in the tuple space. At the service providing peer, the tuple is picked up, translated, and subsequently invoked on the actual service object. The result of the invocation is then mapped into a `RemoteMethodInvocationResponse` tuple which is inserted in the tuple space. At the client peer, this response tuple is then picked up, interpreted, and the result returned to the invoking entity (or an exception is thrown if the invocation had failed).

This implies that similar to R-OSGi, the formal parameters of remote services must be serializable. This raises the question “How can the D-OSGi bundle reconstruct (deserialize) the arguments on the service providing peer”. In other words, “How can the D-OSGi bundle locate the classes required for deserializing the arguments”. Note that method arguments can be one of three types: *primitive*, *standard* (`java.*` or `javax.*`), or *non-standard* (all other types). A non-standard type could have been imported from a bundle which is not visible to the D-OSGi bundle on the providing peer. In this case, there is no way for the D-OSGi bundle to reconstruct the arguments from the serialized stream. Note that this problem concerns only non-standard argument types. For standard argument types, the default class loading delegation process kicks in and eventually the parent/system class loader is used to locate the required types. Also note that R-OSGi does *not* recognize this issue and hence all argument types must be visible to the R-OSGi bundle which is not practical.

D-OSGi solution is to use the class loader of the invocation target (the service object) for locating the argument types. To achieve this, D-OSGi uses a special `ObjectInputStream` which overrides the

`resolveClass(...)` method to *dynamically* locate and use the appropriate class loader. The actual implementation also required changing Lime to use this special `ObjectInputStream` when deserializing `RemoteMethodInvocationRequest` tuples.

6.9 Usage Example

The following describes a usage example of the D-OSGi bundle. It shows the distribution and remote access of an OSGi service of type `IFServiceInterface`.

7 D-OSGi: Evaluation and Research Findings

The D-OSGi design adopts an import/export model in conjunction with the publish-subscribe interaction style. The authors in [5] describe the publish-subscribe interaction model as a system where “subscribers register their interest in an event, or a pattern of events, and are subsequently asynchronously notified of events generated by publishers [5]”. The authors suggest that many variants of the paradigm have recently been proposed with each variant being specifically adapted to some given application or network model. The same authors identify the “full decoupling of communicating entities in time, space, and synchronization” as the common factor underlying all these variants [5].

Publish-subscribe interaction style is anonymous where publishers do not have to know the consumers. Furthermore, it is multicasting in nature where it is possible to send an event to multiple consumers in a single `publish()` operation. This differentiates the publish-subscribe interaction style from other interaction schemes such as: message passing and RPC [5]. Other benefits include: separation of computation from communication, potential for integration, potential for scalability and dynamic adaptation [5]. Because of these perceived benefits, publish-subscribe is widely regarded as a suitable communication model for a wide range of applications [10]. In particular, the suitability of this style for mobile and wireless environments is widely agreed among researchers (see for example, [10]). The publish-subscribe communication paradigm is quite mature with tens of implementations ranging from research prototypes, open source, and commercial implementations.

D-OSGi adopts the same philosophy of R-OSGi, specifically:

- *Transparency.* Using D-OSGi, the details of distribution and the nature of the federation are hidden; services are accessed *exactly* as if they were local. Compare this with R-OSGi which requires clients to establish explicit connections to the remote framework instance (unless automatic connection is enabled).
- *Non-invasiveness.* The only requirements for enabling a service for remote access, is to associate it with a `REMOTE_CANDIDATE` property. This is similar to R-OSGi which explicitly marks services for remoting.
- *Consistent behaviour.* Unlike R-OSGi, D-OSGi does not require registration of listeners that are called when a remote service is found. Instead, remote services are *searched* and *accessed* in same way as any local service.
- *Spontaneous interactions.* D-OSGi supports spontaneous framework federations. No pre-configuration is required. As soon as two or more devices become connected, they are federated automatically. This support is enabled through the use of the Lime middleware.
- *Selectivity.* D-OSGi supports selectivity with respect to framework federations; it is possible to specify and control remote services visibility using the import filter mechanism. Furthermore, developers are able to provide their own filter implementations for easy integration with the D-OSGi bundle.

