ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

**Systems Group**

**Master's Thesis**

# The Virtual OSGi Framework

by
Dimitrios Papageorgiou

January 14th 2008 - July 13th 2008

Advisor:
Jan S. Rellermeyer
Prof. Gustavo Alonso

Systems Group

ETH Zurich
Department of Computer Science
8092 Zurich
Switzerland

# Abstract

OSGi (Open Service Gateway Initiative) is an open industry standard. The OSGi specifications describe a dynamic module system for Java. It is a platform for managing Java software modules, which are called bundles. Bundles can be combined to form larger applications. An important aspect of the OSGi framework is the management of the dependencies between bundles. The OSGi framework was intented to operate on a machine. Recently, different approaches have emerged which are trying to facilitate remote access to services through bridging between the otherwise isolated framework instances. The disadvantages of the bridging approach is that it does not provide a unified access to local and remote services and it does not provide dynamism. Achieving this would unavoidably mean altering the framework implementation. However, the middleware should not impose any restrictions on OSGi services. In particular, it should be possible to use existing bundles and services in distributed setups without any need for modification.

This thesis presents the Virtual OSGi framework, a distributed OSGi framework implementation, which is able to transparently run on multiple machines, yet acting like a single OSGi framework. The Virtual OSGi framework does so by adding a virtualization layer on top of each individual OSGi runtime. The underlying infrastructure of the system is a peer to peer network based on a distributed hash table. Beside maintaining non-invasiveness, other challenges that needed to be dealt with, were the connection and communication of different OSGi frameworks, which are spread across the system. Furthermore, the treatment of bundles and their lifecycle. Finally, for the treatment of remote services, their registration and how an application will call them, the Virtual OSGi Framework makes the contribution of unifying local and remote services, without posing any restriction to the services that are registered in the framework. Finally, in this thesis, we compare the Virtual OSGi framework with other distributed systems, and we discuss what are its advantages and disadvantages of the Virtual OSGi approach.

# Preface

This thesis is submitted for partial fulfillment of the requirements of the degree Master of Science in Computer Science at the Swiss Federal Institute of Technology(ETH) Zürich. The duration of this thesis was six months from January 14th 2008 to July 13th 2008 in the System Group led by Prof. Gustavo Alonso at the Department of Computer Science of the Swiss Federal Institute of Technology Zürich and was supervised by Jan S. Rellermeyer and prof. Gustavo Alonso.

## Acknowledgements

First of all I would like to thank Jan S. Rellermeyer and prof. Gustavo Alonso for supervising and guiding me during the period of the thesis and for giving me the opportunity to work on this topic. Furthermore, I would like to thank other members of the Systems group, Michael Duller, René Müller and Dr.Spyros Voulgaris, for being very willing to help me with their helpful advice and spending some of their valuable time to answer my questions.

This thesis is dedicated to my family and to my friends for their strong support during this period

Zürich, July 13th, 2008.

Dimitrios Papageorgiou

# Contents

# Chapter 1

# Introduction

Nowadays, applications tend to be more demanding in resources and in computing power, which results to a turn towards distributed computing [27]. A continiously increasing number of software applications are developed in a distributed way to exploit larger amounts of resources and to achieve connection in a transparent and scalable way. Distributed platforms have been developed either in a centralized or a decentralized way. Concerning centralized networks, Napster is a representative example. It became famous because of the lawsuit against the service, offered by a famous band [34]. On the other hand, pure decentralized applications such as peer-to-peer networks have also been widely developed, and a lot of derivatives can be found. For example, structured p2p networks like distributed hash tables (DHTs) have been developed. Another example are enstructured p2p networks such as Gnutella [17], which build connections among the nodes in an arbitrary way. PlanetLab [35] is another example of a distributed computing platform, maintained by a community of researches. It is a global platform for deploying and evaluating network services.
In addition, there is also a trend towards Service Oriented Architectures (SOA) [10]. The goal of SOA is to build applications based on large scale components offering standard interfaces to the other components which is the only thing that other components need to know for communication. The services that participate in the application are independent of each other, heterogeneous and distributed. The only information that a client needs is the interfaces of the service. The typically cited example for service oriented architecture is web services [12]. A less obvious representative of SOA is OSGi [11]. The OSGi alliance has released an open business standard for managing dynamic modules in Java. The OSGi platform is a platform for service oriented architectures with minimum overhead, since services are directly accessible through object references. In contrast to other systems (WS, EJB) that add indirections, the OSGi framework allows to build applications out of loosely coupled software modules to facilitate the addition, removal and update of each module independently.

In this thesis the Virtual OSGi framework, a distributed OSGi framework, is presented. It combines these two trends of technology (service oriented architectures and distributed computing). On the one hand, it exploits the benefits from the service oriented architectures such as loosely coupled components (services). Furthermore, it enables integration of pre-built and pre-tested modules

and it reduces maintenance costs. On the other hand, from the P2P networks, it exploits the fact that all clients provide resources, including bandwidth, storage space, and computing power. Thus, as nodes arrive and demand on the system increases, the total capacity of the system also increases. Cloud computing [28] is a similar case, where many servers interconnect and offer huge resources and high capabilities. Users do not care about the underlying techonology of the whole infrastructure. The distributed nature of P2P networks also increases robustness in case of failures by replicating data over multiple peers. The Virtual OSGi framework is an OSGi framework implementation, that is designed to run as a paravirtualization layer atop traditional local OSGi frameworks. As a virtualization layer, it connects nodes in a network and implements the coordinated behavior. The connection and collaboration of the nodes in the framework is being done trasparently. As a result the user does not need to have any information about how many nodes are participating in the network, with which nodes is connected or where a bundle or a service is located. Furthermore, it is responsible for maintaining the connections between the nodes of the network, keeping information about a small amount of nodes and dealing with replication of data so as to avoid losing important information when a node stops participating in the network, and with fault tolerance.

## 1.1 Motivation

OSGi is a rapidly developed technology in the market (enterprise, mobile or open source). The OSGi framework, operates on a single machine. Different efforts [33, 5] have been made to present a distributed OSGi framework but the common approach of all efforts is that they try to bridge between different local OSGi frameworks by facilitating remote service access. The problem with this approach is that the resulting system is static. The breakdown of a node might result in a loss of critical information, and even to the shutdown of the system. Furthermore, the proposed distributed OSGi frameworks do not deal with scalability and fault tolerance. To the best of our knowledge, the approach of making the different instances appear as one coherent framework has not been explored before. In this thesis, a distributed OSGi framework was implemented, but following a different approach, so as to deal with the shortcomings that the previous proposals have. Besides connecting nodes which are running different OSGi frameworks, the problem of scalability and fault tolerance had to be solved. The path that was followed was to have a global OSGi framework which acts as a virtualization layer above the local frameworks offering to the user transparently one framework that is consisted of many nodes. In order to achieve scalability and fault tolerance, the appropriate network infrastructure was selected and basic replication and stabilization techniques. Finally, the system was able to handle dynamic changes of the network (node churn), because the functionality and operation of the distributed framework does not depend on the availability of a particular node of the system and still offer the operations that the specifications currently mandate.

In this thesis the Virtual OSGi framework is presented, which is a distributed OSGi framework running in cooperation with existing local OSGi frameworks. In the Virtual OSGi framework the large scale distribution of OSGi services over

heterogeneous computer networks is explored. OSGi bundles and services can be deployed to a virtual framework. As a next step, the Virtual OSGi framework can offer the autonomous allocation of bundles and services to physical nodes and the rearrangement of the setup as a reaction of changes in the utilization and availability of resources. Concluding the goal of the Virtual OSGi framework was to make a dynamic group of machines appear as a single OSGi framework (OSGi on the cloud).

## 1.2 Outline of the Thesis

Chapter 2 presents the background of the Virtual OSGi framework. Chapter 3 describes the architecture of the system. In the following chapter 4, the implementation details of the Virtual OSGi framework are discussed as well as, the design decisions that were necessary to implement the system. Chapter 5 presents an evaluation of the system and Chapter 6 concludes this thesis.

# Chapter 2

# Background

## 2.1 OSGi

### 2.1.1 Introduction

The OSGi Alliance [11] is an independent, non-profit corporation working to define and promote open specifications for the delivery of managed services to networked environments, such as homes and automobiles. The OSGi specifications describe a platform for universal middleware, establishing a standard service-oriented, component-based environment.

Even though, the OSGi specifications were, in the beginning, targeted at Internet gateways with home automation applications, a lot of its attributes made it applicable to other markets. Mobile industry (Nokia, motorola), vehicle industry (BMW), and enterprise are adopting OSGi specifications. For example the OSGi platform has been made a standart part of he BMW high-end telematics platform and is finding its way into many Volkswagens [15]. The presence of OSGi technology based middleware in many different industries is creating a large software market for OSGi software components. The great advantage of OSGi specifications is that they form a small layer that allows multiple Java-based components to efficiently cooperate in a single Java Virtual machine.
In summary, the OSGi specifications create universal middleware that is cross platform as well as cross industry. Adoption of the OSGi specifications reduces software development and maintenance costs and provides new business opportunities.

### 2.1.2 OSGi Architecture

The basic picture of the OSGi architecture is the following. The architecture is based on layers. Above the Virtual machine layer, the module layer of the OSGi framework exists. Above it the Life cycle layer in the higher layer is the Service layer.
OSGi is a specification for a service platform framework and service bundles. An OSGi implementation has to implement the framework and can optionally provide service bundles. These bundles supply basic functionalities like logging or dynamically wiring producers and consumers. In this thesis we extend existing

OSGi frameworks so as to work in a distributed way.
The framework forms the core of the OSGi Service Platform Specifications. The most famous non commercial OSGi framework implementations are the Eclipse Equinox[13], Apache Felix[1], Knopflerfish[14], Concierge[32]. The OSGi framework provides a general-purpose, secure and managed java framework that supports the deployment of extensible and downloadable applications known as bundles. The bundle is a unit of modularization and it is the basic unit of OSGi applications. An OSGi application can be consisted of many bundles, and a bundle can be used in more than one application of the framework. Bundles communicate among each other through dependencies which are declared in a special file of the bundle called Manifest file under the phrases Import-package, Export-Package. Moreover the bundle is consisted of Java classes and other resources, which provide functions to the user that uses the bundle. These resources are wrapped as a Java ARchive (JAR) file. This JAR file contains the resources and the data which are necessary to provide some functionality to the user. Resources can be Java class files, images, html files or additional JAR files. It also contains a manifest file with additional information, which describes the contents of the jar and provides information about the bundle(bundle name, import and export packages, activator class). Bundles can be downloaded and installed on devices, running an OSGi framework. After that they can be started. Once the bundle is started its functionality is provided to the application and services are exposed to other bundles which are installed to the OSGi framework. Furthermore, before the bundle is started it has to be resolved. This process satisfies the dependencies that the bundle has. For instance, a bundle that imports a package should find another bundle that exports it. If all constraints of the bundle are satisfied then the bundle is resolved and ready to start. The whole resolving process and our solution for the Virtual OSGi framework will be presented in Chapter 3. Figure 2.1 gives us an example of the wiring of bundles when a bundle is wired with two other bundles so as to satisfy its imports and one of them tries to satisfy its import, but this cannot happen because of a conflict in the version of the two packages.

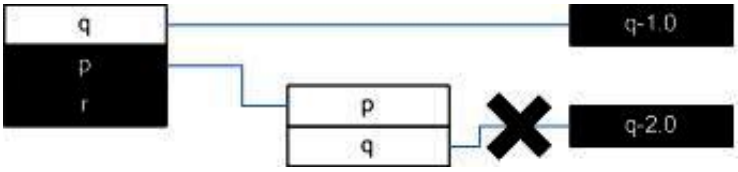



Figure 2.1: Resolving Example

### 2.1.3 Life Cycle Layer

Bundles in an OSGi framework can be managed at runtime. For instance, either another bundle or the intervention of a user can start a new Bundle. This triggers the execution of the Bundle Activator. This class implements the interface Bundle Activator and implements the methods start() & stop(). This methods are called when a bundle is to be started or stopped. When the bundle is started it receives a Bundle Context, whose methods can call. Only through the Bundle Context can the bundle use the OSGi framework. Moreover, for

each bundle installed in the OSGi service platform there is an associated Bundle object. This can be used to manage the bundle's life cycle. A bundle can be in one of the following states:INSTALLED, RESOLVED, STARTING, ACTIVE, STOPPING, UNINSTALLED.
When a bundle is installed, it is stored in the persistent storage of the framework. Whether a bundle has been started or stopped must be recorded in the persistent storage. A bundle that has been persistently recorded as started must be started whenever the framework starts until the bundle is explicitly stopped. In the following picture we can see the state diagram of the bundle lifecycle. In Figure 2.2 a view of the bundle life cycle is presented.

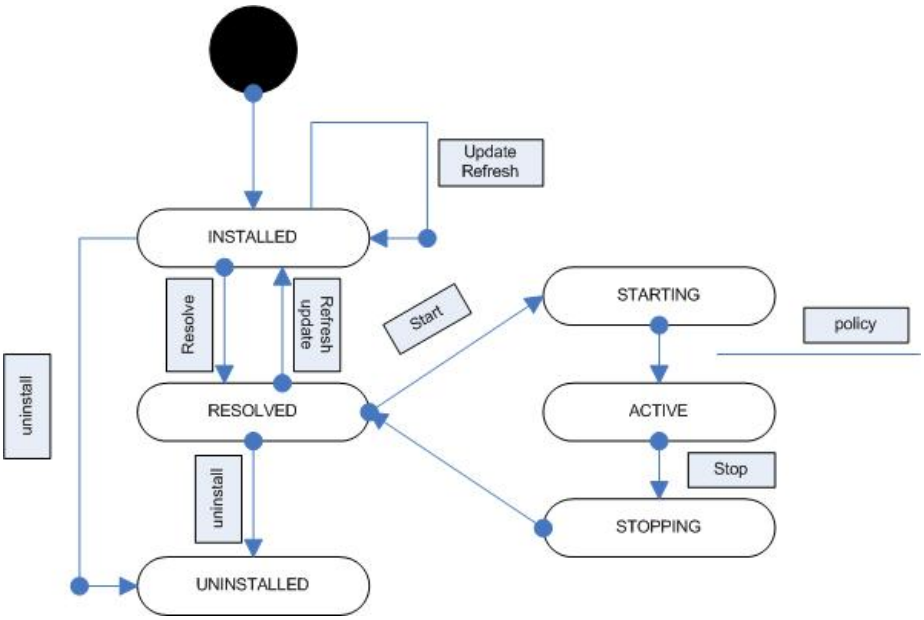



Figure 2.2: Bundle Life Cycle

A bundle is activated by its bundle Activator object if it exists. Information about the bundle activator are on the manifest file. Activation of a bundle can be postponed for a later time from its start. This policy is defined as the lazy Activation policy.

### 2.1.4 Service Layer

A service is a Java object that is registered under one or more interfaces with the service registry. Services are registered by bundles. Bundles also search for them or receive notifications when their registration state changes. A service object also has some properties which identify the service and helps the framework to deal with it. Properties hold information as key/value pairs. The Service properties are intended to provide information about the service object. Properties and filters, play an important role in the lookup of the service. Services are registered by the name of the interface or the names of the interfaces, if they implement more than one. Services must be registered, in order to be accessible by other bundles. Other bundles can get this service only through

the registered interface, by first requesting a service reference object. Moreover, bundles can search for services with the help of filters, which operate above the properties of the service. Its purpose is to check whether the properties of a service object satisfy the constraints that it imposes. After a bundle obtains the service reference object, it can retrieve the service. The OSGi framework maintains a service registry that stores the whole amount of services that have been registered in the framework by bundles. In this thesis we will present a distributed service registry based on a distributed hash table. In the Virtual OSGi framework services can be retrieved either with the use of the filter object or without any constraint.

In addition to these, Service Events are supported by the framework which report service registration, unregistration and property changes. Service Listeners participate in the delivery of these events. The Virtual OSGi framework, deals with service Listeners and service events, which are distributed. Concluding, the service layer must be collaborative and dynamic, and provide mechanisms that make it possible to handle the fact that bundles and their services evolve over time.

## 2.2 Grid Computing

Grid computing [19] is an approach of trying to solve difficult scientific problems by enabling the sharing selection and aggregation of geographically distributed "autonomous" resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements. The goal of Grids is to exploit and share distributed computational capabilities so as to offer their result as a service. Usually a Grid can be a supercomputer composed of a network of loosely coupled lighter computers and performs heavy and large tasks. An important characteristic of a Grid is that it is loosely coupled heterogeneous and geographically scattered. Grids serve for solutions of difficult problems that demand either great computational power or fast access of large amounts of data at the same time. Usually these problems are scientific or technical ones. A first example of Grid is SETI (the Search for Extraterrestrial Intelligence) [41], a scientific effort to discover intelligent life elsewhere in the universe, primarily by attempting to discover radio signals that indicate intelligence. Another example, is the Telescience Project [37] which provides remote access to an extremely powerful electron microscope at the National Center for Microscopy and Imaging Research in San Diego. Users of the grid can remotely operate the microscope, allowing new levels of access to the instrument and its capabilities. Moreover, a lot of universities maintain a Grid for scientific reasons, so as to do heavyweight experiments, such as Xibalba of ETH Zurich.

Virtual OSGi is different from Grid computing, because it is implemented as a structured peer to peer (p2p) system. The goal of p2p systems is to take advantage of the idle cycles and storage of the edge of the Internet, effectively utilizing its "dark matter". This overarching goal introduces issues including decentralization, anonymity and pseudonymity, redundant storage, search, locality, and authentication. Peer to peer's focus on decentralization, instability, and fault tolerance exemplify areas that essentially have been omitted from

emerging Grid standards. On the other hand, a Grid, is a massive resource to which a user gives his or her computational or storage needs. The Grid's goal is to utilize the shared storage and cycles from the middle and edges of the Internet. Furthermore, with grid computing it is not possible to have such a flexible network where nodes leave and join the network continuously, because the network in the grid is basically static and there are not any reliable fault tolerant mechanisms. Moreover in grid computing, it is more difficult for a node to participate to an existing network, above which a framework is running and use bundles and services from the system.

## 2.3 Cloud Computing

Cloud computing [28] is a computing paradigm in which tasks are assigned to a combination of connections, software and services accessed over a network. This network of servers and connections is collectively known as "the cloud". Computing at the scale of the cloud allows users to access supercomputer-level power. Using a thin client or other access point, like an iPhone or laptop, users can reach into the cloud for resources as they need them. For this reason, cloud computing has also been described as "on-demand computing". It is an evolution of Utility computing, and a way to increase capacity or adding capabilities, without investing in new infrastructure. A computer cluster can offer cost-effective service in specific applications, but may be limited to a single type of computing node that allows all nodes to run a common operating system. Alternatively, the canonical definition of grid is one that allows any type of processing engine to enter or leave the system dynamically. Applications based on cloud computing are expanding rapidly as connectivity cost falls. The architecture behind cloud computing is a massive network of "cloud servers" interconnected as if in a grid running in parallel, sometimes using the technique of virtualization to maximize the utilization of the computing power available per server.

Figure 2.3: Cloud Computing

The advantages that cloud computing has, are the following. First of all, sharing of peak-load capacity among a large pool of users, improves overall utilization. Second, a user may no longer have to be tethered to a traditional computer to use an application, or have to buy a version that is specifically configured for a phone. Application services are available, independent of the user devices and network interfaces. Furthermore, users will not have to worry about storage capacity, compatibility or other concerns. Finally, their ability to use external assets to handle peak loads is a very important advantage (it is not needed to engineer for highest possible load levels).

Cloud computing is used widely nowadays, one example of cloud computing is Amazon Elastic Compute Cloud [7] which is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. Another example is the Google File System [4], a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

The Virtual OSGi framework is an application of Cloud computing as it satisfies its basic characteristics, even though it is not a pure application of it. Firstly, the whole number of nodes that participate on the Virtual OSGi framework, can be considered as a cloud, because that's how they appear to the user. However, the user operates a single local machine that atop of it operates an OSGi framework. The hardware is a cloud but the software makes it appear as a single, massively parallel and redundant machine. This machine participates transparently in the Virtual OSGi framework. Secondly, the resources of all the nodes are used for storage or for parallelizing operations by distributing services that are registered by bundles locating in different nodes. Pararellization can be achieved through replication of services to different nodes. As a result, request for services can be served concurrently from different nodes. On the other hand, the Virtual OSGi framework, offers resources from many nodes to a user that runs an OSGi application but it does more than these. Its main target is to offer the OSGi characteristics in a transparent way and to achieve scalability when more and more nodes are joining the network and fault tolerance so as to relieve the system in a fast and efficient way from node breakdowns, and loosing as few data as possible.

One example of cloud computing is MapReduce. MapReduce is a programming model for processing and large scale data. This processing will result to the production of other data. MapReduce hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Programmers specify two functions. The first one process key/values pairs and produces intermediate results and the second one which combines all intermediate values for a particular key and produces a set of merged output values. Examples of functions that can be expressed as MapReduce computations are distributed grep, count of URL access frequency and distributed Sort. Its basic goal is to get as an input huge amount of data, distribute and parallelize the processing and return the result. In the Virtual OSGi framework the main goal is to parallelize an application but having the services that are registered from its bundles to many

nodes, distributing the load from one machine to many different ones. The Virtual OSGi framework is not limited by having to do huge processing to large amounts of data. MapReduce paradigm it just distributes the data to many nodes and all of them do the same processing. The Virtual OSGi framework distributes an application by having installed its bundles in different nodes and thus, the nodes do different processing and dealing with different tasks, according to the services that are provided by the bundles located in these nodes.

