The cirrostratus platform concepts, architecture and evaluation

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH zürich

Masters Thesis

The Cirrostratus Platform Concepts, Architecture and Evaluation

by
Samuel Pasquier

Due date
28. March 2009

Advisors:
Jan S. Rellermeyer
Prof. Gustavo Alonso

ETH Zurich, Systems Group
Department of Computer Science
8092 Zurich, Switzerland

Abstract

Networks have an increasing importance in the current world. The number of users of the Internet steadily grows over the past years. Terms like connectivity everywhere start to take shape. More and more information is spread electronically through networks and computation in distributed settings is often required. The vision the Network is the Computer is becoming a reality. Those observations are motivating a growing importance of applications running in distributed environments. However in a distributed environment a difficult task is to find an appropriate architecture. On the one hand computation and information can be, as much as possible, centralized at one location. On the other hand computation and information can be completely distributed like in P2P systems. In between, many variations exist. Every possible architecture has advantages and no silver bullet approach exists. Choosing an architecture is even more difficult in a continually changing world.

This thesis aims at an application model for distributed, modular and dynamic architectures of Information Systems involving zero code adaptation. In this way architectures of applications in distributed environments should be facilitated. For that purpose a platform, called the Cirrostratus Platform, was implemented. Three pillars build the fundaments of the Cirrostratus Platform and enforce the distributed, modular and dynamic properties. Thus, firstly the Cirrostratus Platform fulfills the requirements of a distributed system. Secondly modularity is supported and used for distribution. Thirdly location transparency is provided for the developers and the users. For the evaluation of the Cirrostratus Platform a modularized application is introduced.

Preface

This thesis is a part of the Cirrostratus Platform research project of the Systems Group at the Department of Computer Science, ETH Zürich. It is submitted for the partial fulfillment of the Master Program (MSc ETH) in Computer Science. The duration of the thesis is 6 months, from September 28th 2008 to March 28th, 2009 in the Informations and Communications Systems Group under the supervision of Jan S. Rellermeyer and Prof. Gustavo Alonso.

Acknowledgements Firstly I would like to thank my supervisor Jan S. Rellermeyer for his engagement. Secondly I would to thank Prof. Gustavo Alonso for giving me the opportunity to work on this exciting research topic. Thirdly I would to thank all my friends who supported me during this thesis, as well as my lovely family.

Contents

Preface **v**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Outline of the Thesis . . . 3

2 Concepts **5**
2.1 Distributed System . . . 5
2.2 Modularity . . . 5
2.3 Location Transparency . . . 6
2.4 Overview . . . 7

3 Architecture **9**
3.1 P2P . . . 9
3.2 Modular Architectures . . . 9
3.2.1 SOA . . . 10
3.2.2 Loose Coupling . . . 10
3.3 Virtualization . . . 10
3.3.1 Duplication of Modules . . . 11
3.4 Summary . . . 11

4 Implementation **13**
4.1 P2P Implementation . . . 13
4.1.1 DHT . . . 13
4.1.2 Data Distribution . . . 17
4.1.3 Communication API . . . 20
4.2 Bundle Layer . . . 21
4.2.1 OSGi Bundle Layer . . . 21
4.2.2 Cirrostratus Bundles . . . 23
4.3 Service Layer . . . 27
4.3.1 OSGi Service Layer . . . 27
4.3.2 Cirrostratus Services . . . 29
4.3.3 Cirrostratus Service References . . . 29
4.4 Virtual Layer . . . 31
4.4.1 OSGi BundleContext . . . 31
4.4.2 The Virtual Layer Bundle . . . 31
4.4.3 Migration & Replication . . . 32

5 Related Work **35**

6 Evaluation **37**
6.1 Stendhal . . . 37
6.1.1 Game . . . 37
6.1.2 Advantages . . . 38
6.1.3 Adaptations . . . 39
6.2 Experiment . . . 41

7 Conclusions **43**
7.1 Future Work . . . 44
7.1.1 Replication Strategies . . . 44
7.1.2 Performance Analysis . . . 44
7.1.3 Open Issues . . . 45

Chapter 1

Introduction

1.1 Motivation

Under the term *The Network Society*, Jan van Dijk defined a form of society increasingly organizing its relationships in media networks gradually replacing or complementing the social networks of face-to-face communication [14]. Even if the validity of this claim in our current society can lead to discussions, there are certain observations advocating for the growing importance of networks. From the early ages of the first email to the current Internet state, the distance between a computer and its network was constantly reduced. E.g., according to statistical studies ([13, 19]) the amount of people using the network has steadily increased. Nowadays, more than half of the Swiss population use the Internet multiple times per week. Looking around on social networking website, downloading music or playing on line games are today common activities. Statistics shows also that network connections were steadily improved [13]. For example the proportion of DSL connections in Switzerland jumped impressively from 7% in 2000 to 69% in 2007. Moreover, thanks to technologies like EDGE, UMTS [3], or even Wireless LAN (Wlan), the expression *connectivity everywhere* started to take shape. For instance, under the slogan, *Internet Everywhere*, telecommunication's providers sell nowadays products allowing everyone to be on line as long as GSM connectivity is available. In western countries the GSM coverage is close to 100 % of the population. At a world wide level, approximately 80 % of the population is covered by mobile networks [3]. Obstacles to a wide use of Internet's connectivity based on GSM technologies are currently the high prices and the relatively low available bandwidth. However this domain is constantly on the move and new technologies like WiMax or 802.11n Wlan might furthermore improve the *connectivity everywhere* idea. In some sense the vision of John Gage [36]: *The Network is the Computer* is becoming more real every day. Computer are more and more linked together.
The motivation to distribute information does not come as a surprise. Sources and sinks of information are naturally distributed, it makes sense to support a natural flow of information. The concrete realization, however, is subject to controversy. In outline there exist two primary types of computing: Centralized Computing and Distributed Computing.
Centralized computing can be defined as computing done at a central location

[35]. Information is then propagated through terminals to the end user. In principal accessing a web application with a browser is making use of the centralized computing paradigm, as long as the browser does almost not process the received data. On the other hand, distributed computing is any computing that involves multiple computers remote from each other, such that each have a role in a computation problem or information processing [29]. Here it is thus primordial that each individual workstation performs a bit of the global work. A typical example of the distributed computing paradigm is a Peer-to-Peer (P2P) file sharing application.
Centralized computing has the advantage of being efficient. Indeed much overhead can be avoided and the system can be highly optimized [17]. Moreover, only one site needs to be deployed and maintained. Finally the administrator has full control over the system. In contrast, distributed computing make use of the resources of every participating computers. Another advantage is scalability: Building a bigger system can just be done by adding another computing unit. Furthermore, distributed computing avoids the needs to build pricey electrical power, environmental controls, and extra infrastructure that a supercomputer requires. Moreover, a distributed system makes it possible to share the responsibility between the participants.
Even if having a distributed computer system is in certain circumstances a desirable goal, constructing a functioning system is known to be difficult. Failures, limited local knowledge and asynchrony are fundamental difficulties of distributed systems [11]. Additionally there are more pervasive problems like for instance hardware heterogeneity.
Although a clear separation was just presented between centralized and distributed computing, any implementations can compose with both concepts to build an appropriate solution. For instance, instead of running solely all the computation on one single machine in a completely centralized system, the model can be changed to support shared computing between multiple computing instances. The result is a distributed back-end system. Despite that modification the client computers do not notice any difference; the distributed back-end still appears as one single unit. In general such system evolution might be justified or even required in certain conditions.
However, turning one computing and communication model to another one is not an easy task. Most systems require the code to be tailored to a specific model. Therefore, changing the computing and communication model require in depth-code adaptation. For example, Java Servlets [4] are coded with the expectations of a client-server architecture. Turning an application written as Servlets into a P2P application is a complete redesign and requires substantial rewriting of the code.
The platform presented in this thesis solves this issue. In fact, this platform works for every computing model and the code running in one model will run in another model without adaptation. Assuming again a distributed back-end system with an attached set of clients running on top of the platform, the application gains the flexibility of operating in completely different modes. One extreme could be that clients should share some part of the computing with the back-end nodes. Another possibility was that only a single node is responsible for a certain part of the system. Through this flexibility, those adaptations can be triggered automatically at runtime to optimize performance.
The architecture of Information Systems is not fixed anymore, it becomes mod-

ular. Thanks to a real decoupling of the functions and the architecture, an architecture can be changed without code's adaptation. Additionally code can also be modified without implying an architecture change. Since every computing model offers benefits, the final goal of a modular architecture is to adapt correspondingly to obtain a maximal benefit.
This thesis describes a platform called *Cirrostratus*. The Cirrostratus Platform is a virtual distributed runtime build on the top of multiple local runtimes. Although the Cirrostratus Platform may be distributed over multiple locations it offers the same functionalities as one single local runtime. Three concepts are fundamental to this platform: Distributed System, Modularity and Location Transparency.
Although the Cirrostratus Platform doesn't have to work in a distributed manner, it can. Therefore, as outlined in 2.1, the Cirrostratus Platform fulfills the requirements of a distributed system. Modularity offers many benefits like a clearer separation of concerns and an improved maintainability [25]. In the case of a distributed environment, modularity is even more powerful because modules can be distributed to improve performance. Modularity in the Cirrostratus Platform is described in 2.2. Location transparency ensures that the developer doesn't have to introduce location dependant aspects. For the user, location transparency provides a similar application's access at every edge of the network. Thus, location transparency matches the vision *the Network is the Computer*. Although the Cirrostratus Platform does not completely prohibit the control of programs' location, it advocates for a high level management of program's distribution. Location transparency is the subject of section 2.3
Each of these three concepts is necessary to achieve a distributed, modular and dynamic architecture of Information Systems. Distribution is needed to cover the cases where the actual architecture is distributed. Modularity allows a distribution where the resulting load can be shared between the distributed participants. Location Transparency is a required abstraction to guarantee a decoupling of functions and distributed architecture.
Even though the Cirrostratus Platform is a novel and highly innovative approach to cope with running software in a much more flexible manner, it partly builds up on well-established components. The principal ingredients are an underlying P2P Networks, a support for Modular Architecture and a Virtualization Layer. As a concrete runtime the OSGi Service Delivery Platform was selected. The OSGi platform is Java-based and was specified by an open standards organization (the OSGi Alliance [6]).