Listing 1: D-OSGi Usage Example

```
/** The service interface */
public interface IServiceInterface extends Serializable{}

/** The service implementation */
Class ServiceImpl implements IServiceInterface{
    public void echo(String s);
}

/** The offering bundle in the service providing peer */
Class ProvidingActivator implements BundleActivator{

    public void start(BundleContext context) throws Exception {
        /* obtain an IRemoting service (which is offered by the D-OSGi
        bundle) */
        ServiceReference remotingRef = context.getServiceReference(
            IRemoting.class.getName());

        if(remotingRef!=null){
            IRemoting remotingObject = (IRemoting)context.getService(
                remotingRef);

            /*obtain an empty group i.e., an empty container
            IMobileGroup group = remotingObject.createGroup();

            Hashtable props = new Hashtable();

            /*mark the service for remoting
            props.put(IRemoting.REMOTE_CANDIDATE_KEY, "true");

            /* In the simplest case, all you have to do is associate the
            Empty group with your service properties. D-OSGi is
            responsible for populating the group with the transferable
            code fragment which is automatically generated by
            D-OSGi. */
            props.put(IRemoting.MOBILE_GROUP_KEY, group);

            /*offer the remote service
            context.registerService(IServiceInterface.class.getName(),
            new ServiceImpl(), props);
        }
    }
}

/** In the client peer, simply look for a remote service of the given
type */
Class ClientActivator {

    public void start(BundleContext context) throws Exception {
        /* obtain the remote service (what you will actually get is the
        proxy service) */
        ServiceReference serviceRef = context.getServiceReference(
            "IServiceInterface", IRemoting.REMOTELY_IMPORTED_KEY);
        if (serviceRef != null) {
            IServiceInterface obj = (IServiceInterface)context.getService(
                serviceRef);
            obj.echo("remotely invoked method");
        }
    }
}
```

- *Generality.* The D-OSGi bundle is lightweight with a total footprint of approximately 600kb which includes the Lime middleware (approximately 400kb) and the ASM byte code manipulation library (approximately 42kb). Therefore, D-OSGi can be used whenever OSGi can be used.

Furthermore, the D-OSGi design is intended to support OSGi service distribution in a way that is independent of the application requirements, network setup or communication technology as described below:

- *Mobility model.* By adapting the muCode mobility toolkit [11], D-OSGi aims to address the general problem of code mobility in the context of the OSGi platform. However, currently D-OSGi only supports weak mobility of bundles and services.
- *Application design paradigm.* Currently, D-OSGi supports the client-server paradigm and a limited form of remote evaluation. It should be possible to support other paradigms more easily when compared with R-OSGi.
- *Resource relocation.* The current D-OSGi implementation supports both resource rebinding and relocation by copy strategies. Other strategies can also be easily supported.
- *Interaction style.* Publish-subscribe style is widely considered as a more appropriate communication style especially for mobile environments. There are literally tens of publish-subscribe implementations with different capabilities, architectures, models, and which target different application domains. The D-OSGi design and concepts are independent of any specific publish-subscribe implementation. This decouples D-OSGi from the underlying transport layer and subsequently it can be used in different network scenarios such as fixed networks and ad hock networks. Different D-OSGi implementations can be created that use the most appropriate publish-subscribe implementation. Furthermore, the adopted export/import model is consistent with existing standard OSGi services such as the UPnP Device service.
- *Communication protocol.* D-OSGi design is independent of any actual communication protocol such as plain TCP, HTTP or UDP and can be configured to use any of these protocols based on network setup and communication requirements.
- *Implementation.* By taking advantage of existing publish-subscribe infrastructures, the D-OSGi design facilitates implementations that are more modular and simpler to implement when compared with R-OSGi. The current D-OSGi implementation consists of approximately 2600 uncommented java SLOC with a footprint of approximately 144kb.

7.1 Experiments

We ran experiments on a single P4/3.00GHz PC with 512MB RAM running Windows XP and Sun's JVM 1.5. We used the Equinox OSGi implementation which is part of the Eclipse 3.2 platform. Subsequently, the experiments were conducted from within the Eclipse IDE. We used LimeII as our publish-subscribe infrastructure implementation which we adapted to run as an OSGi bundle. We launch two framework instances simulating two devices. Various remote service candidates were then registered in the first framework instance and are subsequently exported for use by bundles in the other framework instance. The remote service candidates were developed in a way that exercised different closure calculation and resource relocation strategies. For example, some types were assumed ubiquitous and others were explicitly transferred to the remote peer.