## 2.4 Virtualization

Virtualization [9] is a technique that hides the physical characteristics of computing resources from their users. This includes making a single physical resource (such as a server, an operating system, an application, or storage device) appear to function as multiple virtual resources; it can also include making multiple physical resources (such as storage devices or servers) appear as a single virtual resource. An example of Virtualization is operating systems virtualization, where multiple operating systems, run above of another base operating system. Operating system-level virtualization is commonly used in virtual hosting environments, where it is useful for securely allocating finite hardware resources amongst a large number of mutually-distrusting users. It is also used, to a lesser extent, for consolidating server hardware by moving services on separate hosts into containers on the one server. Other typical scenarios include separating several applications to separate containers for improved security, hardware independence, and added resource management features. OS-level virtualization implementations that are capable of live migration can be used for dynamic load balancing of containers between nodes in a cluster.

The Virtual OSGi framework implements the virtualization layer in a global OSGi environment. This layer is added between the local framework and the bundles and services that are installed or registered. The logic that it has, is that it selects which operation should be treated by the local OSGi framework and which should be treated by the Virtual one. It presents a number of local OSGi frameworks as a global united framework operating exactly as a local one, hidding from the user the network communication and offering to the bundles services either local or remote ones in a transparent way. Furthermore it tries to combine the resources of all nodes that participate on the system.

Operating system virtualization allows system hardware to run multiple instances of different operating systems concurrently, allowing to run different applications requiring different operating systems on one computer system. The operating systems do not interfere with each other or the various applications. The result is a system in which all software capable of execution on the raw hardware can be run in the virtual machine. In the operating systems virtualization the guest operating system has the illusion that that it is dealing with the hardware but in fact it is dealing with the virtualization layer. On the other hand, in the Virtual OSGi framework the virtual layer is aware of the underlying framework and tries to exploit a lot of its operations so as to produce a result for a request of an application. This is closer to paravirtualization where, the two layers are cooperating. On the contrary, OSGi applications, have the

illusion that they communicate with the one OSGi framework, running on one machine. even though they communicate with the Virtual layer, which maps operations to the underlying framework or it executes them. Another difference between Virtual OSGi and OS virtualization is that the first one adds abilities to the application by offering communication to other nodes, such as computing power or information on dependencies of the bundles and does not limit the application by hidding information for the system.

## 2.5 R-OSGi

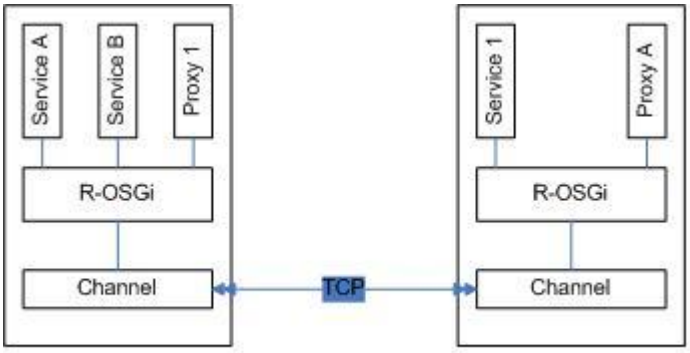



Figure 2.4: R-OSGi

R-OSGi [33] extends the possibilities of existing OSGi frameworks. It is a lightweight system for accessing services on remote OSGi frameworks. R-OSGi uses a service discovery protocol (SLP, described in RFC 2608) [24] to make the services visible to other nodes. In order to acquire the requesting service, it establishes a channel with the peer that has this service. Through this channel, peers exchange specific kind of messages which contain crucial information about the services offered and the request that peers have. An application can decide which service that matches its requirements will be fetched. Fetching a service is the operation where the service interface and its attributes will be transmitted. This is crucial information which makes the service remotely accessible. After that, R-OSGi builds a service proxy bundle in the node that requests the service. This proxy is implemented by bytecode manipulation based on the ASM library [23]. With the help of the service interface, a class that implements the service's methods is created. Because the service proxy is a bundle, an implementation of the BundleActivator is provided, so as to be able to be installed, started and stopped in the framework in the requesting node.

Method invocation in R-OSGi is implemented through messages. The requesting node, sends a message to the node that has the service object, which contains the description of the service together with the arguments. When in the provider this message is received, the real service method is called and a reply message is sent back to the other node. The method of the proxy bundle

will return a value. One limitation is that the arguments and the replies of remote services have to be serializable.

In R-OSGi, there are two use cases. In one case, the application should be aware of the distribution and its purpose is to connect remote services and operate on them. In the other one, if transparency is of the highest importance, an Adapter Bundle should be implemented on the client side, which is responsible for starting connections and fetching services. On the other hand, Virtual OSGi follows a different approach from R-OSGi. Its goal is to connect different nodes running different OSGi frameworks and present them to the user as one entity. Virtual OSGi apart from transparency, additionaly offers unified access to local and remote services, because services are accesible by every node, no matter from which node a service was registered to the service registry which in the Virtual OSGi framework it is completely distributed. The Virtual OSGi framework presents a complete solution for a distributed approach of an OSGi framework implementation. It uses the R-OSGi logic (fetching of the service, creation of the proxy bundle, remote method invokation) to get a remote service from another peer, but apart from that it implements a distributed service registry through which bundles can access registered services.

# Chapter 3

# Architecture

## 3.1 Introduction

The Virtual OSGi framework adds a virtualization layer between the local OSGi framework and the applications that install bundles and register services. In Figure 3.1 the big picture of the Virtual OSGi framework is presented. It is responsible for choosing whether an operation will be executed in the local framework or it will be executed in the virtual layer. It avoids dealing with the most operations that have to do with the moduler layer except for a few cases, such as the constraints solving of a bundle and all operations of the service layer are executed in the virtual layer.

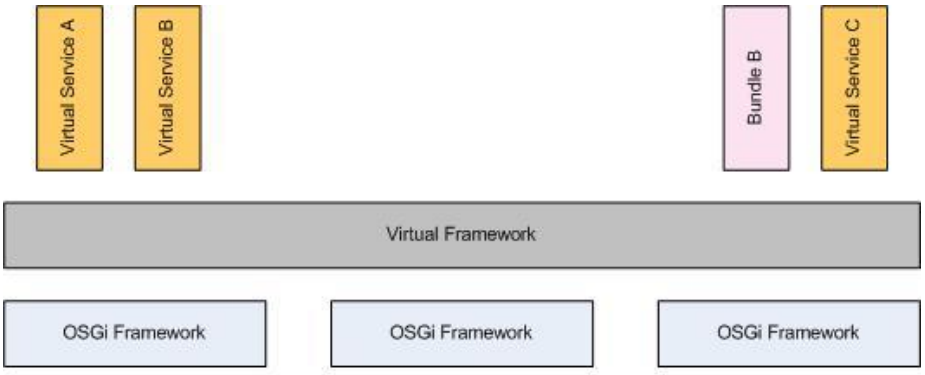



Figure 3.1: The big picture

In the Figure 3.1 is presented how the Virtual OSGi presents the whole system to a user of the framework. Below there are the local OSGi frameworks that operate on each node that participates on the system. Atop of them, the Virtual OSGi bundle runs and hides the network infrastructure, the communication, the number of nodes that participate on the network, localization information and other details, presenting the whole system as one entity.

Virtual OSGi operates above a normal local OSGi framework. It is actually a bundle which is installed in the local framework and it intercepts the communication of the bundles with the local framework. The basic logic that Virtual OSGi adds to existing OSGi frameworks, is to decide when a request from another bundle can be satisfied from the local framework, or network communication for this request is needed. Furthermore, both of the above operations might be needed. Bundles are installed through the Virtual OSGi bundle to the local framework, but they started, stopped and updated by the Virtual OSGi bundle. A node can call every bundle that is installed through the Virtual framework even though this bundle is not installed on this node. Furthermore, the Virtual OSGi framework maintains a distributed hash table as a registry for services and for bundles, in case that another node wants to get a bundle or do operations using this bundle or its bundleContext. In addition to these, in the distributed hashtable information about the resolving procedure of a bundle are maintained in clusters in nodes of the network, where cluster is the name of the object that is used for storing information about the constraint solving of a bundle. In the Figure 3.2 the general idea of the Virtual OSGi framework is depicted, where nodes are connected with some of their peers and they can communicate with them or with other nodes transparently. The circle that the nodes are connected is the Chord ring, which is used as a network backbone of the system.

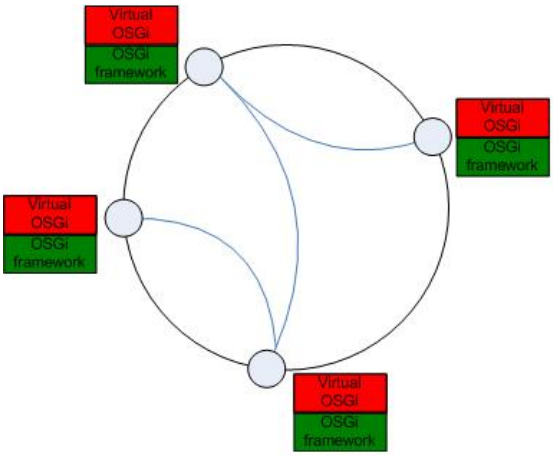



Figure 3.2: The Virtual OSGi big picture

## 3.2 Network Communication

In order to implement the Virtual OSGi framework a network backbone was built for the communication of the nodes that will participate in the whole infrastructure. A lot of proposals have been presented from the research community either centralized or decentralized. Furthermore, a lot of subcategories

exist in these two basic discriminations in distributed systems, making the final decision about what to select and what it is appropriate in our case difficult.

### 3.2.1 Distributed Hash Tables

#### Centralized versus Decentralized Systems

The centralized paradigm is described by one processing unit whose target is to solve one problem at hand. In the case that is needed to solve complex and difficult problems, its algorithms are sophisticated and usually they are specialized to solve one specific problem. The centralized paradigm is able to find the optimal solution of the problem. On the other hand, the decentralized paradigm is able to provide solutions faster than the centralized ones, especially in cases where complex algorithms are needed or more information is needed to be analyzed. Decentralized systems distribute the work load to many nodes, which might not do very complex processing. Usually the complexity lies on the communication between the nodes, to locking mechanism and to the distribution of the data that needed processing.

An advantage of decentralized systems against centralized ones is system robustness. In a centralized system, failure of the server will lead to a complete failure of the whole system. In a decentralized system, failure of one node, might lead only to some loss of data but not to a complete breadkown. In addition to this, decentralized systems have stabilization algorithms and replication techniques that a failure of one node will be passive and not affect the efficiency and productivity of the remaining infrastructure. Another advantage of decentralized systems against centralized is that the reconfiguration of the system is less expensive. As a result in cases where dynamic problems have to be solved, decentralized solutions are in favor.

On the other hand, there are occasions when the centralized paradigm may be favored over the decentralized one. The centralized paradigm is usually favored when the application consists of complex but static problem, for instance, determining the best locations for a set of warehouses and crossdocks. This is because a centralized system is more likely to find the global solution, and not some local maximum. Importantly, with a static problem any costs associated with a suboptimal solution will be faced in perpetuity. Not to mention the fact, however, that decentralized solutions such as Grid Computing have been presented for solutions of very heavy and static problems.
Considering the advantages and disadvantages of these two solutions, a non centralized completely distributed solution was selected. In this case flexibility of adjusting the system is important. Moreover, one of the goals is to simplify access in the whole framework for a simple node who wants to participate on the network and offer its resources to it. Another requirement is fault tolerance and to avoid having a complete breakdown of the system when a node breaks down or gracefully leaves the network.

To conclude with the selection of the network protocol, a decision about what decentralized network will be adopted had to be taken. In this case three options exist, Grid computing, cloud computing and a peer to peer network.

From peer to peer networks unstructured and structured networks exist, such as Gnutella or distributed hash tables (DHT) respectively. In order to avoid search mechanism as flooding, as much as possible, a decision was made not to deal with unstructured peer to peer networks and to focus on grid or structured peer to peer networks.

Peer to peer Internet applications have recently been popularized through file sharing applications (Napster, Gnutella). Besides the copyright issues raised by this applications, these systems have many interesting technical aspects like decentralization control, self-organization, adaptation and scalability. Peer-to-peer systems are distributed systems in which all nodes are equal. No super nodes exist that have more information or more responsibilities than the remaining ones. One of the problems that peer-to-peer applications have to solve is the object location and their discovery by the other nodes in the network. Distributed Hash Tables (DHTs), are peer-to-peer networks that can be used in many applications and they offer fault-resilience, scalability and reliability. In DHTs every node that participates in the network has a unique id, which is obtained through consistent hashing. With the help of this id, each message is successfully routed through the network, to the node that this message should be delivered. The expected number of routing steps is $O(\log N)$, where N is the number of nodes participating in the network. A lot of applications can be implemented having as a network backbone a distributed hash table, such as publish subscribe, a storage facility, or a distributed mail server.

Virtual OSGi is heavily based on a peer-to-peer network, and specifically a Distributed Hash Table. A lot of distributed hash tables have been proposed and developed, many of them having crucial differences. The basic idea, nevertheless, is the same in every DHT proposal. These decentralized networks provide a lookup service based on consistent hashing, with the help of a hash table which is distributed in the nodes that participate in the network. Every node stores information about a small amount of nodes in the network as pairs of a hashtable (name, value), but this does not prevent every node in the network to obtain information about every other node even though it does not store information about that node.

### 3.2.2 Consistent Hashing

The DHT network is responsible for storing values to the node that it belongs to, based on the unique id that this value has, when a hash function is applied to some of its values. In our case it depends of the object that is hashed. First of all, the nodes that participate in the network should have a unique id, which will be used from the framework for routing, lookup, or storing of messages.
The three problems that DHT try to alleviate with the use of the hash function are Network Congestion, swamped servers and the distance between the nodes of the system. Generally two are the approaches of storing data in a system exist. The first one is Monolithic storing architecture where a single big storage exists which is placed in a place so as to serve all the users. The disadvantage of it is that is a single point of failure and that it might be congested. The second architecture uses a distributed approach, where each neighborhood has its own

storage that serves residents in that neighborhood. This approach solves the two problems above, because it will survive a failure and it spreads the work across several machines. Moreover, it has the advantage that parts of the storage are closer to the user and can threfore deliver content faster. The disadvantages are that the hit rate will be lower because distributed storage is smaller and because they receive requests from a smaller population.

A hybrid solution can be also developed. The things that someone should bear in mind are the following. First of all the fact that it has to do with a large distributed system. In this system there is no centralized control as long as it is completely distributed. Furthermore, it must offer robustness and it must scale gracefully. In addition to these, the system must be able to prevent swamping of host spots and it has to minimize the usage of the network. It must also offer load balancing and it has to add low overhead.

Examples of hybrid solutions are hierarchical systems, or systems of cache that cooperate so in a case of a miss a cache server tries to ask other caches instead of going directly to the content provider. In this case a complete distributed peer to peer system is used, and as a result the use of a hash function was adopted. This hash function might map URLs to the set of caches. Selection of a randomly chosen hash function guarantees good expected performance. An example of a family of hash functions is $f(x) = ax+bmod(p)$, where p is a prime and $0, 1 \ldots, p-1$ is the set of buckets and a,b are in $0, 1 \ldots, p-1$. Moreover, something has to be done so as to deal with dynamic environments such as Internet, where the number of buckets is not known always and of course it is not stable. A hashing scheme that meets the above requirements is called consistent hashing. Its properties are:

- Balance: Items are distributed to buckets randomly
- Monotonicity: When a bucket is added, the only items reassigned are those that are assigned to the new bucket
- Load: The load of a bucket is the number of items assigned to a bucket over a set of views. Ideally the load should be small
- Spread: The spread of an item is the number of buckets an item is placed in over a set of views. Ideally the spread should be small.

## 3.3 The Virtual OSGi framework

The Virtual OSGi framework is a bundle that is installed in the the local OSGi framework. It can be considered as the system bundle of the whole framework, even though every node has installed its different Virtual OSGi bundle. This is the concept of paravirtualization, which is a concept of virtualization with the difference that the guest system communicates with the host to achieve a beter performance. Furthermore, it helps to offer transparency to the user that operates the Virtual framework. Bundles are installed and managed by this bundle which is an intermediate to the interaction with the local framework, but adds also network functionality and communication with other nodes and other bundles that are located there and with services of course. Besides the startup of

the framework, it is responsible for the creation of the network when a node starts a complete Chord ring, or for the join of a node to an already existing network. It is also responsible for maintaining the network, keep the persistent connections, and do the operations of the distributed hash table (lookup, insert, delete). The Virtual OSGi framework acts as another framework above the local OSGi framework. As a result, it maintains data structures that helps an OSGi framework to operate. These data structures maintain information about the bundles that are installed in this node, the services, the node information that is essential for the network communication. When it starts, it initializes all the appropriate information for these data structures, and it propagates the request either for a network startup or for a join to the network infrastructure. Furthermore, it is responible to read the configuration file if it exists, and install or start the requested bundles, e.g. the shell bundle. Furthermore, it is responsible for assigning the global ids to the newly installed bundles and to the services. The Virtual bundle is responsible for notifying the local and the remote listeners that a change happened in a bundle or in a Service.

## 3.4 Bundles in the Virtual OSGi framework

In the Virtual OSGi framework bundles, are handled in a different way from the one in the local framework. For instance, in the virtual framework they have to be visible in every node of the system and some requests of them need network communication with other nodes of the system. A bundle is installed in the local framework. The Virtual framework does not deal with classloading of the bundles and the classes that it might request or load explicitly. It is dealing mainly with the life cycle and the service layer. However, the Virtual framework does not deal with the module layer apart from the case of remote bundles. For instance, when a node requests for a bundle that is located in another node then the creation of a classloader is inevitable which establishes a communication with the real bundle and loads class from it, in case where an application requests it. Methods from the local OSGi framework are used and the bundle object is returned from them so as to exploit the logic of the local framework that runs on each node. Apart from registering the bundle in the local framework, it is registered and in the Virtual OSGi framework so as to be accessible from the whole amount of nodes that participate on the framework. As a result, a different object from the bundle object that the local framework returns, is returned to the application so as more operations can be satisfied.

A new bundle object exists. It is a wrapper object of the Bundle object that is returned by the respective local framework. With the introduction of the new object, the interception of specific method calls is achieved, because the communication with the network should be taken into consideration. When a bundle is installed, is installed first in the Virtual Framework. Then it is installed in the local framework and finally, it is registered in the distributed bundle registy. An example that presents why the interception of bundle methods is helpful is when a bundle is used to install another bundle. If the normal bundle object was used, then the newly installed bundle could not be registered in the bundle registry. In the Figure 3.3 is presented how virtual bundle objects are connected with the local ones and how they are presented in the Virtual OSGi framework.

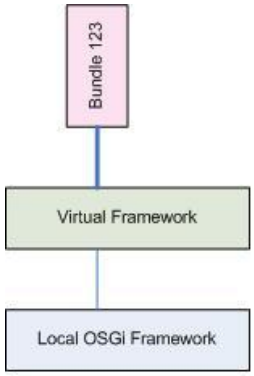



Figure 3.3: Installation of a bundle in the Virtual OSGi framework

Applications that run on the Virtual OSGi framework, communicate with the virtualBundle object which apart from the local bundle object, maintains additional information which is crucial for the operation of the system. Because of the fact that the whole system is distributed, decisions had to be taken about the information that the new object will maintain. These information are a new global id, that is assigned in the bundles because this id should be unique and maintaining the ids that are assigned from the local frameworks might lead possibly to conflicts. According to the OSGi specifications [16] each bundle has its own unique bundle id. Maintaining a global unique id is very important for the system because it is the key through which the bundles are accessed by other bundles. Apart from the new id, information about the exports and imports from the package are maintained, for the resolving procedure of the bundle. Exports, imports and require Bundle information from the Manifest file are kept also in the virtual bundle object so as the Virtual OSGi framework can collect the appropriate information for the resolve process of the bundle and make it possible to start. The resolving procedure involves network communication and it will be examined thoroughly in the Implementation Chapter. Finally, information about the Listeners that are registered from this bundle are kept also in the virtualBundle object and the nodes that have registered Listeners from the remote object of this bundle for maintainance reasons.

Consequently, besides the local bundle object, the BundleContext object that the local frameworks offer is wrapped by a new object, because of the fact that more logic had to be added to the methods that this object has. The BundleContext is the basic connection of a started bundle with the framework, because of the fact that when the start() method of the bundle is called its argument is a BundleContext and only through this can communicate with the underlying framework. So a VirtualBundleContext object is introduced, which besides the local BundleContext it also maintains additional information for the bundle that it is connected with. Moreover, the methods that the BundleContext interface offers are modified because of the fact that network logic needed to be added. As with the VirtualBundle case, the network communication is inevitable, because, for instance, acquiring or registering services have to be processed by the whole

system and this will result to the involvement of other nodes of the system.

Because the Virtual OSGi framework offers transparency and offers the whole system to the user as one entity, bundles that an application use and are located in other nodes had to be added. The application is not able to distinguish whether a bundle object that accesses is local or remote. Even though they are presented as same entities in the application, they had to be implemented in a different way and as a result, the Virtual OSGi Framework handles them differently. First of all, remote bundles must be inquired. For storing information for the bundles a bundle registry is maintained which is distributed to every node that participates on the system. The bundle registry is based on the distributed hash table, which is the network backbone for the whole system. The registration and the retrieval of the bundle is based on the basic DHT algorithms. With the use of a unique id the bundle is stored in the appropriate node. This id is the key that other nodes are using to retrieve the requesting bundle. Some of the methods are executed in the requesting node, other methods are executed on the node that the bundle is located on. It depends on what is needed to be transfered or what the respective method does.