1.2 Outline of the Thesis

Chapter 2 presents the three main concepts behind the Cirrostratus Platform. Their corresponding architectural decisions are described in chapter 3. Chapter 4 explains how each of these concepts was actually implemented. In chapter 5 the current state of the prototype implementation is evaluated with a modularized application. Chapter 6 describes some related work. Finally, chapter 7 suggests some paths for future work and concludes this thesis.

Chapter 2

Concepts

2.1 Distributed System

Since in the Network Society people are linked to one another and have access to information and communication with one another constantly, distributed systems take a fundamental place in the society.
Even though there is no coherent single definition of a distributed system, a consensus about the necessary achievement of a distributed system exists. A distributed system should ensure: (1) that a collection of autonomous computers appears (2) to its users as a single coherent system [31], [2].
According to these requirements, the Cirrostratus Platform is conformed to a distributed system. Indeed the Cirrostratus Platform is build atop local runtimes running on autonomous computers (1). Despite the autonomy of the local runtimes, they appear like a single runtime to the user (2). Thus both conditions are fulfilled. In reality the Cirrostratus offers even further guarantees than coherency for the user. The resulting distributed runtime is as complete as any underlying local runtime since it provides exactly the same functionalities.
Furthermore Cirrostratus offers location transparency. For the user, as well as for the application developer, the actual location of a running application component is transparent. However in does not mean that the location can't be controlled at runtime. Location transparency is explained in 2.3.

2.2 Modularity

Modular Design is an approach that subdivides a system into smaller parts. Each of these parts can be independently created and then used in different systems. In programming, modularity brings a clearer separation of concerns. Since modules can be substituted by other functionally equivalent modules the maintainability and reusability of a modular software system is improved [25]. Moreover modules can be added to the system without affecting the rest of the system.
The Cirrostratus Platform expects from the applications to be modular. Furthermore, to enable the deployment of modular applications, the local runtimes are required to implement a modularization layer. Different to other approaches like J-Orchestra [30], the Cirrostratus Platform does not attempt to automati-

cally partition programs. Automatic partitioning of the program often requires an understanding of the applications internal structure [21]. For instance this approach offers no assistance for highly dynamic interactions and for parallelization. Moreover it provides no control for developer with advanced requirements. In the Cirrostratus Platform, modularity is an assumed property of the programs.
The Cirrostratus Platform takes further advantage of modularity. Since the system is subdivided into smaller parts, these parts can be distributed over the autonomous computers. This would not be possible with a monolithic large system. The goal is that even though a system is very large and contains many modules, each node only needs to know about a small subset of modules. In this way the potential functionalities scales at the level of the network while only the actual used functionalities need to scale at the level of a single machine. Furthermore the performance can be optimized across the network. For instance less powerful nodes can try to delegate as much work as possible to more powerful nodes by requesting only light weight modules.

2.3 Location Transparency

The Cirrostratus follows the idea that every application access should be independent of the current user location. In the context of "*the Network is the Computer*", it should not make any difference if the user resides in front of a specific machine since he is using the same global entity everywhere. This idea is shared by the *Cloud Computing* paradigm. In Cloud Computing the accesses to services do not necessitate any specific knowledge of where they are located [34]. Cirrostratus goes one step further. The deployment's location of an application or of some application modules has no restrictive influence on the future location of that module. The Platform looks similar at every location. The control of the distribution of the different modules happens at a higher abstraction level. In this way, developers do not need to take care of application's location during the developing process. The deployment is also made without consideration to the locations.
Some more critical points of view about location transparency (e.g.,[34]) states that the cloud cannot be fully transparent. Developer must be aware of the network failures and performance limits. In this way the developers should get the control over how their customers are being served.
This work takes the position that a layer dealing with network's shortcomings between applications and the network is full of meaning. Developers should not deal with explicit network difficulties. Instead they should deal with the fact that modules are not always available for other modules (see Section 3.2.2). This is the level of abstraction at which application's developers are involved with failure handling. Therefore the complexity of handling network difficulties is assimilated into the Cirrostratus Platform and applications can run completely unaware of network issues. For the applications which need to be aware and want to interact with the Cirrostratus Platform, interfaces are offered.
Furthermore it can be discussed if the distribution of the application or application's module should really be under the control of a human being. Automatic distribution of modules based on sensors may deliver better results. However, this question is beyond the scope of this thesis and subject of future work. This

work advocates that, in any case, the control of modules' distribution should happen at a higher abstraction level than application development. A static deployment's configuration is also not suited since networks are dynamic.
With a full location transparent approach, modular and dynamic Information System's architecture becomes possible.

2.4 Overview

Figure 2.1 summarizes the three main concepts underlying the Cirrostratus Platform.

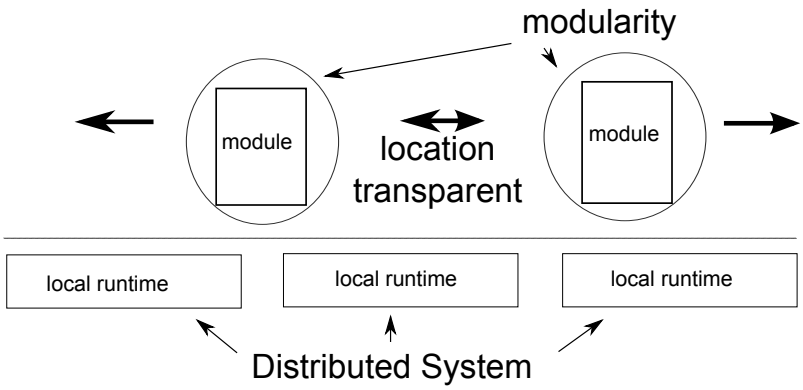


Figure 2.1: The three main pillars of Cirrostratus.

The distributed systems pillar is the result of an observed inherent need for distributed information in the society. Modularity is a way to achieve a dynamic distribution of concerns. Without modularity it would not be possible to distribute diverse and complementary subsystems to different locations. A two-fold location transparency is introduced. Developers should not be required to deal with where their modules will be deployed and users don't need to have prior location knowledge in order to access functionalities. In figure 2.1 location transparency is illustrated by the fact that modules can move freely within the borders of the distributed system.

Chapter 3

Architecture

3.1 P2P

Connectivity everywhere is a fundamental assumption of the Cirrostratus Platform. This assumption is motivated by observations showing that, largely thanks to wireless technologies, connectivity everywhere is becoming a reality. A direct consequence of this assumption is the choice of an *All-to-All* communication model. An All-to-All communication model assumes that every machine (node) is able to directly communicate with every other machine. The Cirrostratus Platform always expects that every node is able to contact every other node currently in the network. Therefore a self-routing algorithm is not part of the Cirrostratus Platform.
On the other hand the Cirrostratus Platform does not assume that a single specific node is constantly running. Machine can join, leave, voluntarily or not, the network at any point in time.
This second assumption implies a fault tolerant architecture and leads to an important design decision of the Cirrostratus Platform: The Cirrostratus Platform needs to implement some P2P decentralized system. Indeed, it should be avoided that a single node or a relatively small group of node might block the system by just leaving it. Furthermore, as demonstrated in section 4.1.1, a P2P approach has good properties in terms of scalability.

3.2 Modular Architectures

Modular architectures refers to the design of any system composed of separate components, modules, that can be connected together [5]. An architecture, without clear divisions between components, is called an integrated architecture. Whereas modularity is a property of a system, a component framework delimits certain rules that modules should fulfill in a certain environment [33]. For instance the way modules can be connected and their interface is defined by the component framework.
The Cirrostratus Platform expects from the applications running on top of it to follow a modular architecture. For that purpose, the description of the modules' connections is an integrated part of the modules' development. The application's developers are in charge of declaring the dependencies between modules.

Thanks to the provided information the platform can then attempt to satisfy the dependencies. However, as discussed in section 3.2.2, a module can not always expect that all the required modules are available.
Complementary to the static connections between modules, within the Cirrostratus Platform, modules can collaborate dynamically at runtime thanks to a Service Oriented Architecture (SOA).

3.2.1 SOA

SOA is an architectural style whose intention is to obtain a loose coupling between the interacting software agents [18]. A service is a work unit executed by a service provider to deliver an end result to a service consumer. Different to object oriented programming, SOA follows the idea that the data and the processing of data should not be binded. In order to achieve loose coupling between provider and consumer, a collection of interfaces, containing only generic information and universally accessible, are published by the provider for the consumers. This means that some mechanism has to be introduced to enable a consumer to discover what can be actually used as services.
In the Cirrostratus Platform the role of a service consumer or provider is played by a module. Therefore services allow dynamic interactions between modules and reduce the coupling between modules.

3.2.2 Loose Coupling

Loose coupling can be defined as a way of interconnecting modules in a system or network whereby each module is as independent as possible of the other modules [29].
However since modules can come and go, no guarantee can be given that a module will ever get all its dependencies fulfilled. Therefore modules might reach a state where they can't work properly. Moreover services exposed by modules can also be added and removed from the system at any point in time. Hence, some required services might be unavailable.
The policy of the Cirrostratus Platform is to expose the problems of loose coupling explicitly. Any missing component is visible to the developer. This means that the developer has to cope explicitly with some uncertainty but the Cirrostratus Platform gives him information and tools to succeed. Eventually the developer might decide to make the unavailability of components visible to the user.
A direct benefit of this approach is that network problems don't need to be treated as a special case. For a service consumer, it doesn't make any difference if a service is unavailable because it was voluntary removed by the provider or because some connection is momentarily disrupted.

3.3 Virtualization

Virtualization is the abstraction from computer resources. Various level of virtualization exists. It ranges from full virtualization (complete abstraction of the underlying hardware) to only storage virtualization. Usually the benefits of virtualization are a better resource utilization and more flexible and secure

environments [27].
In the Cirrostratus Platform, virtualization can be understood as the "glue" between the local runtimes and the platform to achieve location transparency. For that purpose a layer between the local runtimes and the modules is introduced. This layer intercepts the interactions of the modules with the local runtimes and confers them a global semantic. A global semantic means that every module looks similar on every node and that no module is dependent on the availability of a specific node. The deployment of a module on one node should be equivalent to the deployment on another node. Therefore the existence of a module should not be circumscribed by the survival of a node. The concrete implementation of policies to ensure this is beyond the scope of this work.
A necessary instrument to achieve a global semantic is the duplication of modules and their exposed services.