Using import side proxy generation strategy and the service type defined in listing 2, we measured the time for constructing the remote service's corresponding *ExportTuple* as 31ms on average for the first time the service is exported and as an average of less than 1ms for subsequent export attempts. This involves

Listing 2: Exportd Service Definition

```

/** The service interface */
public interface IExportService extends Serializable{
    public void invoke(byte[] data);
}

public class ExportServiceImpl implements IExportService{
    public void invoke(byte[] data){
        System.out.println("Received message of size: "+data.length);
    }
}

```

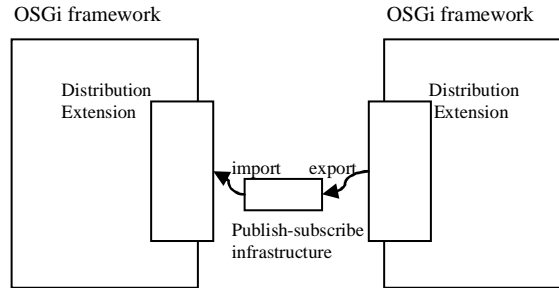


Figure 5: Design Outline of OSGi Distribution Extensions

the calculation of the service’s closure and setup of the service’s RequestHandlerResponseReceiver Agent. The service type `IExportService` is non-ubiquitous; it is transferred to the remote peer along with the service implementation. On the receiving peer we measured the time from detecting the *ExportTuple* in the local tuple space to registering the corresponding proxy service. This was measured as 148ms for the first time the service is imported and as an average of 58ms for subsequent imports. However, these measurements should be considered with a grain of salt as they depend on a number of variable factors such as the publish-subscribe infrastructure implementation. For example, the time for mapping exported data to/from tuples is dependent on the tuple implementation. Device capabilities and the adopted proxy generation strategy are other deciding factors. Furthermore, the current D-OSGi prototype is not optimized for performance.

7.2 Towards a Specification for OSGi Distribution Extensions

As shown in Fig. 5, an OSGi distribution extension is responsible for exporting OSGi services and their subsequent import at destination. At minimum, a specification for such an extension should address the following questions:

- What is exported by the offering peer and the *conditions* that trigger the exporting/withdrawal process?
- How the imported data is *interpreted* and *used* in the receiving peer including how to react to export/withdrawal events.
- In D-OSGi, synchronous remote invocation is modelled by means of asynchronous message communication using the publish-subscribe infrastructure. Therefore, the specification should also define a protocol for this message communication between the peers. However, this protocol should not be dependent on any specific publish-subscribe model or implementation.

A specification for a distribution extension may also describe the *components* of the extension and their responsibilities. The six agents of the D-OSGi bundle are a good starting point for identifying such components and responsibilities.

Other details for realizing this export/import model such as message representation, can be considered as implementation related and thus are left to the particular extension implementation.

8 Conclusion

The current D-OSGi implementation represents a proof of concept prototype of the feasibility and benefits of using the publish-subscribe communication model for OSGi service distribution. Although our main aim for this implementation of D-OSGi was to match R-OSGi capabilities, the D-OSGi design is more general and independent of any underlying implementation technology. We believe this work represents a major step towards a specification for OSGi distribution extensions.

References

- [1] The OSGi Alliance, “Osgi service platform core specification and service compendium, release 4,” August 2005. http://www.osgi.org/osgi.technology/download_specs.asp?section=2.
- [2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
- [3] J. S. Rellermeyer, “Services everywhere: Osgi in distributed environments-r-osi.” Presented in EclipseCon2007, Santa Clara, California.
- [4] J. S. Rellermeyer, “flowsgi: A framework for dynamic fluid applications.” MSc thesis. Department of Computer Science Institute of Pervasive Computing. ETH. Swiss Federal Institute of Technology.
- [5] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The many faces of publish/subscribe,” *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.
- [6] D. Gelernter, “Generative communication in linda,” *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.
- [7] A. L. Murphy, G. P. Picco, and G.-C. Roman, “Lime: A coordination model and middleware supporting mobility of hosts and agents,” *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 3, pp. 279 – 328, 2006.
- [8] A. Fuggetta, G. Picco, and G. Vigna, “Understanding code mobility,” *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342 – 61, 1998.
- [9] E. D. N. Cugola and A. Fuggetta, “The jedi event-based infrastructure and its application to the development of the opss wfms,” *IEEE Transactions on Software Engineering*, vol. 27, no. 9, pp. 827–50, 2001.
- [10] Y. Huang and H. Garcia-Molina, “Publish/subscribe in a mobile environment,” *Wireless Networks*, vol. 10, no. 6, pp. 643 – 52, 2004.
- [11] G. P. Picco, “code: A lightweight and flexible mobile code toolkit,” in *Proceedings of the 2nd International Workshop on Mobile Agents* (K. Rothermel and F. Hohl, eds.), Lecture Notes in Computer Science, (Berlin, Germany), pp. 160–171, Springer-Verlag, 1998.

- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.