The same process is followed for the context of the bundle. A remote bundle-context object is connected with the remote bundle object which has the same approach as the remote bundle object. It acts as a proxy to the real bundle so as to wrap local method calls to remote calls to the bundle and either return a value to the requesting node or do further processing to the real bundle. Furthermore, another difference from the bundle and bundle context object which are referred to local bundles is the addition of a classloader, which is used for an implementation of a specific method but will present it in the Chapter 4.

## 3.5 Resolving Process in the Virtual OSGi framework

Besides services and bundles which are needed to be stored in the Virtual OSGi framework, information about the constraints of a bundle are needed. In the case that dependencies exists between bundles that are stored in different nodes, the bundle that is in the resolving process has to access this information. For instance, which bundle exports a package that this bundle imports. These information will be used in the resolving process of a bundle. In the Virtual OSGi framework Clusters are introduced which is a logical equivalent of the wires that are presented in the OSGi specifications [16]. In the specifications when a bundle imports for instance a package it builds a wire with the bundle that exports this package. This bundle is also wired with a potential bundle that it exports a package that is imported from the middle one. The following figure shows us an example of wiring bundles. If a bundle imports a package from an exporter then the export definition can imply constraints on a number of other packages that the exporter depends on and therefore constrains the resolver for imports. The “uses” directive lists the packages that the exporter depends on.
In the example of Figure 3.4 a bundle imports two packages, p and t. These

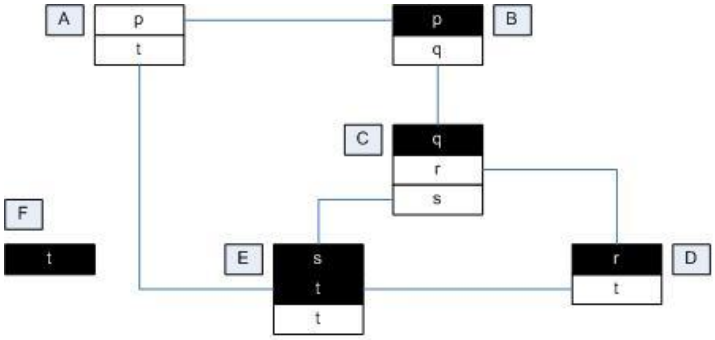



Figure 3.4: A resolving example

two packages are exported by different bundles. In this case, bundle A should be wired with bundle B because of its first constraint. Nevertheless, bundle B adds packages to the set of implied packages for bundle A, by its dependencies, and consequently package t is resolved under the implied connections. The framework must ensure that none of its bundle imports conflicts with any of that bundle's implied packages. Here, bundle A that imports package t, should be wired to package t of bundle D. Wiring this import with package t of bundle F will violate the class space consistency, because bundle A could face objects with the same class name but from different bundles. Moreover for packages that a bundle imports, version checking should be done, so as to find an exported package that satisfies the requirements that the first bundle has from the imported package.

In a local implementation of an OSGi framework, maintaining the wires is a much simpler process. Traversing through them so as to find the implied constraints of a resolving bundle is not a very heavyweight process. In a local implementation of an OSGi framework, network communication is not needed. As a result, the search for avoiding collisions from the same exported packages by different bundles can be done based on the wire paradigm. Apache Felix, for instance, maintains IWires that have information about which bundle is connected with another one that exports an import or a uses package. Nevertheless, in a distributed framework, this implementation is very costful because for every bundle a lot of wires should be maintained, possibly to different nodes. Furthermore, traversing through the wires is very resource consuming, because of the fact that it is needed to search a lot of nodes of the framework that store the installed bundles that export packages, which implies communication overhead. In addition to this, other nodes have to be capable of finding where a bundle that exports a specific package is located. This means that bundles apart from their unique id, are also hashed with the package names that are exported for instance, so as the bundle that imports packages can do a lookup for the package names and obtain the appropriate information. Unfortunately, this technique resorts to high cost of maintainance, because the bundle information will be stored in many different nodes, as many as the package exports. Besides, traversing, maintaining and sometimes rebuilding the wires is very re-

source consuming and it requires a lot of time. Furthermore, consistency issues have to be taken into serious consideration, as far the Virtual OSGi framework is a distributed system. These issues have to do with avoiding rebuilding wires concurently by two different bundles that are trying to be resolved. This procedure is done very carefully, because it is possible to result to a case that the system is not consistent or even to a deadlock.

In order to avoid these situation and to have a more efficient architecture, the wires notation of the OSGi specifications is replaced by the equivalent Clusters. The Cluster is the equivalent of the wire policy between bundles, but it is more efficient and easier to be managed, updated and deleted. The cluster is an object that stores information about bundles and their exported packages that they have. Bundles that are connected with wires, form one cluster which is located only in one node in the network, and this makes it more easily to be managed. Moreover, a lot of network overhead is avoided, because the system does not have to traverse the wires. Following this strategy, consistency issues and deadlocks can be dealt with in a better way. In the example that was presented above in the clusters case, two clusters that have information about the bundles and the exported packages exist. When bundle will try to be resolved, it will see that two clusters exist. The first one has the bundles and the packages that are wired and the second one has only bundle F with package t. During the resolving process it will select cluster one which satisfies its constraints. Consequently, it will participate to this cluster, but it will not add any more packages because it does not export anything. In the case that the new bundle exported packages, a check for conflicts with the already existing packages is made.

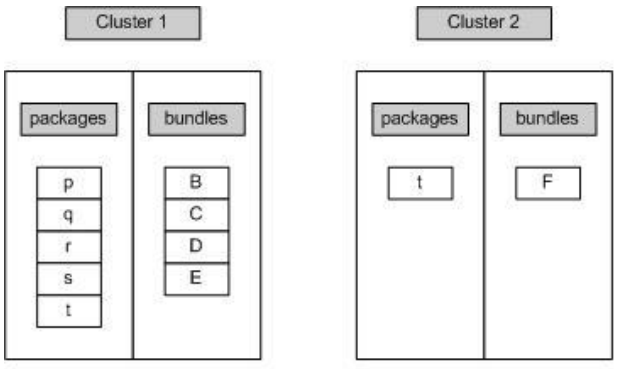



Figure 3.5: Clusters Example

In the Virtual OSGi framework clusters are treated like bundle and service objects. When a new cluster is created it is stored in the network, after being hashed and obtaining a unique id. In the clusters case two strategies can be followed. One of them was to store it in one node and the other one was to have replicas in every node of the system. Both strategies have their advantages and their disadvantages.

### 3.5.1 Replication of the clusters

A first approach is to keep a replica of each cluster in every node of the system. So when a new cluster is registered it is installed in every node, so when a new bundle tries to be resolved it retrieves the local replica of every cluster that exists in the system and does the appropriate processing so as to see whether it can be resolved or not and what are the following operations.

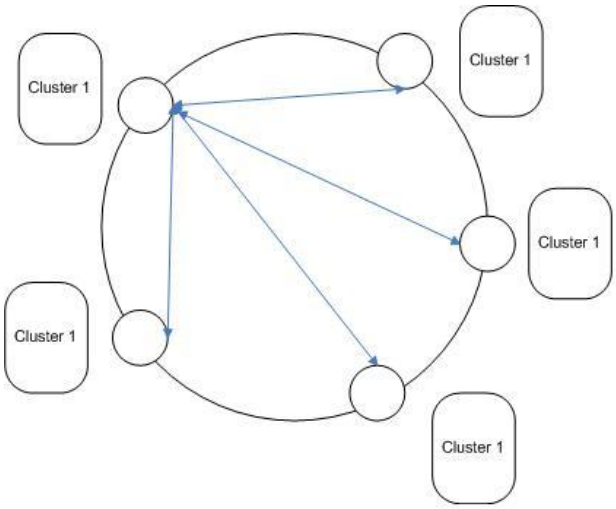



Figure 3.6: Cluster Strategy 1

What it is needed to be taken into serious consideration with this strategy is the consistency of the replicas of a cluster. A possible problem is that two bundles in different nodes are trying to be resolved concurently. They will contact their local replicas, they will find an appropriate cluster formation for their case, they possibly merge or update or even delete some clusters, which in every case they are different. So in order to have in every node the same formation of clusters the cluster selection should be treated very carefully. There should be a locking mechanism that deals with them. So every time that a bundle tries to be resolved and deals with the cluster, it should first obtain the locks of the clusters, or a global lock, then do the processing and then release the lock in case that another bundle tries to be resolved after that. A disadvantage of this approach is that every time that there is a change in the clusters, then every node of the network should be contacted so as its replicas will be updated. Practically this means that this should happen every time a bundle is installed, which adds a very high communication overhead. On the other hand, when a bundle enters the resolve state, if the lock is in this node, then it does not have to communicate with other nodes to obtain the cluster information.

### 3.5.2 Clusters Stored in one node

The second approach examined does not have any replicas. In this case when a bundle tries to be resolved, it contacts the nodes that store the clusters, and after gathering the appropriate information and is resolved, it updates the

clusters. Since the cluster is stored in one node, all other nodes are informed of the existence of a new cluster. This is done with a broadcast, but it is happening only when a new bundle is installed or it is deleted.

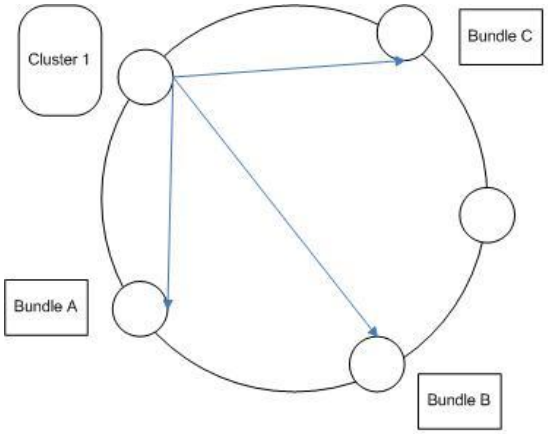



Figure 3.7: Cluster Strategy 2

In this strategy, every time that a bundle tries to be resolved, it contacts every node that stores cluster information. When collecting information about where to find the other bundles, it contacts the nodes that have them so as to install a copy of that bundle locally so as to be able to be resolved. After that it enters an update period where possibly some clusters needed to be udpated. And in this strategy, consistency should be maintained, especially in the case where two bundles concurently try to get resolved. The case of having concurrently updates of clusters have to be avoided, because it causes problems to the contents of the clusters. Hence, a locking mechanism is used.

Finally the second strategy was selected, because of the fact that it fits better to the needs of the Virtual OSGi. The first strategy is better in the case that not so many bundle installations happening. Having a large number of bundle installations leads to many updates of the clusters, and as a result to a very high communication overhead, because and all the replicas of each cluster will be updated. Even though network overhead is avoided by not having to ask for the cluster in the beginning, this cost is multiplied in the update period of the clusters (contact the whole network). If the clusters were static and not so many changes needed to be done, this strategy would be ideal, because almost none network communication is needed. Moreover, the locking mechanism has to be very sophisticated and very slow in this case, because of the fact that more or less every node of the system should be contacted as far as it maintains a cluster replica.

The second mechanism is ideal for the Virtual OSGi framework case, because by maintaining the clusters in a few nodes, only obtaining of the clusters is

needed when a bundle tries to be resolved and after that only the nodes that their cluster has suffered changes are contacted. The communication cost is much smaller in a dynamic system, because in this strategy contact of every node that participates in the system is not needed, every time that there is a bundle update. Only nodes that store clusters are needed to be udpated. Moreover the locking mechanism in this policy is much more easier and more elegant.

## 3.6 Service Registry

In the OSGi service platform, bundles are built around a set of cooperating services acquired from a shared service registry. An OSGi service is defined by its service interface and implemented as a service object. The service interface should be specified with as few implementation information as possible. The service objects are owned by a bundle, which registers or deregisters them for instance. A service can be used by other bundles, but before it has to be registered by the bundle that it belongs to. After its registration information on how to retrieve it are stored in the service registry of the system.

In the Virtual OSGi Framework the service registry has to be distributed. A service has to be accessible by every node, no matter in which node it has been registered. Furthermore, by maintaining a distibuted registry and not a centralized one, bottlenecks are avoided. For the distributed service registry the special characteristics of the DHTs about hashing and retrieving values are used. In the Virtual OSGi framework, services are stored locally in the node that the bundle is located but they are registered in the whole system. Consequently, according to whether the service is stored in the same node that the request was generated or not, two different executions treat each request. The first case is when the service is located in the local framework and as a result it will query the local registry so as to obtain the requested service. In this case a request to the framework is not needed and as a result the network is relieved by more traffic. The second one is when the service is registered by a bundle in a different node. The Virtual OSGi framework searches the network to find where the service is located. It is responsible for finding where the service is located and returns to the bundle a remote service reference object, which can be used to get the service object.

When a bundle tries to obtain a service reference object it first searches the local registry, which is a small part of the service registry of the system, so as to see whether this object is stored on this node and avoid the network overhead of messages that needed to be sent. If the object is not in this node it will eventually search the network by doing a DHT lookup operation. This operation returns the remote service object which is used for retrieving the service from the remote node. Service information are stored with the help of the hash function, some information is used in the hash function so as to produce with high probability a unique id, and following the basic DHT algorithm they will be stored to the appropriate node. A decision that needed to be taken was under what values the services will be hashed. For instance, which properties of the service will be has hashed. Services, for example, have the interface of

the service and the properties under which a service is registered, which might be updated after a period of time. Because of their properties, services offered a lot of options about what information to hash, and not bundles for example, which could be hashed only with their global id and consequently follow the normal DHT operations. The same happens with the resolving information. An option for services then, was to hash them with their name and with the properties. In this case a better distribution can be achieve for them, especially for services with the same name. For example the service ch.inf.ethz.printer, with these properties: 10 pages per second and colored printer. This service is hashed with this key: ch.inf.ethz.print#10#true which is a concatenation of the interface of the service and its properties. The problem that arises in that case is range queries. For example, if a bundle searches for a print service that is coloured and can print 10 pages per minute then it must also find a service that deals with a printer that can print 1 pages per minute. A solution for this problem is to use a hash function which is not so uniform and divide the circle of chord in spaces according to the values of each property of the service. In our example, if the above circle is divided in 8 pieces, then the service with property $pages\_per\_minute = 5$ and lower will be stored in the first piee, other services between 5 and 10 will be stored in the second piece and so on. In this case, the problem is when more than one properties exist. What is done, for example, to get a service that responds to a printer that prints $10 pages\_per\_second$ and has a $max\_queue\_delay$ of 10 seconds. How these two properties will be combined so as to store the service and retrieve the service that satisfies them
In the Figure 3.8 is presented how the connections from one node to its successor is maintained.

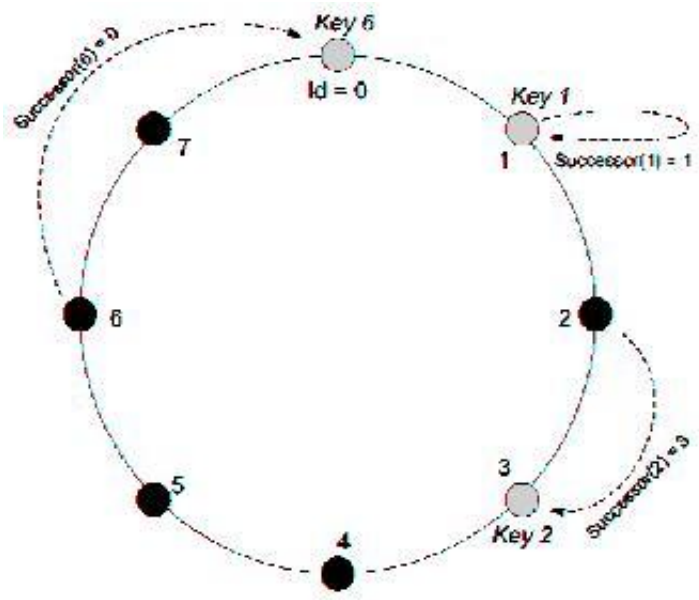



Figure 3.8: A range query example in a DHT

The challenges that arise for this policy are the following. First of all, the combination of all properties is used to store the service to a node, or it is based

only to the most important property or to all of them. Furthermore, wires to other nodes that store the service with the same name or also same other properties have to be decided whether they will be maintained. These are design decisions that are taken when this strategy prevails. A good reason to select this strategy is that a service that satisfies the filters that the requester might pose can be found immediately without obtaining or just searching for services that do not fit the filters of the requester, and avoid processing by the service author and the service keeper. On the other hand, it is a little bit difficult to implement this strategy for reasons that were presented above. Keeping a larger set of neighbors such as nodes that store services with the same name is a possible solution, but the update procedure is very costful in this case.

Another option is to build a hierarchical namespace in the flat identifier space which is not appropriate to manage multiple data structures, by partitioning the identifiers in multiple fields and then have each field identify object of the same granularity. In the case of PIER [30] (which actually operates on a system that looks like a distributed database) a namespace is a relation and is not needed to be predefined. They are created implicitly when the first item is put and destroyed when the last item expires. In PIER a resource id exists, which is generally intended to be a valued that carries some semantic meaning about the object (a value or an attribute). In this case the namespace can be mapped to the interface that the services implement or to the properties of the service. In PIER a DHT also is used, both as content addressable network for routing tuples by value and as a hash table for storing them. The query language that they used might be helpful in our case. A PIER query example is the one in the Figure 3.9.

```
SELECT R.key, S.key, R.pad
   FROM R, S
WHERE R.num1 = S.key
AND R.num2 > constant1
AND S.num2 > constant2
AND f(R.num3, S.num3) > constant3
```

Figure 3.9: A Query example

In this case it selects some attributes from two tables and joins them. In the virtual framework case an option could be a WHERE clause which will put values to the properties of the requesting service and return the service itself if the properties are satisfied or a list of service in the case that namespaces are used with multicast.

Another option for storing and searching values is the Distributed Segment Tree [22], which is a Prefix Hash Tree. This tree is based on prefix search of strings, to satisfy range queries. The data structure that is used is a binary trie built over the data set. Each node of the trie is labeled with a prefix that is defined recursively: given a node with label 1, its left and right child nodes are labeled 10 and 11 respectively. The root is labeled with the attribute being indexed and downstream nodes are labeled as above. The algorithms that the PHT supports are PHT_Lookup_Linear and PHT_Lookup_Binary. It is an application that is built on top of a DHT and its target is to support basically

range queries. In this case a data structure is built and enables efficient range representations which is crucial to range queries. This tree is a fully binary tree with some specific properties according to the space that a node is responsible for. A key is inserted not only in the leaf, but to all its ancestors (replication technique).

Another option for insertions of values in the network is Mercury [21]. Mercury supports multi-attribute ragnge-based searches. It supports multiple attributes as well as performs explicit load balancing. It creates a routing hub for each attribute in the applications schema, which is a logical collection of nodes in the system. Furthermore, to support ragne queries, each routing hub is organized into a circular overlay of nodes and places data contiguously on this ring. Mercury supports queries over multiple attributes by partitioning the nodes in the system into groups called attribute hubs (one node can be part of multiple logical hubs). Each of the specific attribute hubs is responsible for a specific attribute in the overall schema. Nodes within a hub are arranged into a circular overlay with each node responsible for a contiguous range of attribute values. Ranges are assigned to the nodes during the join process. Moreover each node has a link to its predecessor and its successor and it also maintains a link to each of the other hubs. Finally, Mercury deals with node churn and the connections that they have with other nodes in the Mercury network.

To conclude, the problem that it needed to be solved, was how to store the data in the DHT, and especially services which have properties and interfaces. One option is to be based on an exact match store, or to adopt range query based techniques that was mentioned above such as Mercury and PIER. With the second option, services can be distributed uniformly, especially services that are registered under the same interface. However, by reaching one service the node can reach and the others easily. On the other hand, the disadvantage of this strategy is that in an OSGi framework thousand of services can potentially be present, which means that many overlay networks have to be maintained during the execution of the framework, which makes the creation and consistency of the distributed hash table very complicated and heavyweight and the maintanance costs very high. Finally, services are hashed only with their interface and the properties are not taken into consideration in hash process, because not having information about the type and range that the values of a property might have made the adoption of the other techniques complicated and inefficient. Services with the same interface are stored in the same node (and to the successors of it) and when a node request for a service with the help of the filter, the node that has these services will send back the services that satisfy its filter. With the case decision, faster collection of the values that a node wants is achieved, but a loss in uniformity of services that are registered under the same interface happens.

In order to get the service object from another peer, after collecting the appropriate information R-OSGi [33] is adopted. R-OSGi has been the first aproach towards a distributed OSGi framework. R-OSGi allows a centralized application to be transparently distributed at service boundaries by using proxies. The Figure 3.10 shows how R-OSGi operates.