3.3.1 Duplication of Modules

In order to afford the loss of some modules' instances, modules need to be replicated. Since the exposed services are an essential part of the modules, they need to be duplicated as well.
The process of replicating a module consists of creating multiple module's instances out of one original module. Each of the instances represents semantically the same object and none of them is central or essential. Thus, if one of the replica fails it can seamless be replaced by another replica. Semantically equivalent instances need also to be kept consistent as they evolved. Therefore some coordination between the instances is required.

3.4 Summary

Figure 3.1 illustrates the three main architectural decisions of the Cirrostratus Platform. First it was decided to introduce a P2P infrastructure to cope with two assumed network properties, which are an All-to-All communication model and a non-reliable node model. Modules can be connected to other modules. Modules can also interact dynamically with each other by the way of services. To reach the same runtime's functionalities as one local runtime, a virtual layer is introduced. Location transparency should be accomplished by replication of modules and their services. Duplication of modules implies coordination.

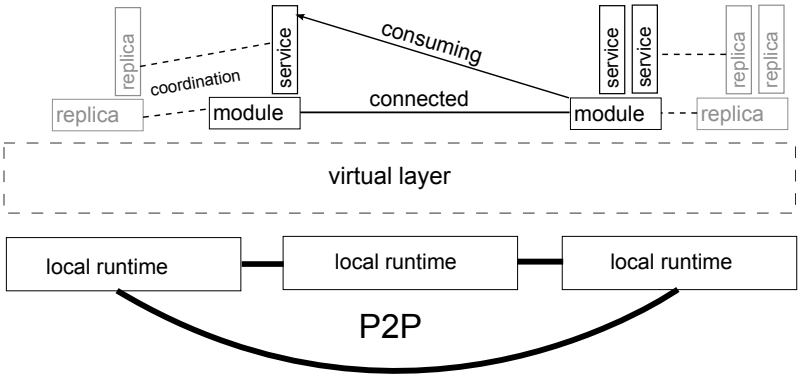


Figure 3.1: The architectural design decisions

Chapter 4

Implementation

4.1 P2P Implementation

In order to implement the P2P design, the Cirrostratus Platform requires a mean to cope with the dynamics of P2P environments. A structured P2P technology called Distributed Hash Table (DHT) was picked. The next section depicted the reasons of this choice and the concrete Cirrostratus implementation.

4.1.1 DHT

DHTs can be defined as a method for storing hash tables in geographically distributed locations in order to provide a failsafe lookup mechanism for distributed computing [1]. A hash table is a data structure associating keys to values. In a DHT, the responsibility for maintaining the mapping from keys to values is distributed among the nodes. For instance the Internet Domain Name System (DNS) is an application of a DHT [32].
In general a DHT's implementation use a variant of consistent hashing. Consistent hashing provides the same methods as a hash table but with an additional advantage: The addition or removal of one storage slot, one DHT node, does not significantly change the mapping of keys to values. This is a valuable property since it avoids large overhead when a node joins or leaves some DHT. Furthermore consistent hashing tends to balance load, since each node receives roughly the same number of keys.
One drawback of structured P2P, and thus DHT approaches, is their difficulty to handle non-exact queries [32]. This maybe partly explain the reason why structured P2P system are not yet widely used in famous P2P applications such as Gnutella. However this drawback is not relevant for this work since the Cirrostratus Platform does not involve approximate queries.
For the Cirrostratus Platform, a concrete instance of DHT, the *Chord protocol* [28], was implemented.

Chord

Chord is a distributed lookup protocol for P2P applications. Like a DHT instance, given a key, it maps this key onto a node.
In Chord any one node needs to coordinate with only a few other nodes in the

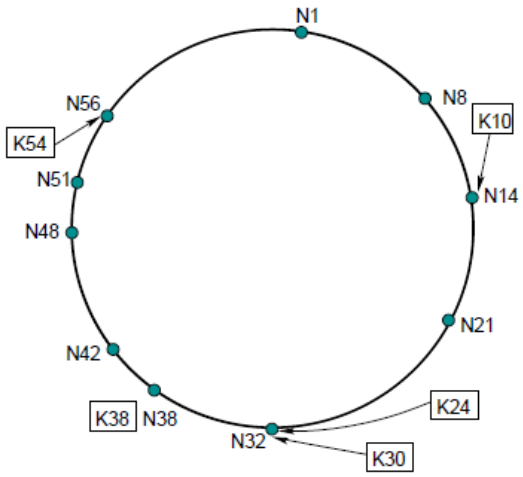


Figure 4.1: A Chord ring, for m = 6, consisting of 10 nodes storing five keys.

system. This is the principal technique used to achieve good performance. In fact, in the steady state, every node only stores information about $O(\log(n))$ other nodes. Furthermore in the steady state, an upper limit of $O(\log(n))$ messages is given for each lookup. Hereby is scalability guaranteed. Moreover the performance degrades gracefully when the routing data are out of date. A single correct routing information is sufficient to ensure proper lookups. The core part of Chord is an algorithm maintaining the routing data even when nodes are joining or leaving the network at a high rate. In this way high availability is ensured. Adding or removing one node from the network happens with very high probability only at the cost of $O(\log(n^2))$ messages. Furthermore the Chord protocol addresses the problem of load balancing. Like a distributed hash function, it spreads keys evenly over the nodes. This provides a degree of natural load balance.
Those excellent characteristics justify the choice of using Chord as P2P implementation. Next is a deeper description of Chord. Later on, in 4.1.2 , it is described how this DHT infrastructure is actually utilized in Cirrostratus to store custom data structures.

Protocol First of all each participating node in Chord is assigned an identifier (ID). In the current case, a Secure Hash Algorithm (SHA) processes the node network's information to transform it into a sequence of m bits. Every key of the DHT is also generated in the same way, implying a transparent handling of node IDs and value IDs. Those IDs are then ordered on an identifier circle modulo 2^m. For a key k , the first node ID equals or following the ID of k is called the successor of k. Figure 4.1 shows that when identifiers are represented as a circle of numbers from 0 to $2^m - 1$ they form a ring. Then the successor of k is the first node clockwise from k.
Each node also stores a routing table called a finger table with m entries. The i^{th} entry in the table at node n contains the identity of the first node s that succeeds n by at least $2^i - 1$ on the identifier circle. This additional table is needed to achieve the targeted lookup time of $O(\log(n))$ with high probability. 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. This is an intuition for the binary search concept hidden in Chord and utilized for broadcasting.
Additionally each node knows about its predecessor, a necessary information for Chord ring stabilization.
Based on those routing data, lookups can be processed. Figure 4.2 describes the lookup algorithm.

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    n' = closest_preceding_node(id);
    return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
  for i = m downto 1
    if (finger[i] ∈ (n, id))
      return finger[i];
  return n;
```

Figure 4.2: Lookup algorithm in Chord

If the searched id falls between n and its successor, find successor is finished and node n returns its successor. Otherwise, n searches its finger table for the node n' whose ID most immediately precedes id, and then invokes find successor at n'. The node n' is especially picked liked that because the closer n' is to id, the more it will know about the identifier circle in the region of id.

Maintenance However this lookup algorithm needs at least some bit of correct routing information to perform correctly. Since node might leave or join freely routing data can constantly change. Therefore some maintenance is required. Figure 4.3 describes the pseudo code for joins and stabilization methods. When starting each node either call the `join` method with a known node n', which is part of the DHT, or it creates a new Chord network with `create`. Each node runs the `stabilize`, the `fix_finger` and the `check_predecessor` methods periodically. The `stabilize` method is utilized to learn about newly joined nodes. A `fix_finger` call checks if the finger table entries are correct. The `check_predecessor` is called to clear the predecessor pointer in case that node failed.

Cirrostratus customizations An additional routing data structure was implemented for Cirrostratus. It is called a successor cache and stores the N next successors. This is used to improve performance in often varying smaller sized Chord rings. This data structure needs to be maintained as well and can be utilized for values replication (Section 4.1.2).

```
// create a new Chord ring.
n.create()
    predecessor = nil;
    successor = n;

// join a Chord ring containing node n'.
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);

// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';

// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers()
    next = next + 1;
    if (next > m)
        next = 1;
    finger[next] = find_successor(n + 2^(next-1));