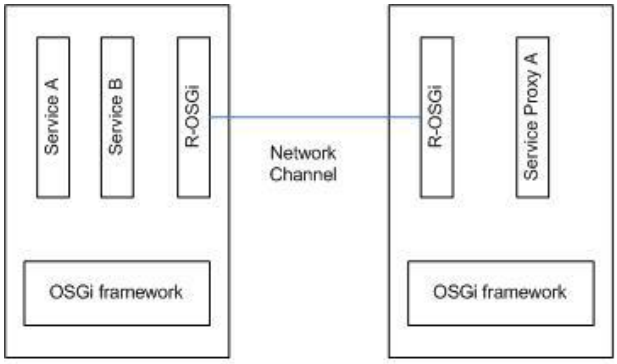



Figure 3.10: R-OSGi big picture

The basic characteristic of R-OSGi that was adopted in the Virtual OSGi framework is they way that a node fetches a services object, and the method invokation of the service proxy. R-OSGi is successfully ported to the Virtual OSGi infrastructure

## 3.7 Listeners

In OSGi frameworks, events of the following three types are supported

- BundleEvent, reports changes in the lify cycle of bundles
- FrameworkEvent, reports that the framework is started, start level has changes, packages have been refreshed, or that an error has been encountered
- ServiceEvent, reports actions concerning a service, registration, deregistration, or a change in a service property

Three different types of Listeners exist, one for each type of event. The listeners that potentially exist in an OSGi framework are *BundleListeners* and *SynchronousBundleListeners*, which they are called when a change in the lifecycle of the bundle happens. Their main difference is that the *SynchronousBundleListeners* are called synchronously during the processing of an event, but *BundleListeners* are called asynchronously.

*FrameworkListeners* are associated with framework events. ServiceEvents are delivered to *ServiceListeners* which are registered by bundles. Events in the case of services are delivered synchronously, and they are a specific type of events. A bundle that uses a service object registers a ServiceListener object to track the availability of the service object, and take appropriate action whe the service object is unregistering. In the Virtual OSGi framework, listeners are registered as in a local OSGi framework. When a bundle in a node register a Listener this one is stored in the local listener registry. What it needed to be done as an addition is to inform the nodes of the framework that there exists a listener in

this node, so as an service or bundle update for instance happen it will inform all the listeners of the framework. The Virtual OSGi framework broadcasts the existence of a listener in one node to all nodes of the framework. Each node maintains its registry of remote listeners, which keeps information about where they are located and when an update happens in this node it sends a message to this node. In order to send the event to the remote listener, this has been registered to that node, otherwise it will not be informed. In this case that the Figure 3.11 depicts a case that a problem might happen.

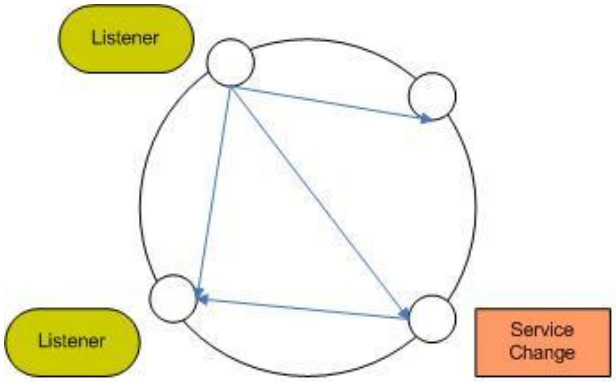



Figure 3.11: Example of informing listeners

In this example a service change happens and the Virtual Framework tries to send the event to the ServiceListeners. Just before the service change, in another node a bundle registers a new Service Listener, after that it sends a broadcast message to the other nodes of the framework so as to inform them for the new registration. When the first node will look at his remote listener registry, the broadcast will not have arrive and consequently, even in real time the listener was registered before the service changed, it did not receive the update. In order not to have a complicated clocks implementation, it is assumed that in every node that a listener was registered only when it was stored in the remote listener registry of that node. So when an event is sent, the node will search its listener registry, it will see which are the registered listeners of the system and it will send them the event.

### 3.7.1 Listener Strategy

Listeners are not hashed and stored in different nodes in the framework so as to avoid data transfer in the network, and because there was no point in hashing and inserting the listener to another node. The broadcast message which will inform the other framework nodes for the existence of a listener will be finally executed, so the hashing of this object with its interface or its another characteristic would not offer anything to the system. As a result the listeners are stored to the nodes that they were registered and a broadcast message is sent over the DHT so as to inform other nodes that a listener is located in that node. Furthermore, replication of the listener in the successor list of the node that the listener is registered is not happening, because if the node that stores

this bundle goes down, then the bundle will be lost. In case that a bundle is replicated in other nodes, then a pointer to the existing listener should be moved, so as to update or remove the listeners that this bundle or a replica of it has registered to the framework. When the listener is removed, then another broadcast message is sent, so as to inform the other nodes of the framework that this listener no longer exists, and consequently do not send events to that nodes.

If the remote context object of a bundle is used to register a listener, the same strategy is followed. The listener will be stored to the node that has this remote object and besides the broadcast message, it will send a notify message to the real bundle object so as to store that a listener has been registered under the context of this bundle and when this bundle is uninstalled, it will send a message to node that has this listener to remove it, together with the remote object.

## 3.8 Virtual OSGi and Brewer's Conjecture

Brewer made the conjecture [26], that it is impossible for a distributed system to provide the following three guarantees:

- Consistency
- Availability
- Partition-tolerance

All three are desirable and expected in real world distributed systems.
Consistency guarantees that there must exist a total order on all operations such that each operation looks as if it were completed at a single instant. This is equivalent to requiring requests of the distributed shared memory or act as if they were executing on a single node, responding to operations one at a time. Availability means that every request received by a non-failing node in the system must result in a response. That is, any algorithm used by the service must eventually terminate. Even when severe network failures occur, every request must terminate. Partition tolerance can be achieved by the network if it will be allowed to lose arbitrarily many messages sent from one node to another. When the network is partitioned, all messages send from nodes in one partition to nodes in the other partition are lost.

In the case that there are network partitions it is impossible to have reliably atomic consistent data. It is feasible, however, to achieve any two of the three properties: consistency, availability, and partition tolerance.

In this section it is discussed which two of the three guarantees are satisfied by the Virtual OSGi framework. The discussion is based on decisions that were taken in the architecture of the system. As it was mentioned above, the Virtual OSGi framework has as a network backbone a distributed hash table (DHT). Nodes are connected with a small amount of their peers and they propagate messages that they receive, according to the ids of their finger nodes and to

the id of the message. In DHTs, in order to have a network partition all nodes that are stored in the neighborhood set of one node should crash, (in the Chord network in the finger table), because otherwise the result of the failure of a lot of nodes will be the delay in the algorithm execution.

The Virtual OSGi framework offers high availability of the data that are stored in it. Services, bundles, resolving information are easily accessible by doing a lookup with the specific id, in order to reach the node that stores the required information. After that, the respective data can be retrieved by contacting the node that stores them. Furthermore, by replicating the data to the successor list, when a node that stores the location information, breaks down, then the lookup operation leads to the first alive node of the successor list of the previous node and finally it returns the appropriate data to the requesting node. With this replication technique that is offered by the DHT, it is possible to achieve high availability which is violated only when all nodes of the successor list of a node are down. These requests are synchronous, meaning that when a bundle in a node requests for a bundle or service reference object, the node sends the message to the network and it will wait for some time. After that, it will be terminated, returning null to higher level of the application which is the bundle installed in the framework. Furthermore, when a node requests something that does not exist in the Virtual framework, then the responsible node for this id, will search its cache, it will see that it does not have the requesting value and it will send back a null object. So a response gets a reply apart from the case that a certain amount of specific nodes break down.

Network partition is a case were all messages from a group of nodes to another group of nodes are lost. In DHTs network partitions try to be avoided with the use of the finger table. An example of a partition tolerant system is web caches, were replication techniques is used so as to have a partition tolerant system, but with very weak consistency. In the Virtual OSGi framework, if there is a network partition, if in the one network is the writer of the property, and in the other one is the node that stores it there exists the following inconsistency case. Another node that tries to retrieve the *ServiceReferenceRemote* object, will obtain a previous version of the properties of the service.
In the case that a partition happens and in one new network is the writer that sends an update message for a property of a service to the node that stores the RemoteServiceReference object, and in the other the requester of this object and the node that stores this information. If the update message is lost, then the requesting node that tries to get the service will not get anything. In the case of a network partition the two divided parts will continue operating as two different DHTs without being able to detect that another ring exists and merge to form the initial DHT.

In the system if all messages are delivered (within some time bound), data that will be returned in executions will be atomic. On the other hand, in the case that a network partition happens, even though, it is not very possible, data will be inconsistent and sometimes unavailable, at least to nodes that are located in a different partition of the one that the data is stored. The network partition is not very highly likely to happen because the whole amount of nodes

that are stored in the finger table of one node should be down. This amount of nodes is $O(\log N)$, where N is the number of the whole nodes that participate in the system.

The Virtual OSGi framework, tries to keep consistency and availability and leaves the network partition to the DHT characteristics. These two attributes are considered more important for the system. In an implementation of an OSGi framework, maintaining the consistency of the data is crucial, and at least it has to be maintained, in as many cases as possible. At least when the network is functional, consistency should not be violated. Concerning availability, it is also considered of the highest importance. Services, bundles and other information should be accesible from other bundles of the framework. In a distributed system availability cannot reach the availability of a local system, but it should maintain also as high as possible. In the Virtual OSGi framework if the system does not suffer a network partition it is satisfied in every case. In the case of a partition data that are stored in nodes of one partition are not accessible to the nodes of the other partitions. DHT algorithms, can guarantee that a network partition can happen with low probability. Unfortunately, in case that happens, then consistency and availability cannot be maintained, because of the reasons that were presented above.

# Chapter 4

# Implementation

## 4.1 Network Implementation

In Chapter 3 the decision about the network backbone of the Virtual OSGi framework was presented. The DHTs are the appropriate solution because they offer a fully decentralized solution so as to avoid single points of failure and bottlenecks. Furthermore, by having a structured network the cost of the lookup of the objects that are stored in the network is reduced. DHT is ideal for the Virtual OSGi framework, because with the lookup mechanisms that they offer, services, bundles and other information that are stored in the network can be retrieved easily. Below some of the most famous distributed hash tables are presented and a discussion takes places on why one of them was selected. The DHTs that were studied are the following.

- OpenDHT [39]
- Tapestry [20]
- CAN [40]
- Chord [31], KOORDE [25]
- PgridS [38]
- Applications based on DHT

#### OpenDHT

OpenDHT is based on Bamboo and it also uses other techniques such as recursive routing, proximity neighbor selection and server selection. It is used and tested in PlanetLab. A problem that PlanetLab needed to deal with, when using OpenDHT, were a few slow nodes, which made the system operate more slowly. The point is that the scheduling latencies inherent in a shared testbed increase the unpredictability of individual machines' performance by several orders of magnitude.

### Basic algorithm

The Key space of Bamboo is $2^{160}$. Each node is assigned an identifier from this space uniformly at random. For fault tolerance and availability, each key-value pair is stored on the four nodes that immediately precede and follow k. These eight nodes are called the replica set of k, denoted R(k). The node numerically closest to k is called its root. Moreover, each node for each prefix (base 2) of the node's identifier, has one neighbor that shares that prefix but differs in the next bit. This group is chosen for proximity. Messages between OpenDHT are sent over UDP, but there exists a congestion-control layer that provides TCP-friendliness and retries dropped messages. This layer also exports to higher layers an exponentially weighted average round-trip time to each neighbor. In order to put a key-value pair(k,v) a put RPC is being sent to an OpenDHT node of its choice, which is called the gateway. Gateway leads a message greedily through the network until it reaches the root for k, which forwards it to the rest of R(k). When six members have acknowledged it, then the root sends an acknowledgment back. Enhancements that can be done in the basic algorithm of the OpenDHT are the following. Delay-aware routing, iterative or recursive routing, which will be presented and which one was selected in our case in a later section and finally maintaining multiple gateways.

### Tapestry

Tapestry [20] is a wide area location and routing infrastructure. It assumes that nodes and objects in the system can be identified with unique identifiers (names), represented as strings of digits of radix b. Identifiers are uniformly distributed in the namespace. Each Tapestry node has pointers to other Tapestry nodes, as well as mappings between object GUIDs and the node-IDs.

Tapestry routing infrastructure is an overlay network between participating nodes. Each node has links to other nodes that share the same prefix with one. Neighbor links are labeled by their level number which is one greater than the number of digits in the shared prefix. Nodes in Tapestry maintain forward and backward pointers. Neighbors are grouped into neighbor sets, which are based on prefixes also. Every node maintains a routing table at each level, up to the maximum length of node-IDs. Tapestry takes into consideration a metric space which implies that in the routing table a node is the closest according to this metric. Object are being mapped to root sets of nodes which are called MapRoots($\psi$). Queries are routed towards one of the root nodes, along neighbor links until they encounter an object pointer for the requested object $\alpha$, then route to the located replica. The root set is unique, regardless of where it is evaluated in the network. In the case that the size of the root set is larger than one, object queries can be retried and tolerate faults in the Tapestry routing mesh. Tapestry nodes maintain additional root links that fill holes in the routing table. In Plaxton, Rajaraman and Richa (PRR) scheme, MapRoots($\psi$) produces a single root node which matches in th largest possible number of prefix bits with $\psi$. The PRR scheme specifies a corresponding function as follows: the neighbor sets, N, are supplemented with additional root links that fill holes in the routing table. To route a message toward the root node, PRR routes directly to $\psi$ as if it were a node in the Tapestry mesh. Assuming that the supplemental root

links are consistent with one another, every publish or query for document $\psi$ will head toward the same root node. This process is called surrogate routing. In order to deal with these, difficult to maintain pointers, Tapestry relies on information local to each node and already present in the routing table. Rather than filling holes in the neighbor tables, routing is implemented around them. When there is no match for the next digit, routing to the next filled entry in the same level of the table is done.

#### Insertions and deletions of nodes

The process that is followed when a node wants to join the Tapestry network is the following. First of all the new node contacts its surrogate node, the node whose id is closest to its id. After contacting it, it gets a copy of its neighbor table, because the neighbor tables should be filled. Then it contacts the subset of nodes that must be notified to maintain consistency. Finally, when the multicast is finished the node is fully functional, though its neighbor table may be far from optimal. Every node that is on the path between the object's server and the object 's root must have a pointer to the object. Two failure cases exists one for correctness and one for performance. Node insertions should also be deal with redistribution of the object pointers and with the fact that object will remain available. Deletions are of two kinds, voluntarily and involuntarily which are handled lazily. That means that when a node notices that some other node is down, it does everything it can in order to fix its own state, but does not attempt to notify any other node for state changes. When a node detects a faulty node, it should first remove it from its neighbor table and then find a suitable replacement. If deleting a node leaves a hole in the routing table then it contacts the deleted node's surrogate which either replies with a replacement node or performs a multicast to all nodes sharing the same prefix as the dead and the other node.

#### Content Addressable Network - CAN

CAN [40] is another DHT proposal. As every DHT, the basic operations performed on CAN are insertion, lookup and deletion of (key, value) pairs. CAN is able to offer efficient insertion and reliable and scalable indexing. Example applications that it can be used, are a distributed DNS system or a large scale storage system. CAN is composed of many individual nodes. Each CAN node stores a piece of the entire hash table. Furthermore, a node stores information about a small number of adjacent zones in the table. Requests for a particular key are routed by intermediate nodes towards the node whose zone contains that key. Design of CAN is completely distributed, scalable and fault tolerant. Can does not need any form of a hierarchical naming structure to achieve scalability and can be implemented entirely at the application level.

#### Design of CAN

The design is based on a d-Dimensional Cartesian coordinate space on a d-torus. The entire coordinate space is dynamically partitioned among all the nodes in the system such that every node owns its individual distinct zone within the overall space. The coordinate space is completely logical and it is not a physical coordinate system. To store pairs of key-value(K,V) there is a mapping of these

pairs to a point P in the coordinate space using a uniform hash function. The corresponding pair is then stored at the node that owns the zone within which the point P lies. Routing in CAN has to do with following the straight line path through the Cartesian space from source to destination coordinates. The routing table of a CAN node contains an IP address and virtual coordinate zone of each of its immediate space, two nodes are neighbors if their coordinate spans overlap alond $d-1$ dimensions and differ along the remaining one. Using the neighborhood coordinate set, a node routes a message towards its destination by simple greedy forwarding to the neighbor with coordinates closest to the destination coordinates. For a d-dimensional space partitioned into n equal zones, the average routing path length is $(\frac{d}{4})(n^{\frac{1}{d}})$ hops and individual nodes maintain $2d$ neighbors. This means that, by adding more nodes, the state of each node does not increase and the average path length grows as $O(n^{\frac{1}{d}})$. When a node enters the CAN network must allocate its own portion of the coordinate space. This is done by an existing node which splits its allocated zone in half, retaining half and passing the other one to the new node. The process has three steps. First the new node must a find a node already participating in CAN. After that, with the help of the CAN routing mechanisms, it must find a node whose zone will be split. Finally, the neighbors of the split zone must be notified so that routing can include the new node

In order to maintain the consistency of the network, messages are sent periodically to the neighbors giving its zone coordinates and a list of its neighbors and their zone coordinates. The prolonged absence of an update message from a neighbor signals its failure. After that it initiates the take over mechanism and starts a takeover timer running.

### Chord

The basic characteristics of current P2P systems are the following. Redundant storage, permanence, efficient data location, selection of nearby servers, anonymity, search, authentication and hierarchical naming. Chord is a DHT proposal that is designed to offer the necessary functionality while preserving maximum flexibility. It uses consistent hashing and it provides unique mapping between an identifier space and a set of nodes. A node can be a host or a process identified by an IP address and a port number. It also addresses these. Load balance (distributed hash function), decentralization (fully distributed), scalability (cost of a lookup grows as the logarithm of the number of nodes), availability (automatically adjustments) and flexible naming (no constraints on the structure of the keys)

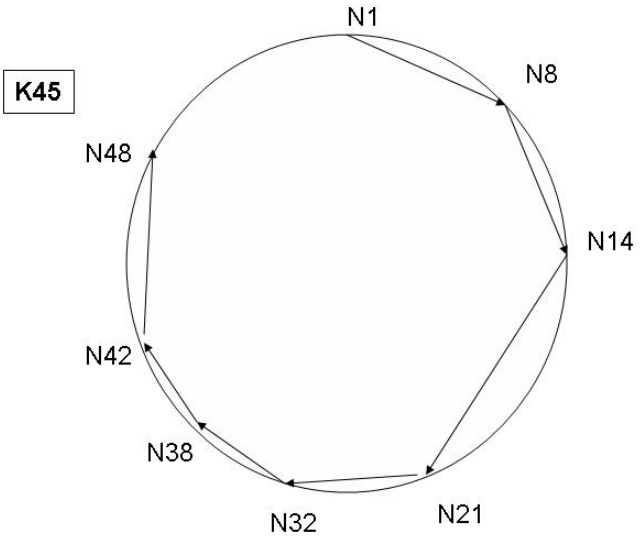
Some examples that Chord can be extremely useful are, cooperative mirroring, time shared storage, distributed indexes and large scale combinatorial search. Chord is efficient. The determination of the successor of an id requires $O(\log N)$ messages to be exchanged with high probability where N is the number of nodes participating in the system. Adding or removing a server from the network can be accomplished, with high probability at a cost of $O(\log^2 N)$ messages.

### Chord Protocol

The basic operation of the Chord protocol is that given a key, it maps it onto a node. Each node maintains information about only $O(\log N)$ nodes of the network. It uses consistent hashing(SHA-1), which with high probability balances the load. IP address of a node is usually used for node hashing. Key identifiers are produced by hashing the key itself. Identifiers are ordered in a circle modulo $2^m$ and the key is assigned to the first node whose identifier is equal to or follows k in the id space. This node is called the successor of k (it is found by moving clockwise to the circle Chord Ring). The Chord Theorem says that each node is responsible for at most $(1+\epsilon)\frac{K}{N}$ keys and when a $(N+1)$ node joins or leaves the network, responsibility for $O(\frac{K}{N})$ keys change hands. In order to find a key each node maintains additional routing information apart from the successor of it. Furthermore, each node maintains a routing table with up to m entries called the finger table. The $i^{th}$ entry of the table at node n contains the identity of the first node s that succeeds n by at least $2i-1$ on the identifier circle. $s = successor(n+2i-1)$. This node is called the ith finger of node n. The first finger is the successor of the node. Since each node has finger entries at power of two intervals around the identifier circle each node can forward a query at least halfway along the remaining distance between the node and the target identifier. As a result the number of nodes that needed to be contacted are $O(\log N)$.
In this table it is presented what network information stores each node.

| | |
|---|---|
| finger[k] | first node on circle that succeeds $(n+2^{k-1}) mod 2^m, 1 \leq k \leq m$ |
| successor | the next node on the identifier circle, finger[1].node |
| predecessor | the previous node on the identifier circle |

Chord is responsible for finding the locations of keys, how new nodes join the system, how to recover from failure of existing nodes. For converging to a stable state it has a stabilization protocol, which run in every node and helps it find whether nodes from its finger table have been added or moved. In the Figure 4.1 the lookup algorithm of Chord is presented.



### Stabilization Protocol

Chord also deals with nodes that leave and join the network even though they do this intentionally or not. It also has replication of the values that are stored to the network by maintaining a successor list which contains the node's successors.

Based on Chord a lot applications can be implemented like layers. For example a group of cooperating users can share their network and their disk resources. A good extension of Chord is DHASH (distributed hash table). Its interface is DHASH::insert and DHASH::lookup. It also monitors node joins and leaves so as to maintain the invariant that values are stored at the successor of their keys. It can also exploit the properties of Chord to achieve greater reliability and performance. Finally, to achieve load balancing, maps document identifiers to a list of IP addresses where that document was available. The basic API for the stabilization protocol, and for the creation or join in the Chord network is the following

- create(), Creates a new Chord ring
- join(n'), Joins a Chord Ring containing node n'
- stabilize(), called periodically and verifies n's immediate successor and tells the successor about n.
- notify(n') notifies for a new successor
- fix_fingers(), called periodically and refreshes the finger table entries. It stores the index of the next finger to fix
- check_predecessor(), called periodically and checks whether the predecessor has failed.