// called periodically. checks whether predecessor has failed.
n.check_predecessor()
    if (predecessor has failed)
        predecessor = nil;
```

Figure 4.3: Stabilization methods

Furthermore a cache of random nodes currently present in the DHT is maintained. This cache can be utilized for a later rejoin of the DHT.
The Cirrostratus Chord implementation also optimizes the maintenance processes in updating the different data structures with different periodicities. For instance the successor is more often updated than a finger entry. Moreover when a communication lost is observed, the maintenance tasks are immediately triggered.
More detailed information about the Chord protocol, its guarantees and their proof can be found in [28].

4.1.2 Data Distribution

Based on the ability to find a value based on some key, three basic primitives were defined: the insertion of a value, the deletion of a value and the lookup of a value.

- `NodeInfo insertObject(Registry reg, Object value)`
- `void removeObject(final Registry reg, Object value)`
- `Set lookupObject(ID key, Object criteria)`

Each of this primitives makes use of the DHT to locate the place holding the value. A `Registry` is more complex than a simple object. For instance a `Registry` directly contains its corresponding key. In this way every objects extending a `Registry` can, thanks to its definition, be stored in the DHT. In fact, as it will be presented later, the Cirrostratus Platform has the necessity to store different kinds of objects. For the rest of the thesis the terms value and Registry will be used interchangeably.
All data structures indexed in the DHT follow a double indirection mechanism schemes. Instead of having the data structures stored directly at the node indicated by the key, only a location's information is stored at that place. Thus, a Registry is just a light pointer. As illustrated in Figure 4.4, a data structure access consists then of one lookup in the DHT and one request to the indicated location.
This double indirection avoids a costly actualization's task of complex values with original copies. Furthermore it is then not necessary to transfer those potentially large data structure each time a change in the mapping occurs due to the arrival or departure of a node as described in 4.1.2.

Extended Registry

In a original DHT implementation the values are only retrieved in accordance to their keys. With extremely high probability no two values have the same key, and the result of a lookup is an unambiguous value. In the Cirrostratus case, it is a desirable property to register multiple values under one key. Indeed those multiple instances represent semantically a single abstraction which should be stored under, and retrieved with, one ID. But in some cases it might be useful to apply selectivity between the different instances during the lookup. Cirrostratus DHT registries offers therefore the possibility to be filtered. A subtype of a Registry is free to contain some properties, as well as logic for matching those

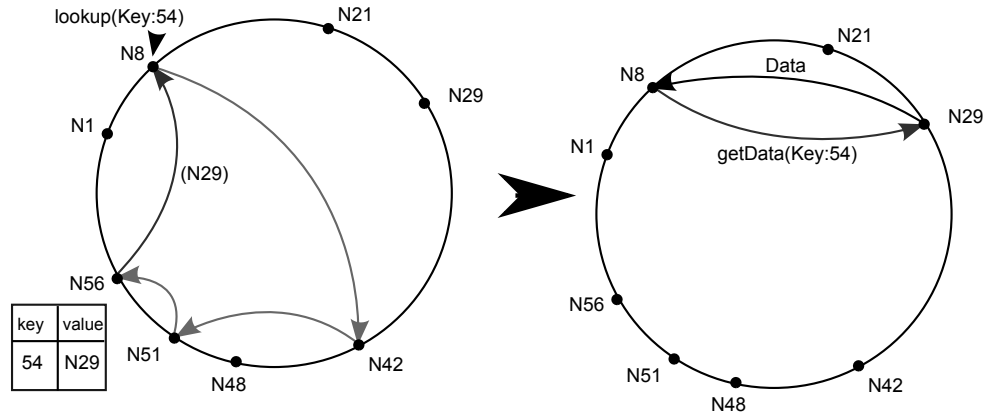


Figure 4.4: A double indirection access consists of two phases. First a key (54) is searched in the DHT. The result is an information (N29) indicating the location of the data structure. Thanks to that information the wanted data structure can be accessed

properties. The characteristics to check against are provided by an additional argument, `criteria`, during the lookup. A `null` criteria means no filtering.
As previously mentioned, DHT registries do not contain directly data structures but light pointers. If no selectivity was pushed to the to DHT Registry, then the node starting a lookup would need to contact all indicated pointer's locations to define a final set of matching data structure. Even if the data structure were directly returned as a result of the lookup, it might imply unnecessary overhead since many of them could be just discarded. The drawback of this approach appears when the filtering properties are modified. In fact, filtering properties needs to be maintained explicitly.
Table 4.1 summarizes the different categories of registry in the Cirrostratus Platform. Details follow in the next sections.

Category	Key	Stored Information
Service Reference	Hash of interfaces	Locations of Service References together with properties
Service	Direct mapping of Service global id	Location of Services
Primary Bundle	Hash of bundle Symbolic Name + Version	Location of Primary Bundles
Secondary Bundle	Hash of bundle Symbolic Name + Version	Location of Secondary Bundles
Package	Hash of package name	Location of cluster containing package

Table 4.1: The different Registries stored in the DHT

Registry Replication

A DHT alone does not guarantee that values are safe. If a node crashes its stored Registries are lost. What is additionally required is some kind of replication of the Registries and logic to maintain sufficient redundancy.
As described in Section 4.1.1, the Cirrostratus's Chord implementation already maintains for each node a list of the next successors. This list was employed to define the location of the replicas. The following rule applies: Each node maintains a replica on the `m` successor nodes. This rule is illustrated on Figure 4.5.

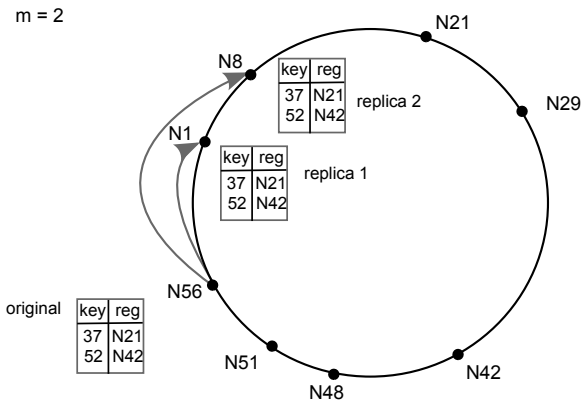


Figure 4.5: Registry's replication on the m successor nodes

The maintenance task incorporates the propagation of the insertions, deletions and updates to the replica, as well as the reinsertion of the lost registries and the deletion of deprecated replicas. The propagation of the registries modification occurs as part of the same transaction as the principal registries modification. Therefore it is ensured that a replica is equivalent to the original copy. In order to check if some registries were lost, each node regularly controls if the nodes for which it holds replicas are still alive. If not, it reinserts the potentially missing registries in the DHT. Since each Registry is only stored once, it does not harm if the lost registries are reinserted multiple times by different replicas. Duplicate insertion's attempts are simply ignored.
Each node knows that it should own exactly `m` replicas. If more replicas are present on one node, then some of them are considered as deprecated, namely the ones corresponding to the nodes with the largest distance. In this lazy manner replicas are deleted. Thanks to the fact that a new node directly inserts its replicas on its successors, a replica is first considered as deprecated before the node with the largest distance stops propagating the modifications to this replica.

Registry Maintenance

Since the node responsible for a specific key might change over time, care has to be taken to reallocate registries if needed. From time to time each Registry's

key is checked. If the key does not belong to its current location anymore, the Registry is reinserted in the DHT and deleted from the current node.

4.1.3 Communication API

Internally the Cirrostratus uses two communication primitives, one blocking the other one non blocking. Depending of the nature of the interaction the appropriate primitive has to be utilized.
Both techniques are implemented with message exchanges. For each received message a new thread is created. In this way a chain of interactions resulting in a deadlock situation is avoided.
For some use cases, like event broadcasting (see Section 4.4.2), these primitives alone were not powerful enough. Therefore a blocking, and a non blocking broadcast, methods were implemented.

Broadcast methods

For broadcasting, the Cirrostratus Platform makes us of the underlying DHT structure. It implements a proposal presented in *Efficient Broadcast in Structured P2P Networks* [16].
The idea is to consider a DHT systems as form of distributed k-ary search, where Chord is a special case with k=2. Indeed a Chord lookup mimics a binary search. A more detailed explanation of the similarity of Chord with a binary search can be found in [15]. As illustrated in Figure 4.6, a k-ary search tree is also a spanning tree. An efficient broadcast can be performed along this spanning tree.

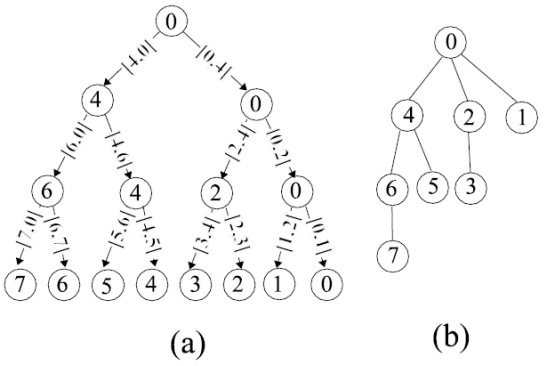


Figure 4.6: (a) Decision tree for a query originating at node 0 in a fully-populated 8-node Chord network (binary search). (b) The same decision tree without virtual hops, also a spanning tree covering the whole system.

The broadcast works in the following way: One node initiates the broadcast by sending a message to all its children in the spanning tree. Every node receiving a broadcast message propagates the broadcast to its own children. In a system of N nodes, the total number of required messages is N-1. This is an optimal bound since N-1 nodes needs to be informed. Moreover this solution gives strong guarantees in terms of coverage. If the routing data of Chord are correct, then every node is reached.

The implemented blocking broadcast variant follows the same algorithm but the spanning is traversed in both directions. The difference is that each node waits until all the children responded. Then it itself aggregates the responses into one message and send it back to its parent. In the end the initiator only needs to unpack all the responses.

Object Serialization

Since Cirrostratus can not control for all the object types to transmit over the network, a custom Serializer is used. This Serializer is able to serialize Java classes which don't implement Serializable. This solution is borrowed from R-OSGi [8].

4.2 Bundle Layer

A requirements to the local runtimes was to implement a modularization layer. The implementation of the OSGi Framework is a modularization layer. In the following two sections, Bundle Layer 4.2 and Service Layer 4.3, the OSGi Framework is presented. Furthermore it is explained how the Cirrostratus Platform extends the OSGi Framework to achieve its modularity objectives.