### KOORDE

KOORDE is based on Chrod and de Bruijn graphs [25]. In de Bruijn graph, each node has a unique d-bit ID. D is the number of dimensions in the de Bruijn graph. Each node is connected with nodes $2i$ and $2i + 1 modulo 2d$. The routing in the de Bruijn graphs is done by shifting the bits of the destination ID. The routing mechanism is the following. If node a wants to route a message to node b. Its id is shifting (one bit at a time) and the message is sent to the respective node, up to when the id of the message has the same value with the id of node b. The main difference with Chord is that instead of finger pointers, nodes have de Bruijn pointers.

## KADEMLIA

KADEMLIA is another well known DHT which is used in commercial applications such as the famous peer to peer file sharing mechanism emule [8]. The basic difference of KADEMLIA from the other DHTs is that it uses the XOR metric fro distance between points in the key space. Each Kademlia node maintains triples IP address, UDP port, Node ID $fornodesofdistancebetween2^i$ and $2^{i+1}$ from itself, where $0 \leq i < 160$. These are called k-buckets where k is a

system-wide replication parameter and is chosen such that any given k nodes are very unlikey to fail within an hour of each other and they implement a least-recently seen eviction policy. The KADEMLIA protocol has four basic RPCs PING, STORE, FIND_NODE, FIND_VALUE. These are recursive procedures and they try to find a set of nodes which are closer to the id that they search.

#### PgridS (peer Grid Service Discovery)

PgirdS is an example of an overlay network which is based on the deployment of more that one DHTs. In PgridS exists a two search space, one for attributes and one for values. It implements the DHT protocol into every space to build overlay networks. AON (attribute overlay network) and VON (value overlay network). AON serves as an entrance in VON.
When a node joins the network, it first joins AON and if it publish some service it will join VON corresponding to attribute service. It should locate the node who is in charge of the corresponding value. In terms of replication The successor or predecessor information in a VON should be replicated into some other nodes so as to avoid when it fails to lose the appropriate information and the message can be routed to those nodes. The DHT that is used here is CAN to support the appropriate replication.
This is a good application example where DHTs can be used in order to store and call services. In this case there is a discrimination concerning the arguments of the service and the values of them. Basically this case is not very helpful in our case, because the type of the services that are stored in the network is not known and as a result discriminate the network according to the arguments of one service and the values of it.

#### Example of Applications based on DHT

#### SOMO Self Organized Metadata Overlay [42]

SOMO must be embedded in the hosting DHT but it also should not deteriorate its specific protocols and performance. It also grows together with the DHT system and it is fault resilient and accurate to the way it disseminates the information.
SOMO can gather and disseminate information in $O(\log N)$ time, it is simple and flexible. Any object of a data structure can be considered as a document, and as long as it has a key, that object can be stored and retrieved from a DHT. Objects related to each other via pointers, so to traverse to the object b pointed by a.foo for example a.foo must now store b's key instead. The sufficient conditions are. First, each object must have a key, obtained at its birth and second, if an attribute of an object is a pointer, it is expanded into a structure of two fields(a.foo.key and a.foo.host)
A data structure is called distributed in a hosting DHT a data overlay. Its main difference from the traditional overlay is that traversing from one entity to another uses the free service of the underlying P2P DHT.
With SOMO is possible to host any arbitrary data structure on a DHT in a transparent way. Moreover fault tolerant techniques can be added to the whole application. SOMO operates above a DHT, it is as scalable as the DHT, is fully distribured, completely self-healing and as accurate as possible of the metadata gathered.

Many topologies can be used such as a tree of k degree where the gathering is being done from the leaves to the root and the disseminations oppositely. In this topology these phases are $O(\log_k N)$, where N is the total number of entities. SOMO nodes has information about the key and DHT node that stores the SOMO node s is called the DHT_host(s). It has also information about its region which is divided by a factor of k, each taken by one of its k children which are pointers in the SOMO data structure. SOMO builds is hierarchy together with the DHT. To gather system metadata a SOMO node requests report from its children. The leaf SOMO nodes simply get the required info from their hosting DHT nodes. The maximum delay for this procedure is $O(\log_k NT)$. This bound is derived when flows between hierarchies are completely unsynchronized. On the other hand dissemination is the reverse procedure. Data tickle down through the SOMO hierarchy towards the leaves. The performance is similar to gathering. Dissemination can piggyback on the return message in the gathering phase or to query the tree. The whole SOMO hierarchy can be recovered in $O(\log_k N)$ time. SOMO can be used when the health of a large distributed system has to monitored, or when powerful nodes in the network called supernodes have to be found.

#### Distributed Object Location in a Dynamic Network

To achieve locality optimizations, it is important for the routing process to do as few hops as possible and these hops to be as short as possible also. Properties that are requested from a location-independent routing infrastructure are, Deterministic location, Routing based on locality metrics and load balance

Several proposals exist such as CAN, Chord and Pastry. Their main characteristic is that they distribute information over a large number of nodes, they can find an object in polylogarithmic number of overlay hops. Unfortunately, while these approaches use a number of overlay hops, the actual latencies incurred by queries can be significantly more than those incurred by finding the object in the centralized directory. Tapestry on the other hand based on Plaxton, Rajaraman and Richa (PRR) yields routing locality with balanced storage and computational load.

From all the available distributed hash tables Chord was selected for this thesis, because of the following reasons. First of all, Chord is quite simple comparing it with other DHT proposals such as Pastry or Tapestry. The finger table that maintains for routing has information only about nodes and avoid the complicated prefix matching, or the two dimensional division of the space in CAN, based on whether nodes join or leave the network. Moreover, Chord has better performance than many other distributed Hashtables. In CAN, for example, each node maintains $O(d)$ state and the lookup cost is $O(dN^{\frac{1}{d}})$, where d is the number of dimensions that the space is divided to. In order to be as fast as Chord, CAN should have $d = \log N$ dimensions, but in a non static network it is very difficult to happen. Finally, Chord offers correctness, through the basic algorithm and with the help of replication of data and the stabilization procedure that happens after specific period of time. If one node requests for data and these data exists on the system, then it is guaranteed that it receives this data. On the other hand, networks like Freenet even though they provide

anonymity, they are based on search for cached copies. Chord, although it does not provide anonymity its lookup operations run on predictable time and always results in sucess or definitive failure. Comparing Chord with Tapestry, its main advantage is that it is simpler, and it more capable of dealing with node churn in a better more efficient way. On the other hand, Tapestry is better than Chord in the fact that it ensures that a lookup will never travel further in network distance than the node where the key is stored. The disadvantage of it is that the join protocol is more complicated, because it needs to collect information from nodes that are visited by the join message.

Selecting the appropriate DHT is not the final step. A lot of other design decisions have to be made so as to achieve the appropriate result. The first decision was whether to use an existing Chord [31] implementation or to implement a new one from scratch. An example implementation is Open Chord [6] which implements the basic Chord algorithm and the stabilization protocol as they were presented in the relevant paper. The basic features that Open Chord offers is that it stores any Java Serializable Object within the network. It provides the possibility to create own key implementations used with DHT by implementing an interface of OpenChord API. Furthermore, it transparently facilitates replication of entries in DHT and maintains the Chord routing. In addition to these, it provides two protocols for communication between chord nodes, one for local method calls which can be used to create a dht within one Java Virtual Machine for testing and visualization purposes and one for Java sockets which creates a dht distributed over different nodes (JVMs).

Open Chord is not very demanding in requirements. The only things that it needed was Java platform Standard Edition Development Kit, Apache Ant build tool and a library from Apache log4j, for logging. On the other hand, even though Open Chord was very helpful and simple, a decision was made to implement Chord protocol from the beginning so as to be able to understand completely how distributed hash table operates and how it deals with lookup, insert and delete requests. Furthermore, because of the fact that a control over the connection layer was needed and because of the fact that local query processing had to be added, a completely new Chord implementation was selected. The local query processing is applied to services in the case that a filter has to be applied to their properties.

### 4.1.1 Communication

In order to create a network, the beginning node (bootstrap node), initializes the network. This is the first node that other nodes must know so as to join the network, but it is not the only one. In the beginning, each node initializes its basic data structures which are needed for the consistency and the maintanance of the network such as its finger table, successor list, successor and predecessor. Furthermore, it initializes data structures that are responsible for the Virtual OSGi framework. For instance, each node maintains a service registry, a bundle registry, a cluster registry for the clusters of the system and a cluster registry for the clusters that are stored in this node and a listener registry. Each node maintains a ChannelFactory which manages the connections of this node. In the

beginning it starts a thread, which is responsible for accepting connections from other nodes. When a node wants to communicate with another one, establishes a TCP channel, through which data are sent and received. Every node maintains a channel registry, which stores information about which nodes are connected with this one. These nodes might be, nodes from the finger table, from the successor list or other nodes from the network, from which a bundle in this node might have requested another bundle or a service. Usually these channels remain open, for possible future communication. In TCP, establishing a channel with the three way handsake which requests the three steps by sending SYN, SYN-ACK, ACK, before sending any data, is expensive. As a result, by maintaining these channels the slow establishment of the connection is avoided. The channel registry is queried when the node wants to send a message to another node. If the channel already exists, it sends the message through this one, avoiding the creation of a new channel. In case that it does not have the channel, it establishes a new connection and sends to the other node a small register message, so as and the other one will store the channel to its channel registry for later use. In the Figure 4.1, the gerenal idea of receiving and sending messages to a node is presented. Each node has its acceptor thread, which initializes a Server socket. When it receives a messages from the TCP channel it passes it to the a receiver thread for this channel and it processes it in order to check whether it is intended for this node or not. If this message is intended for this node it checks what of message is, and does the appropriate processing. For instance, if it is a request service reference remote message, it replies to the requesting node with the appropriate object.

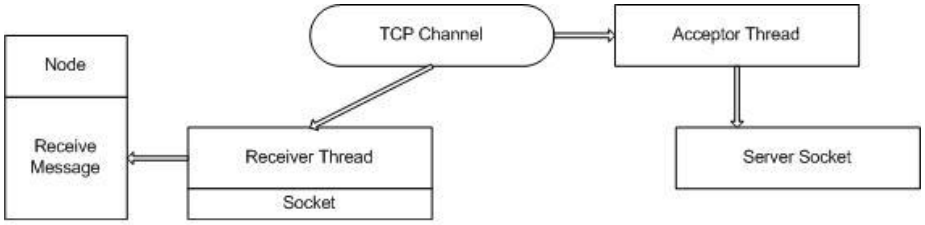



Figure 4.1: Communication between nodes

For connections between the nodes TCP connections is selected instead of UDP. Even though TCP connections are heavier than UDP (three packets for setting up a socket, without sending any data, offers reliability and congestion control, making it as a result heavier), TCP was selected for the following reasons. First of all, reliability was critical for the implementation of the Virtual OSGi framework. Besides the communication that had to do with the network creation, maintanance, and convergence to a stable state, a lot of messages concerning the Virtual OSGi framework needed to be exchanged. Bundle requests, services, or requests for informations about the dependencies of bundle so as to be resolved, are examples of that kind of messages. Besides reliability, the other two basic characteristics of TCP which are the maintanance of the order of messages in the sender and in the receiver, and, the fact that duplication does not exist in TCP. On the contrary, UDP does not offer reliability for the deliverance of a packet and suffers from worse packet loss than TCP. This was not acceptable in the Virtual OSGi framework because the loss of a packet of bundle that one

node request from another one will lead to a lot of packets that was sent for this case, useless. Consequently, the whole procedure of sending the bundle to the requesting node has to be done again.
When a node wants to join the network it has to know one node that already participates on it. This can be the bootstrap node, or any other node in the network. This node is responsible for routing a new_node message to the network, so as the predecessor and successor of the newly inserted node will be found and informed for its existence. In addition to this, a check in the data of the successor of the new node has to be done so as to check which data of the service, bundle, cluster registries have to be moved to the new node, according to the ids that are used for storage in the network. These ids are based on hashing of some values of these objects. In the Chord Protocol the node that is responsible for storing the data is the one whose id is the succeeding of the id of the data, so a checking of the data of the node is crucial to maintain consistency. These should be informed about the changes in their data structures and after that the new node should gather the appropriate information about the other nodes of the network, or at least some of them. As a result, it fills each basic DHT data structures which are the finger tables and the successor list. On the other hand, and the remaining network is also informed about the presence of a new node and the data structures of some nodes are also updated. In order to achieve this, two strategies can be adopted.

The first one, is to wait up to when the stabilization period starts for each node and this node is responsible for finding that in an entry of the finger table a change happened. The other one is, when the node starts sending messages so as to fill its finger table, to send concurently inform messages that are received by the nodes that have to update their finger tables, and as a result retrieve the new network state as soon as possible. The tradeoff between these two strategies is that with the first one messages are saved, because a "lazy" update of the network is happening, and the responsible nodes are updated after a period of time, but with the second one more messages are sent but the network updates its nodes as soon as possible maintaining the consistency of the network. The second strategy was selected, because bringing the network to its appropriate state is crucial so as to handle in a better way continuous node churn. Sending more messages is not so important than having a consistent state of the network fast, and as a result avoiding routing problems and routing inconsistencies. The new node calls update function on existing nodes that must point to the new node. These nodes are in ranges $[j - 2^i, pred(j) - 2^i + 1]$. The number of nodes that have to be updated are $O(\log N)$. Figure 4.2 shows what happens when the update function is executed. In this case Node with id 28 is inserting in the network. It tries to inform the node whose id is 8 because it should replace the entry in its finger table in position $8 + 16$ the node with id 32 with the newly inserted node.

### 4.1.2 DHT messages

The communication in the distributed hash table network is done through TCP connections through exchange of messages. The basic categories of these messages are the following. First of all a register Message exists, which is sent from one node to another for channel information, when it wants to establish a con-

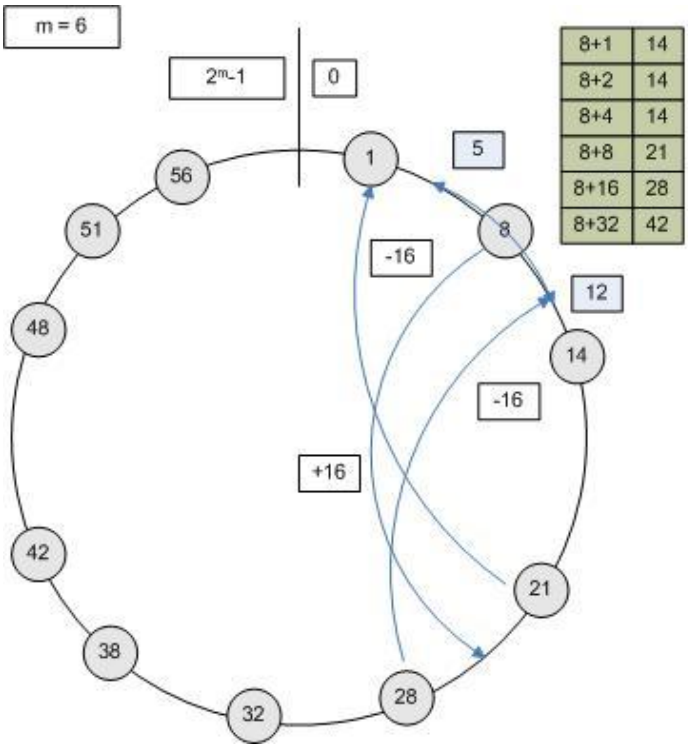



Figure 4.2: Example of a node that joins the network

nection. Other messages are Chord Initialiaze Messages. These messages are sent in the beginning, from/to a node which joins an existing network. These messages have to do with the discovery of the successor and the predecessor of the new node, the nodes that are parts of the finger table and the successor list, the data that they should be stored to the newly inserted node. In addition these message, there are the messages from this node to the whole network so as to inform other nodes that there is a change in their finger table. Besides Chord Initialize messages, Chord Stabilization Messages also exist, which are responsible for keeping the network in a normal state even though nodes leave the network, without any notification. These messages are more or less the same with the Chord Initialization Messages. Furthermore, Virtual OSGi Messages are implemented. These messages have to do with the OSGi specifications and the method that a framework should provide to the bundles that are installed to it. For example bundle request messages, in case that the bundle is installed in a different node. Moreover Services Messages should be provided, for the same reasons that exists messages for bundle communication. When a bundle requests a service, it should send a search message to the network so as to find the ServiceReference object and then get also the service. The basic types of messages are the following. First of all, Bundle messages, which implement the remote calls of the methods of the Bundle and BundleContext interfaces. Furthermore,Service messages, which implement service calls, serviceReference calls, and service updates.In addition to these messages, Virtual OSGi Resolving Messages are implemented. These messages are also part of the Virtual OSGi framework and their target is to do the resolving process that in the new specifications has changed. These messages send information about wiring bundles, updates of clusters and of course, creation and deletion. Furthermore, Listener messages exist, which are used for communication with the listeners (Framework, Bundle, Service). There are listener register Messages and listener informed messages. Finally, R-OSGi Messages have been ported in the Virtual OSGi framework, which were adopter from the R-OSGi project. In this framework, they cooperate with existing ifrastructure so as to help bundles call services from remote nodes, because in the other framework a bundle can find a ServiceReference object but there was no support for getting the service and calling its methods.

In order other nodes can be capable of identifying what type of message is sent, each message has a short number in the head of the message which identifies the type of the message. when a node receives a message, it checks the message id and does the appropriate processing. After that, it should check whether it is responsible for the process of this message (check the id of the message), and if it is not, it propagates it to another node from its finger table, which is the closest preceding node for the id of the message. Moreover, every message maintains a transaction id, which actually is used in messages that participate in synchronous calls, and is used for retrieving the appropriate result, after another thread has inserted the reply of the message in the receiving queue.

In the messages that are send, even though complex custom object should be transferred,objects are decoupled to their fields and are sent either as primitive types or String objects, where it was possible, so as to avoid as much network

data transfer as possible. In the receiving node, when the message is received, the appropriate objects are rebuild, according to the type of message.

### 4.1.3 Consistent Hashing

With the help of the hash function unique ids are assigned to every object that participates in the network. First of all ids are assigned to the nodes that consitute the system. In order to produce this id, its IP address is hashed together with the port that they listen for connections. This id is unique with very high probability.

An example of a hash function the Unit Circle Random function. The Unit Circle Random is one of the most famous hash functions and it is actually called UCrandom. C is the circle of unit circumference. By changing the basic scheme consistent hashing can be used in real life, avoiding the use of real numbers and the use of infinite bits. The use of limited precision numbers is very helpful, because in this case the values differences in the unit circle can be mapped avoiding the use of real numbers. The actual mapping in the unit circle is $|I|+m|B|$ random points on the real circle. $O(\log |I|+m|B|)$ bits of precision can be used, to maintain the ordering on a set of $|I|+m|B|$ random points. Another very famous Hash function, which most of the distributed hash tables use is SHA-1. The original specification of the algorithm was published in 1993 as the Secure Hash Standard, FIPS PUB 180, by US government standards agency NIST (National Institute of Standards and Technology). SHA-1 produces an 160-bit (20 byte) message digest used for creating unforgeable digital signatures, usually. The algorithm is slower than MD5, but the message digest is larger, which makes it more resistant to brute force attacks, which choose messages at random in an attempt to generate the same message digest. SHA-1 is also used to digitally sign jar filest in our case is used for the id generation. PGP uses SHA-1 for digitally signing email. It is computed with the use of a MesssageDigest object. Java offers the package java.security.MessageDigest, which produces the digest as a byte array of an argument, for example a String object. The fact that it uses 160-bit digest, the possibilities of ids are $2^{160}$, which is a large number to achieve uniqueness, which is crucial for the generation of ids concerning bundles, services and of course the nodes that participate in the framework.

In addition to these because of the fact that the DHT is used as a registry for objects that exist in the framework, specific objects needed to be hashed. Specifically, the object that are needed to hash are ServiceReference, bundles and wiring information. Each node maintains in it its storage a small part of these registries. First of all, a ServiceRegistry, which is responsible for storing information about where a ServiceReference object that maps to a Service that implements a service interface is located. A BundleRegistry also is implemented, which is responsible for storing information about the location of the bundles. Furthermore, a ClusterRegistry is implemented and it is responsible for maintaining information about the wiring information for the resolving phase of the bundle. Finally, general information about objects that are stored in other nodes of the system. The whole number of clusters that exist in the system, and of course information on how to retrieve them (the name of the

cluster for example). Furthermore, it should keep information about the listeners (Framework, Bundle, Service) that are installed in the system, so as when a change happens, to send to every listener in the system them the appropriate message, even though it should be synchronous or asynchronous.

As a result ids have to be produced for these entities. The next step was to decide what fields of this object should be hashed, so as the store and the retrieval is rational the operations and method of the OSGi framework should be taken into consideration and concentration on its API have to be made so as a reasonable characteristic of this object for hashing will be selected. Services, for instance, are hashed with their interface and with this id, are stored in the network, because of the fact that, the registering and the retrieving of a service object is done through the interface, which every service should maintain. In the case that a service is registered under more than one interface a slightly different strategy is followed and it will be examined in a later section. This id is not unique because more than one services is possible to implement the same interface, but this is not a problem in the Virtual OSGi framework, because other characteristics of the service are used to offer uniqueness, such as their service id. Concerning bundles, they are hashed with their unique global bundle id (OSGi specifications). Consequently a bundle can be retrieved from any node that participates in the framework, only if it knows its global id. In the Figure 4.3 and in the Figure 4.4 an example of an insertion of a ServiceReference and of the lookup of this ServiceReference object is presented. The ServiceReference object needed to be stored in the appropriate node, so as other nodes will be able to retrieve it by routing a message with the correct id to the correct node.