4.2.1 OSGi Bundle Layer

In the standard Java platform only limited support for modularity is given. For instance dependencies are only implicit. For the successful resolving of JAR files depending on other JARs almost solely a good documentation can save a programmer from a `ClassNotFoundException`. OSGi solves this problem by defining a unit of modularization, called a bundle. Parts of a bundle are Java classes and other resources, which together can provide functions to end users. Java packages can be shared between bundles in a well-defined way. Some bundle can export packages and some bundles can import those packages. The process of matching exporter with importer is called resolution. The resolution process for the Cirrostratus implementation is described in Section 4.2.2.
Bundle are deployed as a Java ARchive (JAR) file. Among other things, this archive contains a manifest file describing the contents of the JAR file and providing information about the bundle. The headers in the manifest file provide important indications to the OSGi Framework but only the `symbolic name` headers is mandatory to create a valid bundle.

Bundle Life cycle

Bundles are managed through a *Life Cycle Layer* defining for instance how bundles are installed, updated and uninstalled. Therefore bundles are stateful objects and their states can change over time given a predefined logic. For instance a bundle can be started through a Bundle Activator implementing the `BundleActivator` interface [7] and therefore move from the state RESOLVED into the state ACTIVE. The complete state diagram is represented in figure 4.7. More information about the life cycle of bundles can be found on the OSGi Alliance Web page [6].

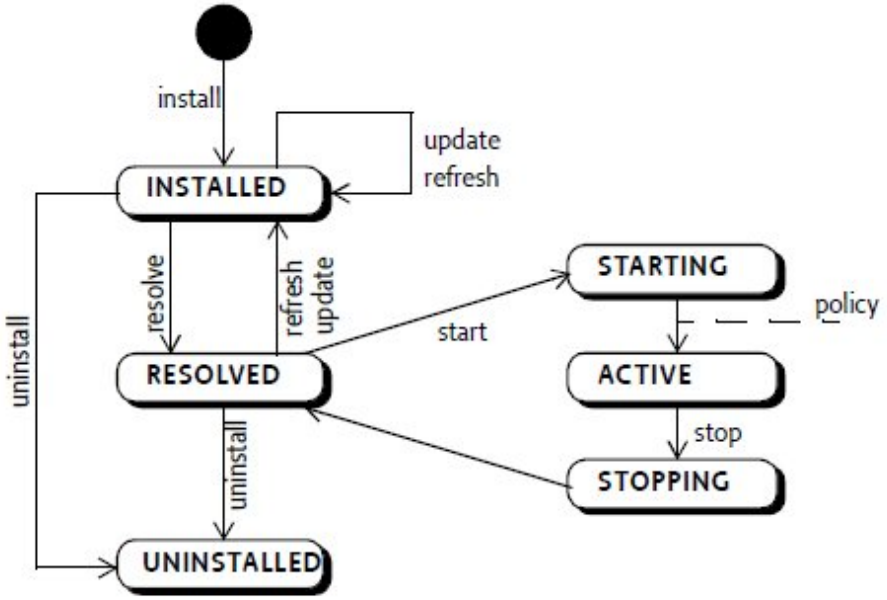


Figure 4.7: Bundle Life cycle from the OSGi R4.1 Specifications [7]

Resolving process

In order to move successfully a bundle from the state INSTALLED to the state RESOLVED, an OSGi Framework needs to resolve the bundle's dependencies as statically described in the manifest. Most frequent dependencies are *import and export packages*, as well as *required bundles* constraints where a required bundle constraints triggers the import of all exported packages from a bundle. Resolving is then the process of matching imports with corresponding exports. In the end, the result is a wiring between bundles. A wire is only possible under certain conditions.
For instance, in order to be a matching package, an exported package needs to satisfy the version constraint. Version constraints are a mechanism whereby an import definition can declare a precise version or a version range for matching an export definition. Likewise attribute matching is a generic mechanism which allows the importer and exporter to influence the matching process in a declarative way. Indeed the attributes specified by the import definition must match the values of the attributes of the export definition, in order to establish a valid match.
Another example are implied package constraints. If a bundle imports a package from an exporter then the export definition of that package can imply constraints on a number of other packages. An example of an implied package is presented in Figure 4.8. The bundle A imports two packages: x and y. Since only bundle B exports package x, A needs to wire to B. Likewise B is wired to bundle C to respect the import of package z. Then this implies a constraint on the import of package y. For that reason, bundle A can't be arbitrarily wired to bundle C or D. It has to be wired to bundle C.

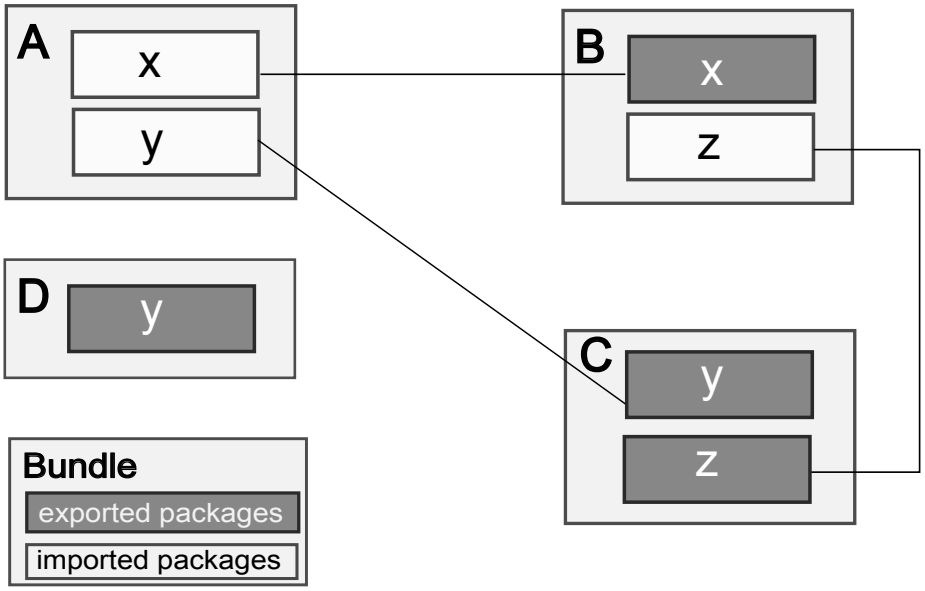


Figure 4.8: Implied packages

Implied packages have the goal to ensure class space consistency. For instance if bundle A would import the class y from D, it might be confronted with objects with the same class name but from different class loaders. This situation can potentially lead to `ClassCastExceptions`.
In fact the presented implied package is a simplification of the OSGi R4.1 specifications [6] where exported packages should list the packages that they depends upon with a `uses` directive. Here it is assumed that exporter depends upon all the packages in the bundle.
How the resolving process, and more specifically the implied package, concepts are implemented in Cirrostratus can be read in the next section.

4.2.2 Cirrostratus Bundles

In the Cirrostratus implementation it is a necessity to distinguish between four different type of bundles.

- Primary bundles
- Secondary bundles
- Redirection bundles
- Proxy bundles

Each of these bundle types implements the `Bundle` interface [7]. They also have in common to wrap a so called *host bundle* object. A host bundle is the result of a bundle's JAR installed on the local OSGi framework, and thus, its implementation is the responsibility of the local OSGi framework. Each wrapper holds supplementary metadata and implements a customized logic for each exposed method of the bundle interface.

Primary bundles represent the bundles explicitly installed on the Cirrostratus Platform and the copy of those bundles created through a replication or migration processes like explained in 4.4.3. Primary bundles replicas needs to coordinate their metadata since they should represent the same logical instance. Therefore the metadata of primary bundles are restored after replication/migration step. In order to keep up to date those metadata, each action performed on one of the primary, is multicasted to all the primary replicas.
Secondary bundles are the result of the resolving process. When a host bundle needs to be installed just to fulfill the dependencies of another bundle, then a secondary bundle is created. Secondary bundles have the same host bundle as at least one primary bundle but they don't need to participate in the coordination of their metadata. Indeed secondary bundles are semantically separate entities.
Redirection bundles are a very light representation of primary bundles. They do not hold any reference to a local host bundle. Instead they contain the location indication of one primary bundle. Each method call is forwarded to this primary bundle. Redirection bundles are created when the local framework needs a bundle representation but this bundle instance is not going to be used extensively. This is for instance the case when a command listing all the bundles of the framework is called.
Proxy bundles are wrapper for a special kind of host bundle, namely host bundle created for proxy services like described in 4.3.2. These special host bundles contain a proxy class as well as a minimal set of necessary types to obtain an operable proxy. Proxy bundles are part of the R-OSGi [26] middleware and were integrated as part in the Cirrostratus Platform implementation.

DHT indexing

Each bundle is uniquely identified by a `SymbolicName` plus a `Version`. Therefore bundles are indexed in the DHT through a key build out of `symbolic name + version`. The same key can be converted into a bundle id as defined in the `Bundle` interface [7]. Moreover, a bundle id can be converted back into a DHT key. This bijectional mapping is a convenient way of avoiding redundant identifier for bundles.
Primary bundles are registered into one Registry category. Among other things, these primary bundle Registries are used for the multicasting of coordination informations between primary bundles. Secondary and Redirection bundles are registered into a second Registry category. The unique purpose of these Registries is to enable the complete clean up of bundles when a corresponding primary bundles is uninstalled. Proxy bundle are not registered at all and their removal can be decided unilaterally by a node.

Resolving Process

The resolving process is recursive, since the resolving of one bundle might trigger the resolution of other bundles. The bundles just installed as part of a resolution process are called secondary bundles. Thus, primary and secondary bundles needs to be recursively resolved. Proxy bundles require a set of imports consisting of all the reachable imports of the original bundle. Therefore proxy bundles needs also to be recursively resolved. Redirection bundles don't need to be resolved since redirection bundles don't hold any other logic than forwarding

calls to the primary bundle.
Usually resolving constraint of implied package requires the wires to be traversed recursively. In Cirrostratus, this would imply contacting potentially many nodes for each resolution process. Hence, a concept which enables the resolution of implied packages with one single remote access is introduced. This concept is called *Clustering*.

Clustering A *Cluster* can be considered as a set of metadata representing implied packages. Metadata stored for each package are, among others, package name, package version, as well as bundle id and bundle version. In this set no two packages can have the same name. This rule is set to ensure class space consistency and represents the same semantic as the implied packages constraint. This is the cornerstone idea of clusters.
A cluster always resides on a single node. Only for a persistence reason, invisible cluster replicas are created (Figure 4.2.2). Clusters are not indexed directly in the DHT, only packages names are. Thus, each DHT package Registry corresponds to a package name and points to the different nodes where clusters containing this package's name are located.
A bundle which wants to resolve will go over each imported packages. For each imported package it will retrieve all the clusters this package name is part of. When asking for a specific cluster, it will also attach the complete list of imported packages. The node responsible for the cluster will check if each submitted import definition matches a package of the cluster. If some cluster's package has the same name as one import but with non matching version or attribute, then this cluster cannot be used in the current resolution. Otherwise the subset of matching cluster's packages is send back to the resolving bundle. Once the resolving bundle has gathered all the possible clusters, it tries to combine them into a (new) cluster respecting the implied constraints. I.e., it is not allowed to merge two cluster containing similar package names. If a solution covering all the given import definitions can be found, the resolution process is successful and the selected clusters are merged into one cluster. Additionally all the exported packages of the resolving bundle are added to the renewed cluster. For example, Figure 4.9 represents a situation of bundle resolution. Bundle A wants to resolve and therefore needs to match three import definitions. After asking in turn for clusters with package x,y and z he obtains cluster 2 and 3, where cluster 2 fulfilled packages x and y, and cluster 1 fulfilled package z. Cluster 1 is no candidate because he does not fulfill the version constraint of z. Since cluster 2 and 3 do not conflict and satisfies all the imports together the resolving is successful. Cluster 2 and 3 are merged into one cluster and the export t of bundle A is added to the newly created cluster.
A cluster always resides on a single node. Only for cluster's persistence invisible cluster's replicas are created (Figure 4.2.2). Clusters are not indexed directly in the DHT, only package's names are. Thus, each DHT package Registry corresponds to a package name and points to the different nodes where clusters containing this package's name are located. When clusters are merged, the location of the biggest cluster is selected to host the new cluster and all the other selected clusters are removed. Hence, clusters are never moved from one node to another node. They always stay at the place they were created, or they get deleted.

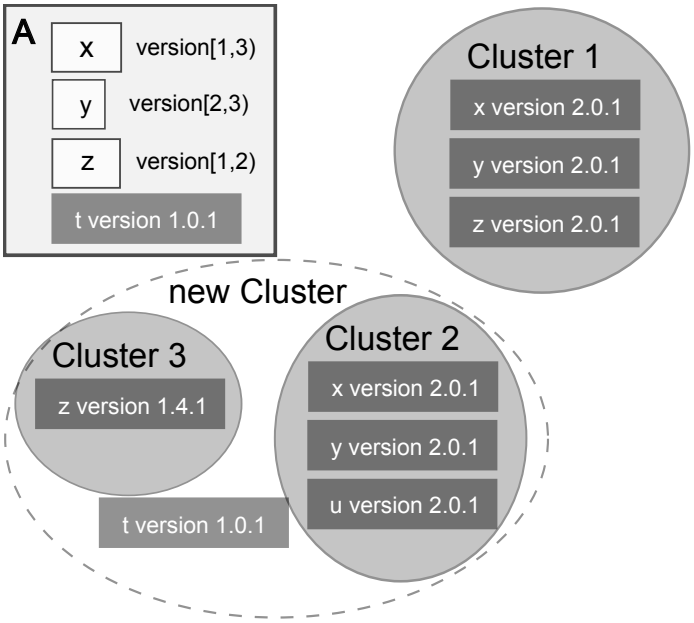


Figure 4.9: Cluster resolution

Additionally a mutual exclusion mechanism is introduced to guarantee consistency against concurrent cluster's modifications [24].

Replication of Clusters The observation that a cluster never moves is true only as long as no node leaves the network. Clearly, the information stored in one cluster may impact more nodes than just the one storing the cluster. Therefore cluster replication has to be implemented to prevent important information lost. The actual implementation stores always a replicated cluster data structure on the successor node. If a nodes notice that the original cluster's node left the network, it will integrate the replicated data structure to its actual cluster data structure.

Example

Figure 4.10 summarizes the different bundle types present on the Cirrostratus Platform. The primary bundle A is replicated at location 1 and 2. Both replicas need to be coordinated. Since bundle A imports package x, it was wired with the secondary bundle B which exports a package x. The original primary bundle B resides at location 3. For some reason, at location 4, a redirection bundle B was created. On this figure only bundle's types visible to the rest of the world are drawn. Therefore no proxy bundle is represented.

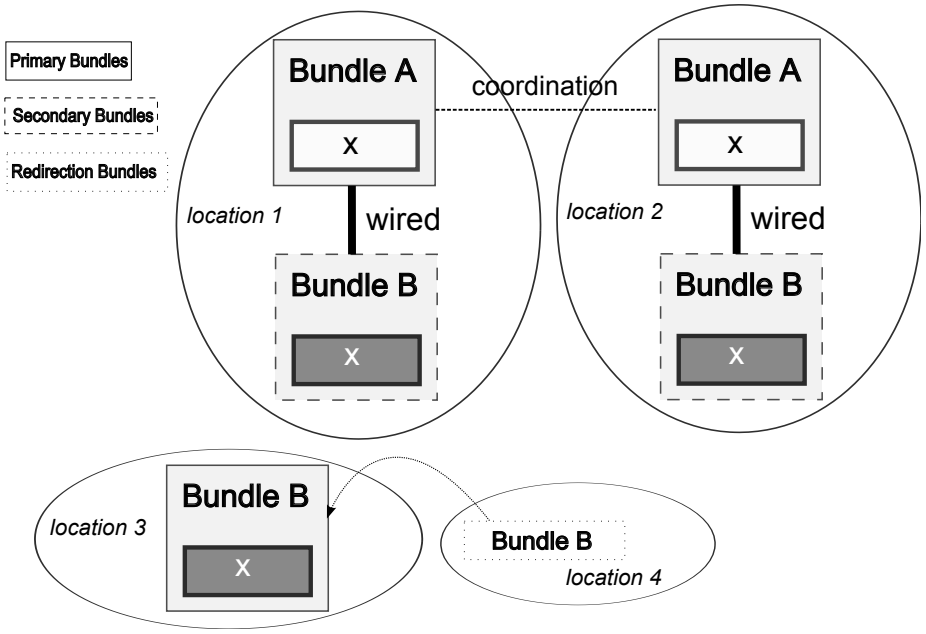


Figure 4.10: Visible Cirrostratus bundle's types

4.3 Service Layer

4.3.1 OSGi Service Layer

An OSGi service is a standard Java object owned by, registered by a bundle. Any bundle can then search for this object and use it. Therefore the OSGi service model follows a standard publish, find and bind model. OSGi services are registered with a Framework Service Registry under one or more Java interfaces and, potentially, together with properties. Thus, the OSGis Service Registry is considered as a form of Service Oriented Architecture [6]. In fact the interface of a service is the only necessary shared knowledge between interacting bundles. For its part the concrete implementation of a service remains a black box for the consuming bundle.
Services are registered by the name of the interface or the names of the interfaces. Service properties hold information which should characterize a service. This information can be important to select between services registered under the same interface. In order to select between services, optional filters are introduced. Figure 4.11 illustrates the successive steps involved in registering and consuming a service.
The returned object of a registration procedure is a sort of receipt called a service registration. A Service Registration can be used to update the properties, to unregister, or to obtain a Service Reference. A Service Reference holds the properties and other metainformation about the service object it represents. A Service Reference is central since, even though, it does not allow a direct access to the service, it can be used by a bundle to obtain it. Dependencies between the bundle owning the service and the bundles using it are managed by the Framework. Acquiring first a Service Reference and not the service itself,

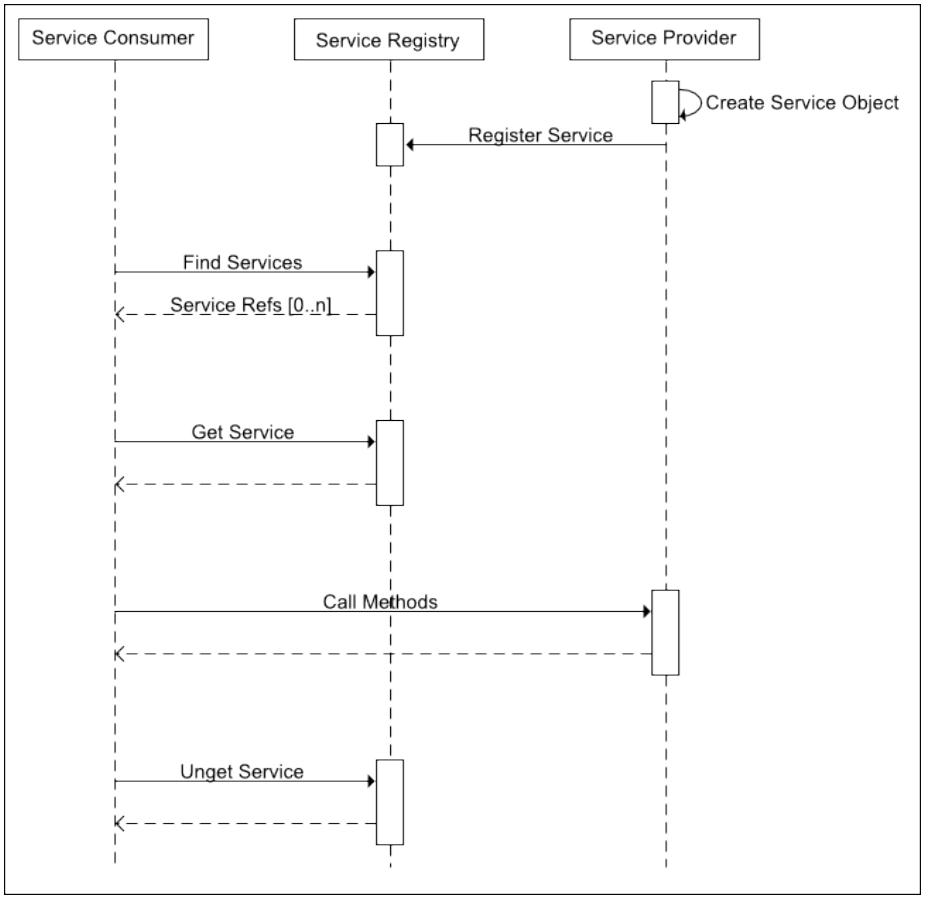


Figure 4.11: Necessary interactions for registering and consuming a service (taken from [12]).

avoids creating unnecessary dynamic service dependencies between bundles, if some bundle only needs to know about a service but does not actually use it.

4.3.2 Cirrostratus Services

The Cirrostratus implementation distinguishes between two types of service:

- Cirrostratus Service
- Proxy Service

A Cirrostratus Service just looks like any other local service. The only difference resides in its ability to replicate and migrate in the network. Cirrostratus Service replication/migration is explained in Section 4.4.3. Replication/migration of services means that multiple instance of the same service can execute in parallel on different nodes. In this way redundancy is achieved and the survival of a service is uncoupled from the node survival.
A Proxy Service implements every method described in the service's interface as a remote method invocation to some Cirrostratus Service. Proxy Service can be created at any time to satisfy the need of a node to access a remotely located Cirrostratus service. In comparison to the replication of a Cirrostratus service, creating a Proxy is a much lighter process. Therefore the creation of a Proxy should be preferred over replication in the case of rare service usage. More detailed explanations about Proxy Services can be found in [26].

DHT indexing

Each Cirrostratus Service receives an unique id when it is for the first time registered on the Platform. This id is also inserted as a property entry of the service. Moreover this property can't be removed or changed and therefore will stay with the service, and its replicas, for their life time. Thus replicas can always be recognized.
Proxy Services also holds a field with the same unique id as the referenced Cirrostratus Service. When a Proxy Service contacts a remote node, it attaches the corresponding id to indicate which service should be used. Moreover if a Cirrostratus Service replica disappears, thanks to this global id, a Proxy Service can always rebind itself to another replica if one exists.
The id is also used as a key for the DHT. One Registry stores the different location holding a Cirrostratus Service replica. On the other hand Proxy Services are not indexed in the DHT since they never need to be addressed by remote nodes.

4.3.3 Cirrostratus Service References

For each type of service a corresponding Service References is defined:

- Cirrostratus Service Reference
- Proxy Service Reference

Both types implements the ServiceReference interface [7] but they differ in the type of service they represent and in their implementation. A Cirrostratus Service Reference is attached to a Cirrostratus Service, a Proxy Service Reference

is attached to a Proxy Service. Each Service Reference always sits on the same node as its corresponding service. For instance this implies that a Cirrostratus Service Reference has to be replicated if its Cirrostratus Service is replicated. Therefore the metainformation and properties of Cirrostratus Service Reference's replicas needs to be coordinated.
A Proxy Service Reference does not hold any other metainformation than the global id, the name of the interface(s) and the location of the current referenced Cirrostratus Reference. Therefore almost every method call is a remote access. When a Proxy Service Reference is created is does not necessary directly implies the creation of a Proxy Service. A Proxy Service is only created when an object instance is really asked by some bundle.

DHT indexing

Cirrostratus Service References are indexed by their interface(s). This means that the interface name is used to generate the key. Then, for each Cirrostratus Service Reference implementing that interface name, the location of the Cirrostratus Service Reference is inserted under the corresponding key. Furthermore, the Service Reference Registry's category makes use of the presented ability, in Section 4.1.2, to push properties filtering as part of the lookup. This implies that for each change to some service reference property, all the Service Reference Registries have to be updated.
Proxy Service References are not indexed at all in the DHT. As for Proxy Services, it is not a required operation to lookup for Proxy Service References. In fact, there is a trick to locate Cirrostratus Service References based on their Service global id. Since each Cirrostratus Service Reference is located at the same place as a Cirrostratus Service, it is possible to lookup the service registries with the global id and to interpret the result as the location of the Cirrostratus Service References. Put in another way, the service registries are a special case of the service reference registries, where the results are equivalent if the service reference registries are searched against every interfaces and the property filter is only set to the global id. A service reference lookup for every interfaces needs to contact every nodes participating in the DHT. Therefore it is best implemented with the broadcast methods presented in 4.1.3. However those methods are much more time and resource demanding than simple lookups. Since the lookup of a service global id is a common operation, it makes sense to preserve two separate Registry categories.