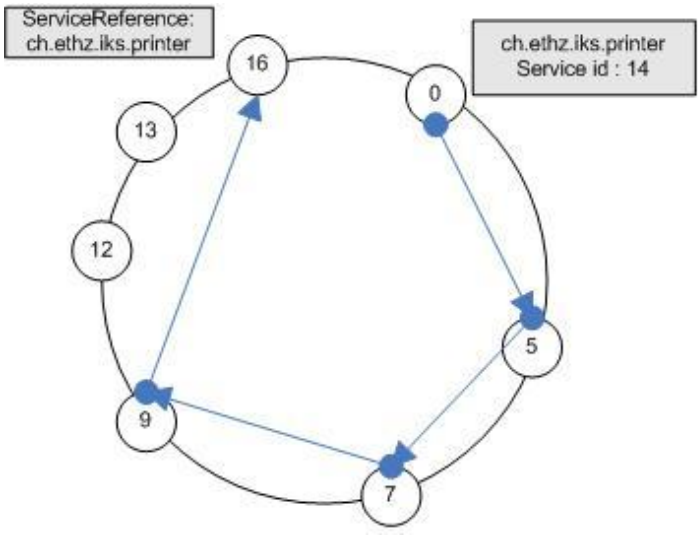



Figure 4.3: Registering a service

In this example node with id 0, wants to register a service which implements the interface 'ch.ethz.iks.printer'. The interface of the service is hashed and the id that is produced is 14. This object should be stored in the node whose id is

preceding of the id of the service. This node is the one with id 16. The routing of this object is presented on this figure. In the next one, a bundle from another node, the one with id 12, request a ServiceReference object that implements interface 'ch.ethz.iks.printer'. The id which will be produced by this interface, is 14, so the node will do a lookup for this id, and as a result it will contact the node with the preceding id of 14.

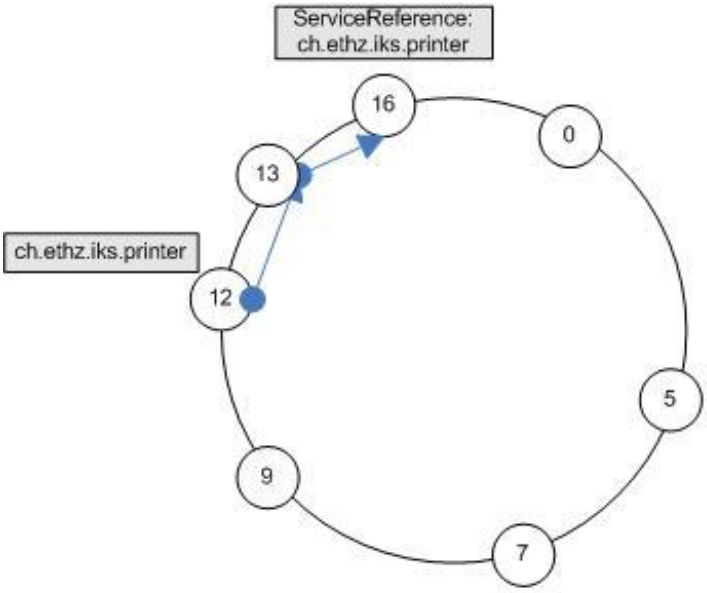



Figure 4.4: Retrieving a service

### 4.1.4 Chord

The communication in the Virtual OSGi Chord implementation is done through exchange of messages through channels and they are established when a node wants to send a message to another node.

#### Routing in Chord

Another design decision that had to be made, and the general Chord protocol does not justify is whether the routing will be iterative or recursive. With recursive routing, each request is routed through the network to the appropriate node that stores the object that the client has requested. In contrast, gets can also be performed iteratively, where the requesting node contacts each node along the route path directly. Figure 4.5 and Figure 4.6 show how these two routing techniques work. In both cases, node 5, requests a ServiceReference object which implements the interface 'ch.ethz.iks.printer' and its id is 14 and is stored in node with id 16.
In the Figure 4.5 is presented how recursive routing propagates the request for the message in the network.

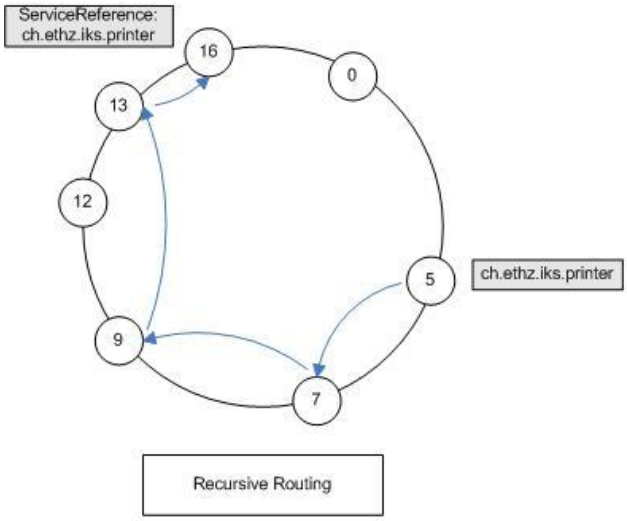



Figure 4.5: Recursive Routing

And in Figure 4.6 is presented how the intermediate nodes of a path in iterative routing send back a reply with information about the routing path. The requesting node contacts every intermediate node, so as to gather information about the next node in the path and approach the destination, or for the node that has the object that has requested. Iterative routing works as follows. The requesting node sends the request (it can also be a lookup) and when a node receives this request, it computes the node that is relatively closer to the id of the request.
While iterative requests involve more one-way network messages than recursive ones, they remain attractive because they are easy to parallelize. However, in the Virtual OSGi framework the first solution was adopted because of the fact that a lot of messages are saved, by avoiding replies from the intermediate nodes of the path to the requesting node, with information about where the object that it has requested is located. Concluding, to adopt this policy, what is needed to be added in the message, is information about the requesting node, so as the final node of the path that has the appropriate information, sends directly to that node the information and avoid routing it through the whole path involving also intermediate nodes.

### 4.1.5 stabilize

One of the important characteristics of Chord protocol that plays a very important role in the maintanance of the network is the stabilization process that a node tries to find whether one of the nodes that belong to its successor list or to the finger table. Each node has a thread which is called stabilization thread, and it is executed between time intervals which are random for each node. Furthermore, it is important to avoid the synchronization of the nodes which are trying to update their entries concurrently. By using random time intervals, with high probability, nodes are trying to run the update process in different time and

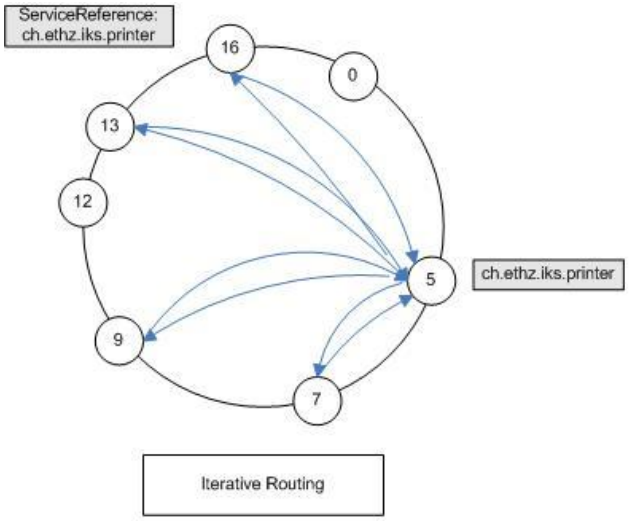



Figure 4.6: Iterative routing

as a result network congestion that might arouse from concurrent stabilization periods is avoided.

### 4.1.6 Extensions and Modification of the basic DHT interface

In order to satisfy all requirements from the OSGi specifications, modifications to the basic Chord algorithms needed to be done and more functionality has to be added to the DHT.

Because the DHT's goal was to offer a network backbone for a distributed OSGi framework,besides the normal functions lookup, insert, delete, some modified methods are needed to be added, whose logic is exactly the same with the usual DHT functions, but in some terms, are dealing with more specific objects and situation, because of the fact that the Chord that is implemented serves the functionality of the Virtual OSGi framework. For example, the objects that are stored are of type ServiceReference, Bundles, Clusters. Some of these methods return an object (Service registration for example in the case of services), some of them do not, so modified methods should be used in every case.

#### Synchronous and asynchronous calls

Some methods of the specifiacations have specific requirements and were needed to be treated analogously. First of all, usually sending messages in the Distributed Hash Table is done asynchronously. In the Figure 4.7 the operation of asynchronous calls is depicted.
Usually one node sends a message, for example a finger_table message and later on it will receive a reply from this node that should fill the specific entry of the finger table. On the other hand, a lot of methods of the specifications wait

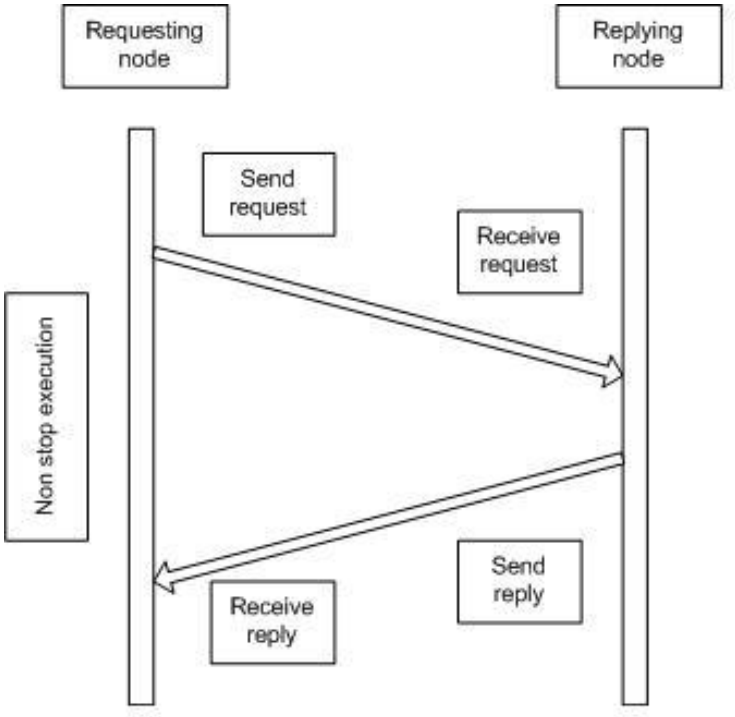



Figure 4.7: Asynchronous call

for a reply which might be a bundle object, a ServiceReference object or a property from the service. In the case that the reply is located in another node and network communication is needed, the thread that has requested for the reply until a reply will be received has to block. To achieve this synchronous call a receiving queue is used, where the reply will be received. The requesting thread blocks in that queue and waits for a notification to unblock or for a timeout to pass, in case that a reply was not received, and when the reply will be received either by the receiving thread that waits for new connections or from an already established channel (If the node is lucky and the node that has this information belongs also to its finger_table) there is a notification that something was inserted in the receiving queue and the blocked thread will try to get a result and continue its execution. Otherwise it will still wait either up to when another thread puts a reply in the queue, or when the timeout finishes and returning a null for a reply. The Figure 4.8 shows how the synchronous call generally works.

### Broadcast over Distributed HashTables

As it was mentioned above, the Chord was supposed to be used as a backbone for the Virtual OSGi framework. As a result some method that should be implemented by any OSGi framework implementation needs a special treatment. For instance, the getBundles() method of the Bundle class returns all bundles that are installed in the framework. In order to achieve this in the Virtual OSGi framework every node that participates in it has to be queried, and the bundles

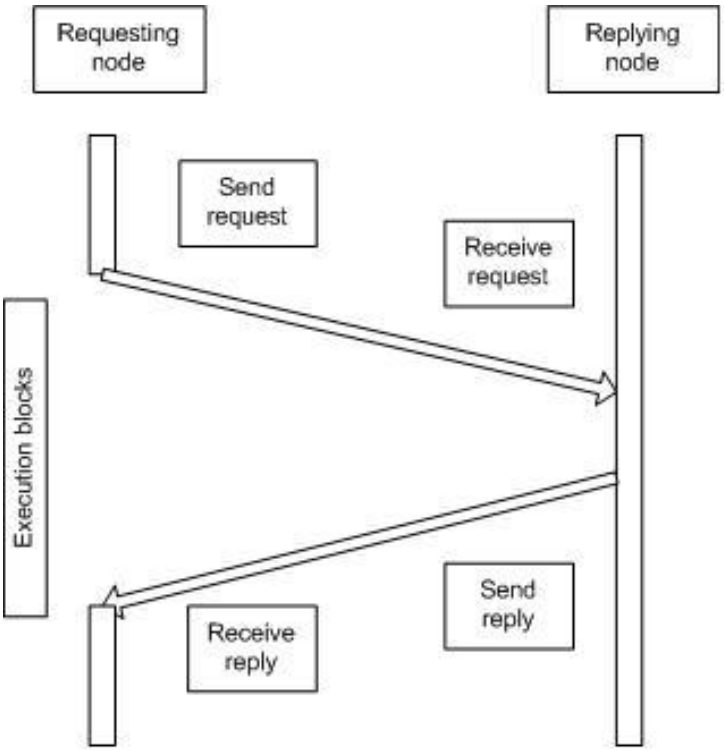



Figure 4.8: Synchronous call

that are stored to this node, and specifically in the bundle registry. In order to achieve this a broadcast should operate above the Distributed Hash Table, so as to reach every node, to achieve this with the minimum number of messages and to deal with node failure during the broadcast time. The general idea of the broadcast algorithm [36] is to exploit the DHT connection between nodes and to build above them a broadcast tree (Broadcast Partition Tree). The algorithm, gradually divides the ID space represented by its node to a d-ary BPT. The building process of the BPT tree is the following. The root node has the whole ID space. It first splits the whole ID space region into d particions with equal size and select the succeeding node of the beginning ID of each region (going clockwise among each region). This node is called the Representative node of this region (RPN). These RPNs constitute the children node set of the root node. This procedures continues up to the case where there is only one node in this region and as result the procedure finishes, having built the Broadcast Partition tree, having as root the node that has called a method which returns a values that needs a broadcast (getBundle() or getAllServiceReference()). Because of the assignment and manipulation of the IDs in the Chord network (the first node met going clockwise direction among each region is the successor node of the first id of this region), the primitive DHT lookup functions can be used for the build of the broadcast tree by finding the RPN of each region. In the process of regional division, the RPNs are pruned so as to be sure that each node will be appeared only once in the BPT. Those RPNs of spitted region constitue the child node set of the RPN of current region and those RPNs continue to split the region they represented. When there is no node or only one node in current region, the region will no longer be divided. As in the DHT region the lookup operation to get the corresponding successor list of id needs multiple hops, the latency overhead cannot be ignored. Through this steps, a d-ary broadcast partition tree has been constructed, whose height is $O(\log_d N)$. The Figure 4.9 below, shows an example of a DHT network consist of 10 nodes and the BPT that was constructed. The tree is a 2-ary BPT and its height is 5.

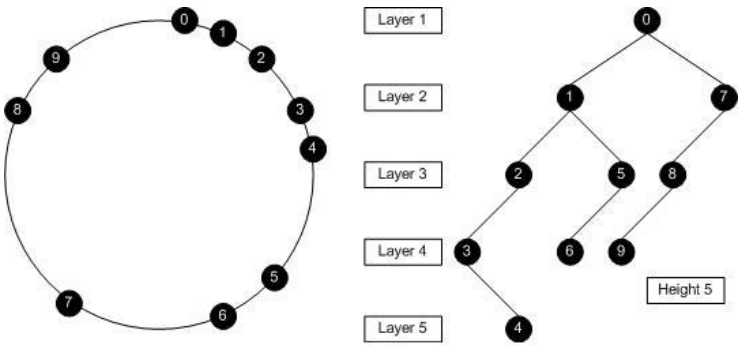



Figure 4.9: Building the Broadcast Partition Tree

In this example, the root node 0 is responsible for the whole ID space, assume that it is $[0, 16]$, this one will be divided from $[1, 8]$ to $[9, 15]$ and nodes 1, 7 are the RPNs for these regional spaces. This procedure will be continued up to when only one or not any node exists on the regional space that was created

from a space division.

This is the basic operation of the BPT for broabcast messages. According to the method of the framework that is called, there might be some modifications of the basic algorithm. For example in the getBundles() method returns the whole amount of bundles that are installed in the framework. This means that first of all, the requesting node should inform all nodes of the network that it wants the bundles that are stored in the registry of the nodes. Nevertheless, it waits for a reply. In order to propagate replies from every node in the network. A policy in that case would be that every node will know which is the requesting node and send its bundles to it. Unfortunately, this policy might create a bottleneck in the requesting node, because it will receive messages from every node in the framework and the number of these nodes might be very large. Another option, which was finally selected, is that each node will propagate its data to its father, the RPN of the regional space that it was divided into itss regional space and to other ones. As a result, each father waits for d replies from each of each d children, and when it receives all of them it sends to its own father everything that it has collected plus its own data. Finally, the root node will receive only d messages from its own children which will contain the whole amount of bundles of the system. For instance, in the virtual framework case that each regional space is divided in 2 subspaces, and each node will send two lookup messages in the network so as to find the RPNs of the newly created spaces. It will also receive only two messages from these two nodes.

Other operations need simple broadcast operations. In Chapter 3 there is a description of how listeners are registered and inform when a new event arrives. When a bundle from a node registers a listener in the framework it should notify all nodes in the network and it does not have to wait for a reply. The general way that listeners are registered and informed will be presented thoroughly later on in the next chapter.

The BPT was another reason that Chord was selected for the development of the Virtual OSGi framework. The Chord ring that the ids of the nodes are organized was very helpful for the use of the broadcast algorithm, because the division of the ID space was easier for the Chord case. In the example of the method getBundles() these two extensions (synchronous calls, broadcast) are combined in one method call. Both of them were crucial for the implementation of the Virtual OSGi framework and their adoption and implementation was very helpful.

## 4.2 Virtual OSGi framework

The Virtual OSGi framework is a bundle that is installed in an existing local framework. After installation, it is responsible for either creating a new Chord ring or joining an existing one. After the initialization of the appropriate data structures (finger table, successor, predecessor, successor list, registries) it starts initializing the framework. It searches, whether a configuration file exists, so as to install or to start other bundles from the beginning of the framework, or to retrieve the properties of the system. These bundles will be installed of

course to the local framework but the bundle object that will be returned is the VirtualBundle object.

The Virtual Framework is responsible for assigning the ids to the bundles that are installed and to the services that are registered in this node, which are different from the ids that have in the local framework. These ids must be unique for the Virtual framework according to the OSGi specifications. In order to achieve uniqueness in the ids the port of each node is hashed and the number that is produced every time a new bundle is installed is increased by one. It maintains data structures for retrieving the local services or the bundles with their interface or with their ids respectively. In addition to these data structures, it maintains data structures for the local listeners, so as to inform them when it receives an update event, and finally it is responsible for maintaining the consistency of the ids. The main class of the framework is the *VirtualFramework* where the initialization processing is done. Other important classes of the Virtual OSGi framework are *VirtualBundleImpl* and *VirtualBundleRemoteImpl*. These two classes are responsible for implementing the bundle objects in the framework. Both object implement the Bundle interface but the implementation of the methods is different. This means that when another bundle requests for another bundle it searches with the id of the bundle so as to get it. In order to retrieve a bundle that it is located in another node, a lookup bundle message is propagated to the network, up to when it reaches the responsible node for the bundle information. This node will send back the appropriate information for the bundle. For instance where it is located. After obtaining these information, it can build a proxy bundle which will communicate with the real one and it will obtain through the network the appropriate information. Concerning the context of the bundle object the classes that implement it are *VirtualBundleContextImpl* and *VirtualContextRemoteImpl*. To conclude the basic classes of the Virtual OSGi framework are the following:

```
public final class VirtualFramework
public class VirtualBundleImpl implements Bundle
public class VirtualBundleRemoteImpl implements Bundle
public class VirtualBundleContextImpl implements BundleContext
public class VirtualContextRemoteImpl implements BundleContext
public class VirtualClassLoader extends ClassLoader
public class VirtualPackage
public class VirtualServiceReferenceImpl implements ServiceReference
public class VirtualServiceReferenceRemoteImpl implements ServiceReference
public class VirtualServiceRegistrationImpl implements ServiceRegistration
public class Version
public class ServiceInf
```

Figure 4.10: Representative Classes of the Virtual OSGi framework

## 4.3 Virtual Bundles

Bundles that are installed in the framework are of two types concerning how an instance of the Virtual OSGi framework deals with them. Local bundles that are installed in the node that the specific instance operates and remote

bundles that are installed in other nodes. These two types of bundles are treated differently and their objects are the *VirtualBundleImpl* for the local bundles and the *VirtualBundleRemoteImpl* for the remote ones. In the Figure 4.11 is presented how the Virtual OSGi framework deals with the local and the remote bundles. If the bundle is local, then apart from the processing that it does it contacts and the local OSGi framework to process it. On the other hand, if the bundle is remote, which means that it is located in another node, it does not need to contact the local framework, because it does not have any information about this bundle.