Example

In Figure 4.12, a Service called *HelloWorld* is replicated at location 2 and 3. Since *HelloWorld* is semantically one single service, some coordination between both instances is required. Additionally at location 1 a proxy service is created. This proxy currently accesses the original service at location 3 but if, for instance this location fails, it could redirect its remote calls to location 2. With every Service there is always a corresponding Service Reference at the same location.

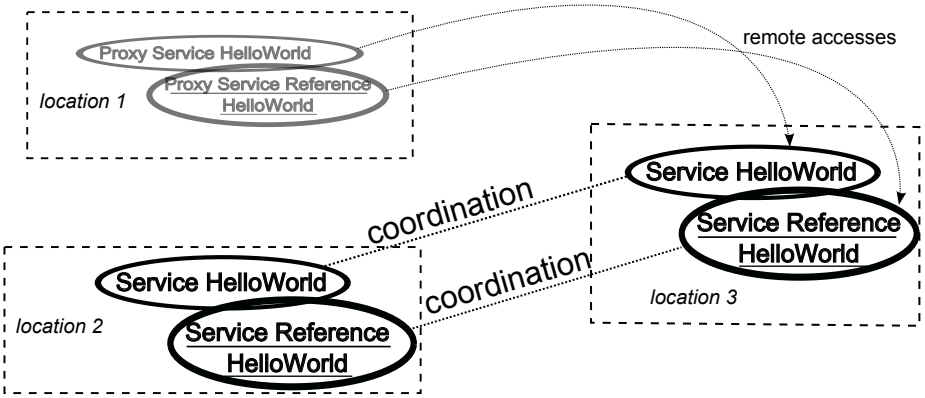


Figure 4.12: A Service, *HelloWorld*, in the Cirrostratus Platform present at three different locations. The Service is represented together with its ServiceReference at each location. Location 2 and 3 holds a Cirrostratus Service whereas location 1 only has a proxy refering to the instance at location 2.

4.4 Virtual Layer

4.4.1 OSGi BundleContext

Every bundle interacts with an OSGi Framework through an object called a *BundleContext*. A BundleContext object represents the execution context of a single bundle within the OSGi Service Platform, and acts as a proxy to the underlying Framework. For instance a BundleContext is used by a bundle to register services. The list of the further operations available on a BundleContext can be found in [7].

4.4.2 The Virtual Layer Bundle

The Cirrostratus Virtual Layer is itself also a bundle. This bundle is installed at each location on the local OSGi Framework. Every other bundle is installed on the top of the Virtual Layer Bundle. When started, the Virtual Layer Bundle initializes the elements responsible of making the local node part of the global Cirrostratus Platform. For instance either a new Cirrostratus DHT is created or an existing Cirrostratus DHT is joined.
Cirrostratus is able to interfere between the bundles installed atop of the Virtual Layer Bundles and OSGi Frameworks by overriding their BundleContext during installation. Each time a bundle wants to perform some operation on its BundleContext, he is not calling a local OSGi Framework's BundleContext but a custom Cirrostratus BundleContext. This custom BundleContext dispatches the operations to give them a global semantic. For each case, it has to be decided which actions are performed on the local OSGi Framework and which actions need to be undertaken on other remote OSGi Framework as well. The involved logic for services and bundles was previously presented in section 4.2 and 4.3. Events are described in 4.4.2.
The Figure 4.13 illustrates which actual components where selected for the implementation. The bold lines show that each bundle installed on the Cirrostratus

Platform has to communicate through the Virtual Layer Bundles. Virtual Layer Bundles forwards some operations to the local OSGi Framework or communicates with other Virtual Layer Bundles depending on the requested operation. Other bundles can be installed on the local OSGi Frameworks. However those bundles cannot communicate with the bundles installed on the top of the Cirrostratus Platform.

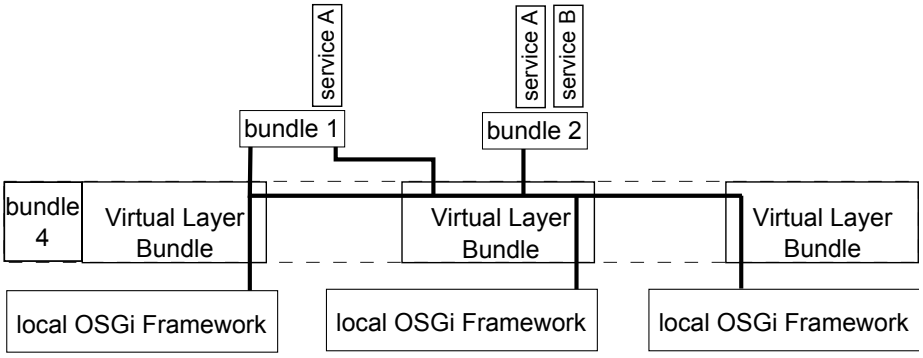


Figure 4.13: Arrangement of bundles, Virtual Layer Bundles and local OSGi Frameworks. Bundles don't communicate directly with the local OSGi Frameworks but only through the Virtual Layer Bundles.

As justified in 3.3, at some point in time the Virtual Layer needs to offer the possibility to migrate or replicate bundles and services. The involved mechanisms are described after the following paragraph.

Events

OSGi Events OSGi specifies an event mechanism for the life cycle of bundles and of the framework, as well as for the service layer. When a bundle moves from one state to another a corresponding event is triggered. The same is true for some framework properties. Service events report the registration, unregistration and changes of services. In every case listeners can be defined to catch specific events.

Cirrostratus Events In the Cirrostratus Platform events need to be propagated to the different distributed OSGi Framework. A simple strategy was selected: Each event is broadcasted to every node in the system. Then every Virtual Layer Bundle selects the events which are locally relevant. The efficient broadcast algorithm used for that purpose is presented in 4.1.3.

4.4.3 Migration & Replication

In order to enable the initiation of a replication or migration process, the Cirrostratus Virtual Layer bundle registers a custom service, called a Replica Manager, on each local OSGi framework. The interface of this service is described below.

- `public void replicate(ServiceReference sref, InetAddress node, int port);`
- `public void replicate(ServiceReference sref, InetAddress node, int port);`
- `public Set getAllLocalServices();`
- `public boolean changeProvider(Long serviceGlobalID, InetAddress node, int port);`

This service interface can be used by any Cirrostratus external bundle to manage the locations of the Cirrostratus Service replicas. Moreover, the link between the Proxy Services and their corresponding Cirrostratus Services can be set thanks to the method `changeProvider`.
The Cirrostratus Platform does not itself enforce any replication strategy. In this way any custom and extendable replication's strategies can be implemented. Even though the replication's strategy might differ from case to case, this work suggests the elaboration of generic strategies in future works 7.1.1.
The Figure 4.14 illustrates a typical external management's infrastructure consuming the Replica Manager service exposed by Virtual Layer Bundle to manage the services' replicas.

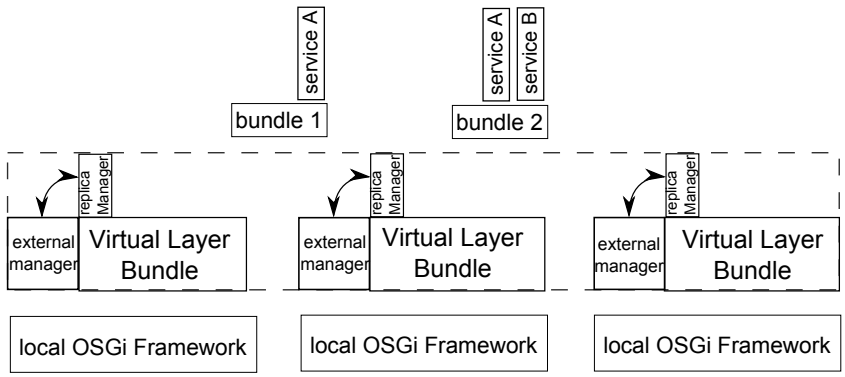


Figure 4.14: External Replica Manager consuming service exposed by the Virtual Layer Bundle

Service Replication

Service replication implies on one hand the actual object or thread migration, on the other hand the maintenance of the service and service reference metainformation. The first part is based on a previous work [22]. The main idea consists of injecting, at load time, a consistency protocol into the bytecode of the application classes, without the need of changing the application. The Virtual Layer Bundle then takes care of linking the injected logic with the local node and to handle this additional logic.

Metadata Layer When a service has to be replicated or migrated, the Cirrostratus platform has to undertake certain actions to guarantee consistent metainformation about that service.
First of all the bundle which registered the service needs to be replicated. When this task is done the node initiating the replication or migration process will serialize the corresponding Service Reference and send it together with the actual service to the recipient node. During the serialization of the Service Reference most of the ServiceReference fields are omitted. For instance the link to the local Framework, to the registering bundle and to the actual service are not send as part of the Service Reference. Those data are restored on the receiving side. Moreover the recipient node registers the replica locally. In the case of a migration, the original service needs to be unregistered from the initiating node. Finally the Service and Service Reference Registries need to be updated to reflect the location's modification.

Bundle Replication

A necessary condition for some service replication is the presence of the bundle which registered the service on the destination node. In fact the recipient node has to know about the service code before the service starts running. For that reason the registering bundle needs to be replicated prior to the service.

Metadata Layer The node initiating the replication tells the recipient node to install the bundle and waits until the installation is successful. For his part, the recipient nodes needs to make sure it has installed a corresponding Primary bundle, like defined in 4.2.2, before acknowledging a successful installation. To that end it first checks if it already holds the same bundle. In other words, it controls if the bundle is already installed as a Primary bundle or as a Secondary bundle. In the first case the node is done and the service replication can start. In the second case the node can reuse the host bundle but it needs to obtain the Primary bundle's metainformation. The metainformation is acquired from another Primary bundle. Concretely the Primary bundle is serialized except some fields related to the local environment. On the receiving side, the missing fields, related the local environment, are restored. If none corresponding Primary or Secondary bundle was found on the node, a copy of the host bundle is first retrieved, and only then the metainformation is duplicated as described.
The actual host bundle transfer is just a byte array transfer, requiring no special attention.

Chapter 5

Related Work

The idea of a distributed computing platform is not a brand new invention. *Toward Internet Distributed Computing* [23] is a previous attempt to describe an Internet wide computing platform. The Cirrostratus Platform shares with this proposal its two key design principles: embedding intelligence in the network and creating self-configuring, self-organizing network structures. While their work focus more on the conceptual level, their prototype is a combination of heterogeneous technologies. The Cirrostratus Platform follows the same conceptual path but in addition has the advantage to be the implementation of an established standard.