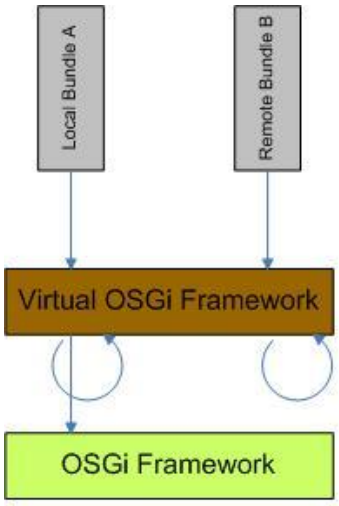



Figure 4.11: Discrimination of the two different bundle objects

The local bundle class is a wrapper of the bundle class that is implemented in the local OSGi framework on each node. The new classes were added, because of the fact an interception of the methods of the bundle had to be implemented, so as to add selection about which framework the local or the Virtual one will execute this operation. The Virtual OSGi framework is not based on implementation characteristics of the host framework and as a result it can be installed in every OSGi framework implementation. Up to now, it operates above eclipse equinox. The *VirtualBundleImpl* maintains the bundle object and the bundlecontext of the local framework, which is used for local operations in some methods of the new bundle class. Other data that this object stores is the *Node* class of the framework which is used for network communication such as registering the bundle or updating the state of it in the network. Furthermore, it uses the *Node* class so as to get the clusters for its resolving process. Information that is retrieved from the Manifest file of the bundle is stored in the *virtual bundle* object which are needed for the resolving of the bundle and for the creation of the Clusters. These information have to do with the strong dependencies between the bundles of the system. A basic difference in the *VirtualBundleImpl* class is that the start of a bundle is implemented with the help of another method of the local bundle object. The *VirtualBundleImpl* object stores information about the activator of this bundle. After doing the appropriate checking in the state of the

bundle, it tries to use the loaclass() method of the local bundle class so as to get the activator class and calls reflectively the start method of the bundle activator class. This is done because the Virtual OSGi framework does not deal with any class loading, at least in the case of the local bundle objects. Moreover, the Virtual OSGi framework does not deal with the module layer except from the resolving process were it was inevitable, because deciding whether a bundle can be resolved, demands communication with other frameworks, and consequently the Virtual OSGi framework should be involved in that process. Methods that have to do with the retrieve of an entry or information about the manifest file, and general do not have to do with network communication are wrappers of the methods of the local framework.

The *VirtualBundleRemoteImpl* is connected with the bundle remote object. The Virtual OSGi framework is a global framework which is consisted of connected OSGi frameworks. The whole system is presented as one entity, so in every node, every bundle that is installed in every other node of the framework must be accessible. When a bundle is registered, critical information about were it is located, are hashed and stored in the network so as to be accessible by everyone in the network. When one node requests a bundle which is stored in a different node it does a lookup to gain this information and when it gets it, it creates the *VirtualBundleRemoteImpl* which is the remote object for the bundle. The basic objects that has this class, is the *Node* class which is responsible for communication, by establishing connections and sending messages to other nodes and the *BundleRemote* object that has information about the location of the bundle. Furthermore, a new classloader is created which is used for loading classes of the bundle in the other node. In the remote bundle case, the classes are implemented differently from the local bundle objects. In this case, this object acts as a proxy to the real bundle and wraps the local calls to methods of the bundle, to remote calls to the peer that stores the real bundle object. Furthermore, some methods can be satisfied by the data the bundle remote object stores, although the intention was to be kept it as simple and small as possible. This object apart from information about the node that the bundle is located it has other information for the bundle such as the headers of the bundle and the final updated state of it. In the remote bundle case, when the user requests a bundle to start, it builds a start message and sends it to the node that has the bundle. This node tries to start the bundle, according to its state. It checks, for example, it is resolved, so as to continue the procedure. For the remote bundle object a new classloader *VirtualClassLoader* is added because of the fact that the method *loadClass()* of the OSGi specifications is implemented. In this case, the remote bundle object contacts its classloader which sends a message to the node that has this bundle. When it receives this message, the node gets from the framework the bundle that it wants to load the class and it calls the getEntry() method of the local framework so as to receive a URL for the class. After that, it will use the stream to fill a byte array with the data of the class and it will send them back to the classloader of the remote bundle object. When it receives the reply message, it calls the defineClass method of the java classloader to convert the bytearry to an instance of a class Class. Finally it will return this class to the application that used this bundle to get a class. In the Figure 4.12 is presented how the communication through

the new classloader is acieved.

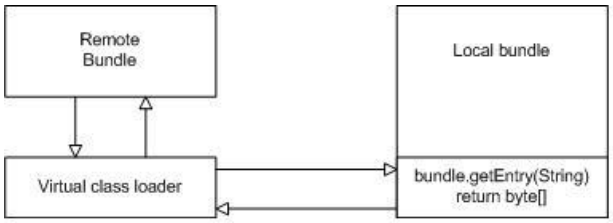



Figure 4.12: The ClassLoader of the Remote Bundle object

In OSGi, every bundle has its own bundleContext which is the mean of communication of the bundle with the framework. In the Virtual OSGi framework two BundleContext classes are implemented, one for local objects and one for remote ones, whose target is to treat different the specific request of an application and in the user this process will be trasparent. The *VirtualBundleContextImpl* has information about the virtual bundle that it is connected, and a pointer to the BundleContext of the local OSGi framework. The BundleContext is used when a bundle wants to register a Listener, register service, get a ServiceReference or get a Service object. *VirtualBundleContextImpl* acts as a wrapper for the ServiceReference object of the respective local framework and it adds logic that has to do with the network communication of the Virtual OSGi framework. For instance, where the RemoteServiceReference object will be stored or the informing of the other nodes of the system that a new listener was registered in this node. Furthermore, this object is used from bundles to obtain another bundle, or all the bundles of the system. The first method, for instance, if the bundle is located in this node it does not need any network communication, only a retrieval from the local data structures. The second one needs a broadcast to the whole network so as to visit every node and obtain the bundle information that it stores. Concerning services, the *VirtualBundleContextImpl* object does the appropriate processing to the responsive request. If the request is, for instance, a service registration it is responsible for creating the object that will be hashed under the interface of the service or the interfaces if the service has more than one.

Apart from the *VirtualBundleContextImpl* the Virtual framework offers a BundleContext for the remote bundle objects. *VirtualContextRemoteImpl* is the class that implements it and it is the equivalent of BundleContext interface as *VirtualBundleRemoteImpl* is for Bundle interface. It is also a proxy for the real BundleContext object that is located in the node that the bundle is located. In most cases, what it does is that it maps local calls to remote ones to the VirtualBundleContextImpl object. Nevertheless, listeners are treated in a different way. When an application uses the remote context to register a listener, this listener is registered in this node and not in the host node of the bundle. What is done after that, is that the VirtualBundle is informed that in a specific node there is a Listener installed under its remote context object and when it is uninstalled it should remove this Listener from the framework. Some

methods are implemented in the same way as in the *VirtualBundleContextImpl* class. For instance, the getBundle(id) or getBundles(), are implemented in the same way in both cases. And the services are implemented in the same way in the remote and in the local case. On the other hand, some methods are not implemented at all, because it was irrational to have them in a remote object. the getDataFile() method which returns a File object for a file in the persistent storage area provided for the bundle by the framework. In the remote case, returning a file object for bundle that is stored in another node does not make sence, because of the fact that this files are in the persistent storage of the bundle that is located in another node and this one does not have access to that area.

## 4.4 Resolving Process

An important procedure in the OSGi framework is the resolving of a bundle so as to be able to start. Constraint solving is the case where imports and exports of bundles are matched. In order a bundle to be resolved, other bundles have to offer packages that satisfy the imports of this bundle. Packages that are imported and exported, should match and in the Version. Version constraints are a mechanism whereby an import definition can declare a precise version or a version range for matching an export definition. In the Figure 4.13, bundle A imports package p, and bundles B,C export it, with different versions, and one of them does not match the version constraints of the import and the resolving process will result to selecting bundle B to build wires.

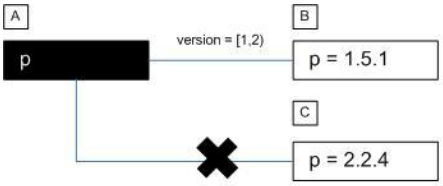



Figure 4.13: Version Matching example

The Virtual OSGi framework has the *VirtualPackage* and *Version* classes which deal with these cases. The first class is used for storing useful information for the constraints of a bundle, either this constraint is an import, export or a require-bundle. It stores its name, its directives and its version, either it is a range version or not. In addition to this, it offers a static method that takes as an input a string object and builds a package storing the useful information. *Version* class is building the version objects of the packages. Moreover, it offers a static method which compares the versions of two packages, and sees whether they match. Solving the constraints of a bundle in an R4.1 OSGi framework demands a different procedure than the previous versions of the specifications. Now, bundles that satisfy constraints of other bundles, are wired with them implying constraints on a number of other packages that the exporter depends on. In Virtual OSGi framework instead of the wiring procedure that was very costful *Clusters* were introduced, which are equivalent to the wires. The advantage of clusters is that only one node is accessed to obtain information about bundles

that are connected together, and there is no need for traversing through different nodes every a time a bundle tries to be resolved. In the clusters case when a bundle tries to be resolved, it contacts the nodes that they have a cluster stored in their registry and tries, locally, to find the appropriate combination of clusters to merge, or just one cluster that satisfied its constraints. The node sends to the cluster nodes information about what are the imports of the bundle. Thes nodes check the message and compare the constraints with the cluster information, if it satisfies some of them it sends the cluster information, otherwise it sends an empty object meaning that this cluster cannot help in the resolving procedure of the specific bundle. When this step finishes, the bundle has collected the candidate clusters that might be helpful for the resolving process. After collecting all possible candidate clusters it is able to find a combination of them that satisfies all its constraints. Having collected all possible clusters that might satisfy its constraints, a data structure built which for every constraint that the bundle has. It stores the whole number of clusters that satisfy it. Then, it tries to find whether one cluster can satisfy all its constraints so as to avoid network communication with a lot of nodes that store the clusters for the updates. If no such cluster exists, then it tries to find a combination of clusters that satisfy all bundle's constraints and they do not have conflicts. If such a combination does not exist then the bundle cannot be resolved and its state does not change to RESOLVED. On the other hand, if a possible combination was found then it should update the clusters of the system.

### 4.4.1 Locking of clusters

In order to maintain consistency, the mutual exclusion between the nodes that try to modify the clusters is implemented. In the case that two different bundles are in the resolve process, they are not allowed to update the clusters concurrently, because, for instance, one will try to delete one and the other one will possibly try to update it. In order to avoid this case, each cluster maintains a lock, which the bundle, that wants to do the update, has to obtain before modifying any cluster. Furthermore, the bundle must obtain every lock of the clusters before udpating them because otherwise, a deadlock might happen. Consequently, two phase locking 2PL [29] is adopted. This condition is used in the database transactions, to guarantee that a schedule of consistent transactions is conflict serializable. The 2PL is the following.

- In every transaction, all lock requests precede all unlock requests.

The two phases referred to by 2PL are thus the first phase, where locks are obtained and the second phase, where locks are relinquished. Two phase locking is a condition on the order of actions in process. Two phase locking works in the database case, but it is modified it so as a possible deadlock is prevented. In the Virtual OSGi framework, when a bundle has selected the cluster that will satisfy its constraints, try to udpate them. The update is consisted of three phases. Acquire Locks, Update Clusters and finally Release Locks.

The first phase is the aqcuiring of the locks. The requesting node contacts all nodes that store the clusters that the resolving bundle wants to use and try to acquire the lock of every cluster. If it acquires all of them, it is ready to start the cluster update. If it cannot acquire all locks, then it immediately releases

all cluster locks that it has obtained and waits for a random time interval before retrying to be resolved, from the beginning. The retry is happening, because of the fact that modifications in the clusters might resulted to the case that a selected cluster has a conflict with the exported packages of the bundle. The cluster procedure in this case will start from the beginning. In the case that the bundle acquires all locks, it selects one cluster that will be updated. The other ones that will be merged with the first one, should be deleted from the nodes of the framework. The requesting node sends them a delete message which has information for the cluster that will be merged with them so as to send them their data and with its exporting packages, so as to check whether there was a change in the clusters, which created a conflict with the packages of a newly resolved bundle. The update message are sent by the nodes that have the clusters that will be deleted. The requesting node only needs to send the delete and the update message. The update message has also potential exports of this bundle which should be added to the cluster and will imply constraints for other bundles that will try to resolve at a later stage. After that, the other nodes will continue the update procedure, by sending their data to the remaining cluster. When the final cluster receives new data from all clusters that participate on the update process. This cluster knows how many update messages will receive by receiving in the update message a number that is equal with the number of the clusters that will be deleted. After receiving all updates, it will release its lock, so as other bundles that needed to resolve and use this cluster will continue their processing.

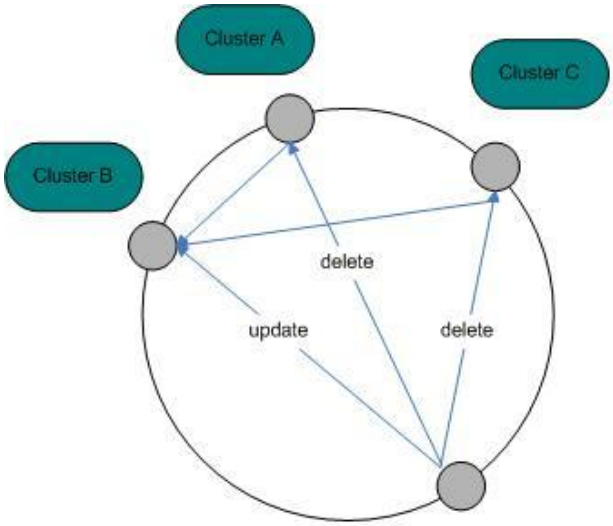



Figure 4.14: Resolving Example, obtaining the cluster information

After the update case, the deleted clusters inform the other nodes that they deleted their clusters, and they send a broadcast message to all nodes of the system so as to be informed that this cluster does not exist in the framework anymore. In case that another node tries to obtain information about this cluster before receiving the delete broadcast message, it receives an empty message for

a reply without knowing whether this cluster does not exist or does not satisfy any of its constraints, so the only thing that is lost in this case is the propagation of two messages. Concluding, the important decision in the locking phase was to lock the clusters only when they are needed to be updated, so as not to exclude them for a long time, and even when they are not used by any bundle. If Clusters were locked when a bundle tries to resolve, then other bundles would have to wait for the release of all clusters even though this is not needed by the bundle that is in the resolve process. With this strategy, bundles might doing their resolve process, based on clusters that they are already changed, but when they try to do their update they realize that a change happened and they have to redo the resolve process. What is achieved with this strategy is that the number of the clusters that they are locked and the time that they remain locked is limited. The tradeoff for this strategy is that bundles might try to be resolveg using clusters that might not even exist, but the possibility for this is not high because of the fact that this cluster should change only if it was used by another bundle, otherwise it will not be locked and thus it is accessible to a bundle that needs it.

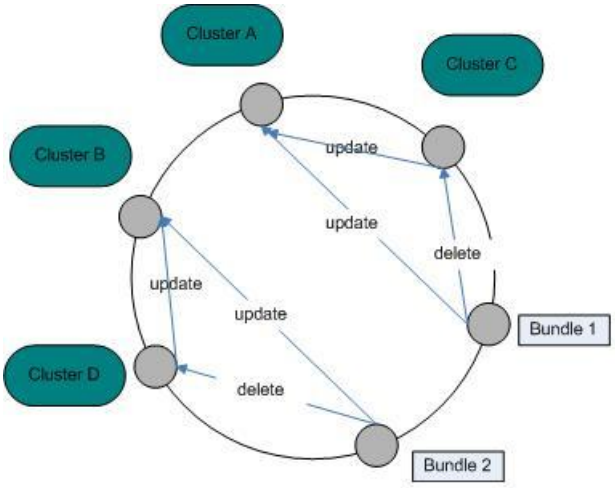



Figure 4.15: Example of two concurrent resolving process

In the Figure 4.15, an example is presented where two different bundles are trying to be resolved. Finally they select two different clusters each. With this policy, they will do the update concurrently because the two using cluster sets do not intersect and as a result the one process will not affect the other one. If a decisionc for locking all clusters when a bundle requests them, these two resolve should be executed sequentially, without any specific reason, as far as their clusters do not intersect.

The final step for the resolving process of the bundle is to obtain the bundles that are needed in this node so as to be able to start, as far they have the appropriate classes installed in this node. After the requesting node sending the update or the delete message, it waits to gather information about where the

bundles that have the requesting package are located. This node will establish a channel with these nodes that have the bundles, and finally they will receive from them the appropriate data. The appropriate bundles will be installed on the requesting node. The bundle is resolved, its state is marked as resolved and it can be started. In the Figure fig:clusters2 is presented the whole resolving process in the Virtual OSGi framework, where, after the communication with the cluster, that this node contacts the other node to retrieve the bundle that its export is needed for the requesting bundle.

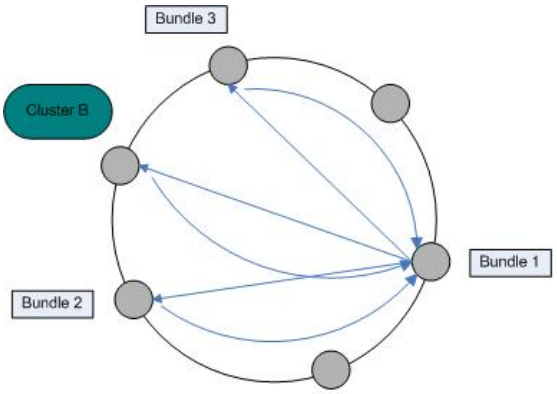



Figure 4.16: Resolving process

## 4.5 Virtual Services

Service is an object registerd under one or more interfaces together with properties. The Virtual OSGi framework has a refined way to deal with them. The service object is connected with the bundle that owns it. This bundle registers the service to the service registry so as to make it available to other bundles of the framework. The dependencies between the bundle owning the service and the bundles using it are managed by the framework. When the bundle is stopped, for instance, all the services register with the Framework by that bundle must be unregistered. The Virtual OSGi framework deal with the services that exist in the system. *VirtualServiceReferenceImpl* and *VirtualServiceReferenceRemoteImpl* are the classes that implement the *ServiceReference* interface of the OSGi specifications. The differences in these two classes is that the second one has to do with accesssing the service object and its properties remotely.

### 4.5.1 Registering a Service

When a bundle registers a service, it calls the respective register method of the Virtual BundleContext. This method returns the ServiceRegistration object

which is implemented by the *VirtualServiceRegistrationImpl* class of the Virtual OSGi framework. After registration of the service then it can be retrieved by other bundles of the framework. As it was described in a previous chapter, the Service Registry of the Virtual OSGi framework is completely distributed based on the distributed hash table which is the network backbone of the system. Services are hashed with their interfaces, and if they are more that one, they are hashed in different nodes of the system. One for every interface that the service implements. In the node that the bundle that registered the service object is located, it maintains the ServiceRegistration object which will be used to unregister the service from the framework. To unregister a service the unregister message to the nodes that store the service is sent. By knowing the interfaces that the service implements, with hashing, the service ids can be retrieved and the messages can be propagated to the network. When a bundle in the same node tries to obtain a service reference object from a service which has been registered by a bundle which is located in the same node then the whole network communication is avoided by first quering the local registry and if it does not get a result then it will search the whole network for obtaining a service reference object for the interface that it requests. This strategy is selected, for avoiding sending messages to the network, in case where the request can be satisfied locally. According to whether the service is stored in the local node or in another one, it gets as a return value from the respective method a local or a remote ServiceReference object. In case that more than one services exist that implement this interface and all of them are stored in the node, the one that arrived last will be selected. The arrival time of a servicereferenceremote object is kept as a criterion for selection, so as to offer the latest service addition to the system. Furthermore, a bundle is possible to request a service reference also by sending a string filter which will be applied to the properties of the service. When a node requests a servicereference with a filter, it will send a message to the network, and when this arrives at the responsible node, which stores the servicereferenceremote object, it will try to check whether the filter matches to the properties of the service object. This is achievable because of the fact that the remote objects maintain the properties of the service. For the matching process the local framework will be used to create a filter from the string by calling the createFilter() method and will match it with the properties of the service to check whether it is satisfied or not. If there is an object that satisfies them, then it will be returned to the requesting bundle.

When the bundle that has registered the service in the framework updates a property of the service, then an update message is propagated to the network so as to inform the serviceReferenceRemote object, which is stored in the distributed service Registry, that it has a new value in one of its properties. Furthermore, when a bundle is unregistering a service, it is removed by the Service registry, by sending an unregister message, and it notifies the listeners by sending a ServiceEvent.UNREGISTERING.

### 4.5.2 Obtaining the service object

After collecting the ServiceReference object, a bundle can get the real service object. If the bundle is "lucky" and the service is offered by another bundle located in this node the local OSGi Framework will be used to get the service

object. Otherwise R-OSGi [33] is the solution to get a service object. The whole process for retrieving a service in the Virtual OSGi framework is presented in the Figure 4.17.

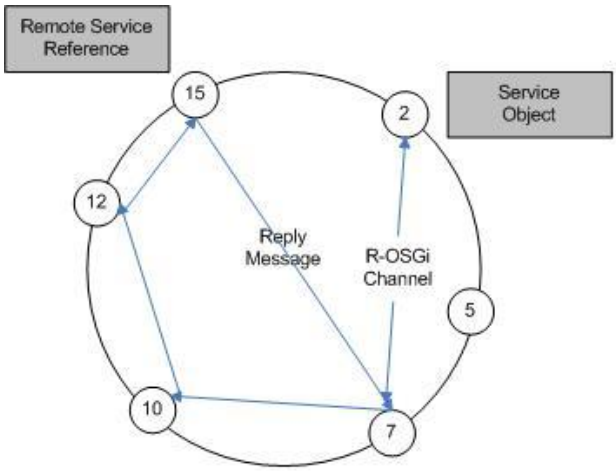



Figure 4.17: Retrieving a Service

In order for a bundle to get remote services, R-OSGi is ported to the Virtual OSGi framework, and especially the functionality that it offers for fetching the service and invoking its methods. When a bundle wants to get a service from a remote bundle it estabishes a Channel with this node by creating a ChannelEndpointImpl. The communication for services is completely decoupled from the other network communication, The channel registry of the node is not queried, and even if it has establised a connection with the other node, it creates a new channel for connumication with the service. After the channel is established, the two peers communicate through it for the service method invokations. The information for the remote node are obtained by the remoteservicereference object that is stored on the network. This will be used for the creation of the *VirtualServiceReferenceRemoteImpl*. This maintains information for the remote peer and the service that is registered from this node. Having these information a proxy object of the service can be created. This procedure is done when a fetchService message is sent to the other peer, which will send back a reply message with information about how to build the proxy of the service. Proxy generation is implemented using bytecode manipulation.
In order to call a method from the remote service, the R-OSGi sends a message that transfers the description of the called method together with the arguments of the call to the remote peer. On the service provider side, the real service method is called. A response message is generated and sent back to the calling peer. R-OSGi creates proxies for remote services.These proxies are bundles and behave as a local service. These proxies, send the appropriate messages the node that has the service object. An important characteristic of R-OSGi is that no code needed to be transferred and no skeletons or stubs are needed also. Furthermore, it has to be assured that the generated proxy is resolvable and

this is achieved through type injection. If the type that occurrs in the service interface, is contained in the service bundle and the package is declared to be exported by the service bundle, the corresponding class is added to the injection list. Whenever a client fetches the service, the injections are transmitted. During proxy generation, the injections are stored in the proxy bundle. The packages of all referenced classes not included in the injections are declared as imports of the proxy bundle. The packages of all injected classes are declared as exports to ensure type consistency within the framework.

On the client side, a proxy bundle is created which implements the interface of the service and the Bundle interface also. The methods that the proxy bundle implements, wrap the local method calls to calls to the R-OSGi network channel, which is different from the channels that are used for communication between nodes in the network. On the other side of the channel, the first step taken is to lookup the corresponding service. If the remote method call succeeds, the result value is sent back.

## 4.6 Serialization

A problem that needed to be solved in the Virtual OSGi framework, was the case that a node of the system has to send an object that do not implement the serializable interface. For instance, by trying to test Jonas so as to work in cooperation with the Virtual OSGi framework apache ipojo bundle was used, which registers a service that has a property that its value is not a standard Java object. Even though this object has fields that are primitive data types and standard serializable Java objects, this object could not be sent through the standard Java serialization mechanism. In the Virtual OSGi framework, the messages contain objects that are serializable and as simple as possible. Complex objects are deconstructed in their simple components (primitive types), and when the responsible node receives this message, it uses the components to reform the initial object. In order to offer the possibility to bundles, to register services with non standard Java Objects as properties, an extension of the serialization mechanism of the Java is implemented. With the help of jvmti [3] and Java jni [2] a new serialize and deserialize methods are introduced. The JVMTI Tool Interface (JVM TI) is a new native programming interface for use by tools. It provides both a way to inspect the state and to control the execution of applications running in the Java virtual machine (JVM). JVM TI supports the full breadth of tools that need access to JVM state, including but not limited to: profiling, debugging, monitoring, thread analysis tools. The Java Native Interface (JNI) is a programming framework that allows Java code running in the Java virtual machine (JVM) to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages, such as C, C++ and assembly. In the Figure 4.18 is presented how JNI bridges java and native applications.
The serialization that is implemented uses the jvmti methods to obtain information about the class of the object that is needed to be sent. In the beginning the fields of the class are searched in order to find out whether all of them are either primitives types or they implement serializable. It is possible that an object might implement the serializable interface and one of its fields does not,

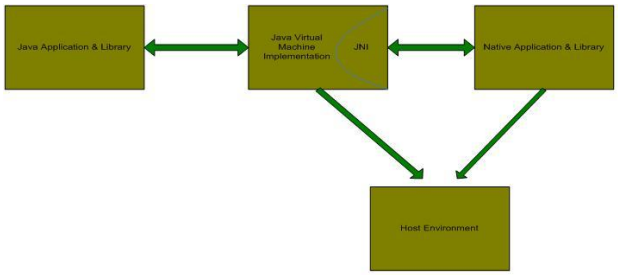



Figure 4.18: JNI

even though this might not be a common case. If all fields satisfy these requirements then the standard java writeObject() from the ObjectOutputStream class is called to write the object. On the other hand, if a field of the sending object does not implement the serializable interface then our serialization sends the object. After that, a search for static fields is executed so as not to send them. After that each non static field of the class is checked. If it is a primitive type it is written normally, but if it is a custom object the serialization method is called recursively to do the same execution to its fields. The same procedure is done in the deserialization of the object in the receiving node, where the signature of the object is read, and after that the signature of the fields, and according to the content of the signature, the appropriate java methods are called.

With this implementation of the serialization, bundles, can register services with properties of values that are objects of custom classes, and generally non serializable objects can be sent through the network.

# Chapter 5

# Evaluation

The Virtual OSGi was tested with the Event Admin Service [18] to test the overhead that it adds to a local OSGi framework and to test its scalability in the case that nodes that are added to the network in the case that one node calls a service remotely from another node.

The Event Admin service provides communication to bundles, which is based on the publish subscribe model. In the publish subscribe model, a publisher publishes messages which are characterized into topics, without any knowledge about the subscribers which have registered themselves in the system. Subscribers express interest in one or more topics, and only receive messages that are of interest, without any knowledge for the publishers. In this model Event is the object that is used for communication. Each Event has a topic, which is the name of the Event type and EventProperties. Events are published under a topic, and with a number of properties. Event Handlers can specify a filter to control the Events they receive. The participants in this publish subscribe model are the Event Admin, which is the provider of the publish subscribe model, the EventHandler which is the receiver of the events, the Event Publisher, which is a bundle that sends events through the Event Admin Service.

In the first experiment, an EventAdmin bundle is installed in one node, an Event Publisher is installed in another node and an EventHandler is installed in another one.
The Event Publisher keeps posting events to the EventAdmin service and this one propagates them to the EventHandler. This situation is tested by adding nodes to the framework so as to see whether the Virtual OSGi scales well in the service remote calls when more nodes join the system. First of all the overhead that the Virtual OSGi adds to the local framework was tested, in the case of only one node in the system. Running on a local framework the number of events that the EventHandler received was 150893, while with the Virtual OSGi framework running above the localframework the number of events was 148403 which is 1.65% smaller than the amount of events that the EventHandler receives in the plain local framework case. In the case of adding nodes to the system, it is expected that the call of services will not be affected because when the connection with the remote service is established, other nodes that participate in the network do not generally affect the communication between the bundles.

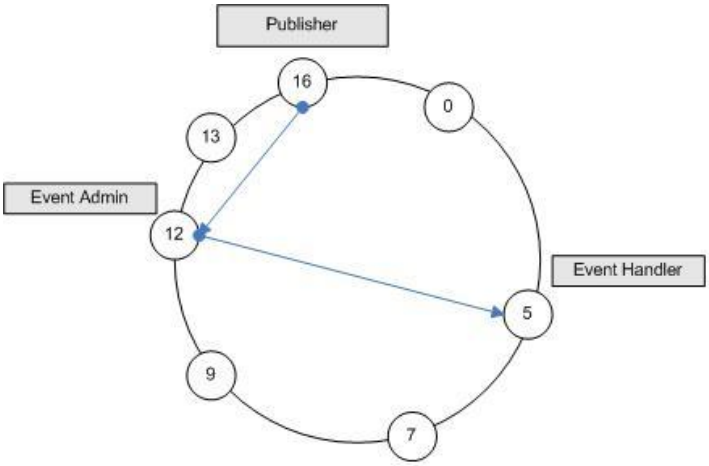



Figure 5.1: One EventHandler in the system

The only cost that it is higher when more nodes are located in the network is the registration of the Service, because the Event Handler spends more time for inserting the service in the network, but for this case, the diferrence in the cost is not increased. In the Figure 5.2 is presented the amount of events that an EventHandler receives in a period of one minute, while the number of nodes in the system is increased. The number of event that the EventHandler receives, does not have big variations, and it is not highly affected by more nodes inserting the framework.

| | 3 Nodes | 4 Nodes | 5 Nodes | 7Nodes | 9 Nodes |
|---|---|---|---|---|---|
| EventHandler1 | 29109 | 31060 | 27300 | 27656 | 27168 |

Figure 5.2: Number of Events per minute

In the second experiment the same situation is tested, when two EventHandlers are located in different nodes and they have to be informed when an Event is published to EventAdmin service.
In this case, two EventHandlers from different nodes register themselves as services and the Event Admin bundle, locates them and stores them so as to send them event that interest them. In the case of the plain OSGi framework the whole amount of event that was delivered to the two EventHandlers in one minute was 197360, 98680 per EventHandler. The number of events that are delivered to the two EventHandlers with the Virtual OSGi framework is 193616, which means that there is a reduction of 1.87%. In both cases the small overhead that the Virtual OSGi adds, is because of the fact that actually it does not know that this node is the only one in the system, and consequently it does some checking in order to send information through the network. In this example the Publisher continiously posts events also to the Event Admin,

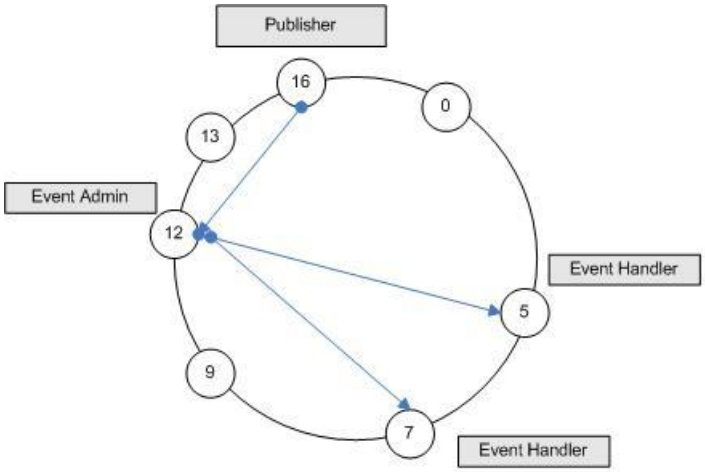



Figure 5.3: Two EventHandlers in the system

after obtaining it as a service, and the Service Admin posts the events to the Handlers that are installed on other nodes. In the Figure 5.4 the number of events that each node receives is presented. And in this case the number of events that the two EventHandlers receive is not decreasing highly in the case that more nodes participate in the network. The conclusion that can be drawn is that the number of event that are handled by an EventHandler is not affected from the increasing number of nodes

| | 3 Nodes | 4 Nodes | 5 Nodes | 7Nodes | 9 Nodes |
|---|---|---|---|---|---|
| EventHandler1 | 18245 | 17487 | 17932 | 19942 | 17627 |
| Eventhandler2 | 19221 | 18250 | 19368 | 18642 | 16845 |
| All Delivered Events | 37466 | 35737 | 37300 | 38584 | 34472 |

Figure 5.4: Number of Events per minute

From the experiments, the scaling of the Virtual OSGi framework is presented, while more nodes participate in the system, in terms of remote Service invocation. The cost for the EventHandlers to be informed for an Event does not have variations when more nodes join the network, because of the fact that it exploits the point to point connections with the service provider. Furthermore, the Virtual OSGi framework add a small overhead (1.65%-1.87%) to a local framework where only one node exists in the system, because of the fact that it propagates all operation to the local framework. The small overhead exists, because of the fact that the one node does not know that it is the only node in the network and it has to do a checking before propagating the operations to the local framework.

# Chapter 6

# Conclusions

The Virtual OSGi framework is a completely distributed OSGi framework implementation. Based on a structured peer-to-peer network (Chord DHT) it connects transparently different OSGi frameworks in a collaborative way. Different peers, that run OSGi frameworks, by installing the Virtual OSGi bundle can participate in the system. Each node can use bundles and services that are located in every node of the system, even though it does not have any prior information about the node that the bundle or the service is located. A node, by installing bundles or registering services to the Virtual OSGi framework makes them accessible by all other nodes. Registries for these objects are completely distributed in the system and they use the algorithms and logic that the DHT offers. By exploiting the DHT funcionality the Virtual OSGi framework, every node does not need to maintain information for every node that participates in the system. In the Virtual OSGi framework each node maintains information only for a few nodes, and based on them it can send messages to all other nodes in the network. Furthermore, the Virtual framework deals with fault tolerance by maintaining copies of the stored values in the successor list that each node maintains.

The important characteristic of the Virtual OSGi framework, is that it gives the user the impression that he is using one framework (OSGi on the cloud). By having acces to every bundle and service of the framework. The challenge here, was how the Virtual framework will be dealt with the remote objects and do not violate the OSGi specifications. Remote objects are treated in a different way, involving network communication, in order to, firstly, gather information about where the object is located and then to retrieve it or communicate with it. Even though two classes are implemented for every object, one for local and one for remote ones, the user or the application is not able to discriminate whether the service of the bundle that it uses is stored on the node that it operates.

The Virtual OSGi framework, deals with the new resolving process of the bundles, which was introduced in the R4 version of the OSGi specifications. The resolve entities are also distributed to the framework in many nodes. In order to present a functional system, a lot of decisions had to be made such as how the clusters will be stored, how other nodes will be informed, how the update will happen, how consistency will be maintained. The fact that the

Virtual OSGi framework is a distributed system, means that every decision should be made very carefully, and every consequence of it is taken into serious conisderation. The decisions and the reasons that were taken, are presented in the respective chapter.

Concerning services, they are registered in the distributed registry which operates atop the DHT. They can be retrieved by every node of the framework, by following certain steps. Firstly, a lookup operation is done to retrieve the servicereference object. After that, in order to get a service the R-OSGi logic is followed.

The Virtual OSGi framework is the first step towards modifying applications and making them more efficient without changing the code of the application but by just redistributing bundles and services. The goal will be to parallelize as much as possible service invocations, which will be held by different nodes. This will result to invoking different service methods in different nodes concurrently without any interfererence in the execution of them, because of the fact that two nodes do not communicate and as a result they are completely independent.

The involvement with the implementation of the Virtual OSGi framework was very interesting and very fruitful. It offered the oppurtinity to the developer to be involved in an implementation of a distributed hash table, which is a rapidly evolving technology and is used in many commercial applications such peer-to-peer file sharing applications (emule, bittorrent). Apart from this, dealing with OSGi is a very interesting process, because OSGi is a technology that is dealing with the modularization of Java applications, making them more easily reusable. It is a highly evolving standard that is used by more and more companies such as IBM, BMW and NEC corporation. Furthermore, through this thesis, the opportunity of a thorough understanding of Java operates with network communication and serialization of objects is achieved. Finally, as far as I am concerned it was very helpful, participating in an ongoing project of the systems group at ETHZ Zurich, and cooperating with intelligent people, who have very interesting ideas and were always willing to help me with problems that arouse during this period of six months. A lot of experience was obtained during this period in implementing a whole system from the beginning, having to take design and implementation decisions, and having to think the consequences of it. Even though six months seemed a very long period, the fact that

## 6.1 Future Work

Improvements that can be done in the Virtual OSGi framework are the following. First of all, the installation of bundles can be treated differently and in a more dynamic way. For instance, in the case that a lot of bundles are installed in one node and other ones do not have a lot of bundles and are idle, while the first one should satisfy a lot of requests, the framework could distribute the bundles more uniformly, so as the load of the system will be distributed also to more nodes, avoiding the creation of a bottleneck.

Furthermore, in the case that a node sends a lot of requests for a specific service or a specific bundle, a policy can be implemented that this service or bundle will be installed to this node, so as to avoid network communication. The service or the bundle will be installed locally, and the method invokations of the service will be satisfied by the local replica of the service. Of course in this case the framework should be dealt with the consistency of the replicas and their communication with the real object. A policy that will switch among these two cases would be very a possible solution. Its goal would be to choose whether this service is accessed many time by specific nodes and consequently replicate it to these nodes, or the remote access of the service is sufficient for the amount of the remote calls for its methods.

Virtual OSGi framework up to now operates in desktop computers. An extension would be to operate in embedded devices. Virtual OSGi can be a useful infrastructure for a lot of embedded devices that will participate in a p2p network offering services to the other services that participate. A device can register services to the network, which will be stored in the distributed service registry and they can be retrieved by other devices using the DHT infrastructure. In this case the network is very dynamic because devices might leave and join continiously the network. The DHT algorithms and especially the stabilization protocol, make the system strong against high node churn and as a result makes the Virtual OSGi framework a charming solution for deploying one OSGi framework where a lot of devices participate and offering services to them in a transparent way.

Moreover, future work for the Virtual OSGi framework would be the adoption of streams, especially in the case where large amounts of data should be transferred from one node and they should be processed in another one, such as in the case that one node wants to load a class from the another one and this class has big footprint. With the use of streams, the requesting node, should start receiving data which will be processed immediately, while the receiving of the data will not have finished. Instead of waiting to receive the whole amount of data and after that it will start the processing, it will do this two procedures in a pipelined way.

# Bibliography


[1] felix.apache.org.

[2] http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jnitoc.html.

[3] http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html.

[4] http://labs.google.com/papers/gfs.html.

[5] http://newton.codecauldron.org/site/index.html.

[6] http://open-chord.sourceforge.net/.

[7] http://www.amazon.com/b/?node=201590011.

[8] http://www.emule-project.net/.

[9] http://www.kernelthread.com/publications/virtualization/.

[10] http://www.opengroup.org/projects/soa/doc.tpl?gdid=10632.

[11] http://www.osgi.org.

[12] http://www.w3.org/2002/ws/.

[13] www.eclipse.org/equinox/.

[14] www.knopflerfish.org.

[15] About the osgi service platform. Technical Whitepaper, June 2007.

[16] Osgi service platform, core specification release 4, 2007.

[17] E. Adar and B. Huberman. Free riding on gnutella.

[18] T. O. Alliance. Osgi service platform service compendium, 2007.

[19] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Programming, composing, deploying for the grid.

[20] J. S. S. C. R. A. D. J. Ben Y. Zhao, Ling Huang and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, VOL. 22, NO. 1*, January 2004.

[21] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols*, Portland, OR, Aug. 2004.

[22] S. L. S. S. Changxi Zheng, Guobin Shen. Distributed segment tree: Support of range query and cover query over dht, 2006.

[23] R. L. E. Bruneton and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. Technical report, France Telecom, November 2002.

[24] C. P. E. Guttman and J. Veizades. Rfc 2608:service location protocol v2 ietf, June 1999.

[25] D. R. K. Frans Kaashoek. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[26] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services.

[27] B. Godfrey. A primer on distributed computing. *www*, 2006.

[28] D. Q. L. L. H. H. Greg Boss, Padma Malladi. Cloud computing, 2007.

[29] J. W. H. Garcia-Molina, J. Ullman.

[30] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier, sep 2003.

[31] D. K. M. F. K. Ion Stoica, Robert Morris and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.

[32] G. A. Jan S. Rellermeyer. Concierge: A service platform for resource-constrained devices. In *In: Proceedings of the ACM EuroSys 2007 Conference, Lisbon, Portugal*, March 2007.

[33] G. A. Jan S. Rellermeyer. Services everywhere: Osgi in distributed environments. In *EclipseCon 2007, Santa Clara, CA*, 2007.

[34] C. Johnson. Metallica rips napster. *www*, 2000.

[35] M. F. S. M. L. Peterson, A. Bavier and T. Roscoe. Towards a comprehensive planetlab architecture. Technical report, PlanetLab Consortium, June 2005.

[36] Z. P. L. X. L. Y. Li Wei, Chen Shanzhi. An efficient broadcast algorithm in distributed hash table under churn. In *Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on*, 2007.

[37] e. a. Lin, A.W. The telescience tools: Version 2.0. In *Proceedings of the 1st International Conference on e-Science and Grid Computing, pp. 56 - 63*, 2005.

[38] W. W. D. Y. Qi Xia, Ruijun Yang. Fully decentralized dht based approach to grid service discovery using overlay networks. In *Proceedings of the 2005 The Fifth International Conference on Computer and Information Technology (CIT05)*, 2005.

[39] J. K. Sean Rea, Byung Chun and S. Shenker. Fixing the embarrasing slowness of opendht on planetlab. In *Proceedings of the Second Workshop on Real, Large Distributed Systems WORLDS*, 2005.

[40] M. H. R. K. Sylvia Ratnasamy, Paul Francis and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, 2001.

[41] D. P. A. J. C. E. K. M. L. D. Werthimer. Seti@home: An experiment in public-resource computing. In *Communications of the ACM, Vol. 45 No. 11*, November 2002.

[42] S.-M. S. Z. Zhang and J. Zhu. Somo: Self-organized metadata overlay for resource management in p2p dht. 2003.