There has been multiple tentative to perform distributed computing on ad-hoc networks. *Smart Messages* [20], distributed programs executing on nodes of interest, are one of those. Even though the Cirrostratus Platform like Smart Messages (SM) also offers execution migration, it fundamentally differs in the assumptions. The Cirrostratus Platform assumes an All-to-All communication model while SM insists on the importance of self-routing. Looking at the evolution of small mobile devices, an All-to-All communication model seems to be a reasonable assumption.

Techniques implemented for execution replication in the Cirrostratus Platform are similar to the one presented in TerraCotta [10] [22]. TerraCotta is a clustering software for Java applications. Both systems instrument the code in order to transparently ensure consistency of distributed execution. However they differ strongly in the architecture. TerraCotta is a client/server architecture with a single point of failure. The Cirrostratus Platform execution replication mechanism avoids this weakness [22].

Distributed Java Virtual Machine (DJVM) also make use of thread migration for cluster wide parallel computing. For instance Jessica2 [37] is a DJVM running in JIT compilation mode allowing transparent multi-threaded Java applications to run on cluster. Besides migration technique Jessica2 implements a Global Object Space(GOS). In contrast to the Cirrostratus Platform this solution is not P2P oriented. Therefore the failure model is different. Furthermore the Distributed Service Registry (DSR) offered by the Cirrostratus Platform gives more flexibility than an GOS. In fact collaboration between applications becomes possible.

A first concept study was developed under the name *The Virtual OSGi Framework* [24]. However this work was largely based on ad-hoc solutions whereas

Cirrostratus comes out of a coherent design. Moreover the Cirrostratus Platform goes much further in terms of actual realization.

Chapter 6

Evaluation

To evaluate the Cirrostratus Platform, an existing application was adapted to fulfill the OSGi R4.1 specifications. Therefore this adapted application can run on the Cirrostratus Platform. In this chapter the original and the adapted application, the gathered results and the lessons learned are presented.

6.1 Stendhal

Stendhal [9] is a fully fledged free open source multiplayer online game, also called a Massively Multiplayer Online Role-Playing Game (MMORPG). Stendhal was originally built on the top of a framework/engine called Marauroa. This engine was created for developing Internet based multiplayer games. It should provide a simple way of creating games on portable and robust server architecture.

6.1.1 Game

Presentation

Stendhal is an adventure game in which each player receives a customizable character. Characters can move around to explore towns, buildings, plains, caves and dungeons. Many interactions between characters are possible. For instance characters can discuss and attack other characters. Furthermore, they can kill evil monsters, harvest many useful instruments and save objects into a bag. Characters can also interact with non-player characters to eventually acquire tasks and quests. While progressing into the game, characters will develop and grow and with each new level up become stronger and better.
Figure 6.1 shows a screenshot of the Stendhal game where some characters are currently interacting.

Implementation

Marauroa is based on a design philosophy called *Action - Perception*. Each turn clients can ask the server to perform any action in their name. The server gathers all the actions, computes the state of the new world and then send a perception to the clients explaining what they currently perceive.

Figure 6.1: Screenshot of the Stendhal game

In reality, the Stendhal world is divided into zones. Zones are an abstraction inherited from Marauroa. Each of these zones are an unique part of the world and stores a set of objects. The perception of the world is in fact only computed at the level of the zone. When a player arrives at the border of a zone A and wants to continue in another zone B, he is removed from zone A and added to zone B with a complete different perception. Therefore the change from one zone to another zone is not transparent to the user. In Stendhal only player characters are allowed to move to other zones.
The game implementation consists of Marauroa project and a Stendhal project. Although the Stendhal part is much larger than the Marauroa part, it is an extension of the Marauroa part. Thus, the core classes of Marauroa are extended by Stendhal classes. The Marauroa and Stendhal projects are both divided into client code and server code. The client can be considered as a thick client since it is in charge of the lasting presentation work. The server provides a MySQL backend and uses a TCP transport channel to communicate with dozens of clients.

6.1.2 Advantages

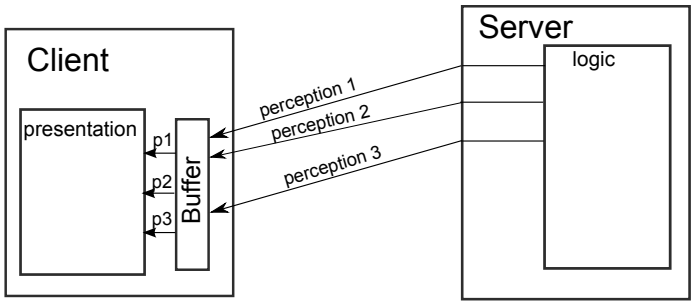
There were at least four main reasons that made Stendhal a good choice as a test application. Firstly, the legacy Stendhal game is a relevant comparison point in terms of performance since it is also used in a distributed way. For a user, playing the original or the modified game should not make any difference. This would, for instance, not have been the case if a multiplayer offline game would have been selected. Secondly the given separation of the game into different zones

is a chance to investigate how the Cirrostratus Platform can take advantage of locality. Since most of the information needs to be exchanged within a zone, a desired observation would be a higher amount of communication between nodes in the same zone than between nodes in different zones. Thirdly, this game is large enough for a realistic evaluation of the Cirrostratus performance aspects. Fourthly, a game is a fun application for participants if, in a future work, user-driven benchmarks are needed.

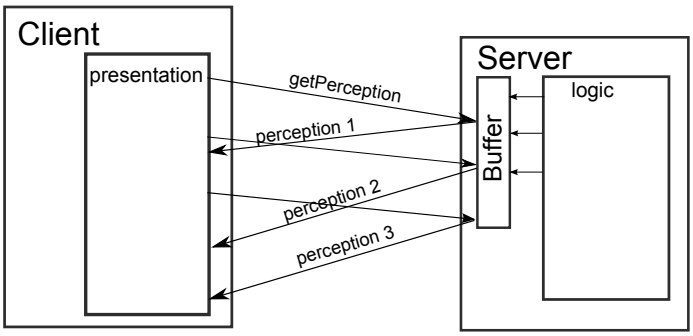
6.1.3 Adaptations

Thanks to the fact that the game was programmed in Java, most of the code could just be reused without adaptation. Most of the concepts were just left unchanged as well. For instance the adaptation keeps intact the strict separation between the Marauroa part and the Stendhal part. Likewise the separation between client code and server code was not fundamentally modified. Moreover the *Action - Perception* philosophy was preserved.
The real modifications concern state durability, the client and server's internal organizations and the interaction between the client part and the server part.
First the original database backend was removed. In Cirrostratus durability is achieved by replicating the services since it can't be assumed that a specific node is available. Hence, all the elements stored previously in the database are just registered as OSGi services and replicated. For instance it implies that objects representing characters or player accounts are registered under a newly created interface together with some properties. A property of a player is for example its username. In this way efficient player service retrieval is possible when the actual player wants to restart the game. The management of the service replicas 7.1.1 should ensure that enough replicas exist to prevent the lost services.
The original client and server part were built around a series of Singleton Patterns restricting the instantiation of some classes to one object. On the client side the Singleton Pattern were removed to allow the start of multiple clients within one JVM. Contrarily the Singletons on the server side were semantically relevant since for each server component only one instance should exist in the whole system. Therefore each server Singleton was transformed into an OSGi service implementing one unique interface. A modified Singleton access attempts to retrieve a server under the unique interface. If the retrieval is not successful, it is assumed that the service was never created and a new service is registered under the unique interface. At any time this mechanism ensures no multiple server components. Moreover, instead of exposing a monolithic service, the server is itself composed of multiple services. Although only one server *main interface* is registered, the different server internal services can potentially run on different nodes.
No service is registered by the client side. The client part only does a lookup for the server main interface. Then every communication between the server and the client happens through method invocation on this main interface. This is a principal difference to the original implementation were all the communication was done by customized message exchanges.
A fundamental change in terms of communication semantic is represented by the Figure 6.1.3. It shows the communication involved when the server part transmits the current perception of the game to some client. In the original case, the server, from time to time, transmitted the perception and the client

was buffering those perceptions until the next refresh of the presentation. Since every interaction in the adapted version is a method call, the client needs to ask the server part for the next perception. The server stores, for every client, the already computed perceptions into a data structure and sends provisioned data as the result of the method call. Even though the adapted approach requires



original Perceptions transfer



modified Perceptions transfer

a two-way communication, the method calls can also be used to manage a necessary list of alive clients. In the original case the clients had to ping the server from time to time to notify their presence.
The code was partitioned in two bundles: a client bundle and a server bundle. The client bundle imports some packages of the server bundle. Each client needs to starts the client bundle whereas the server bundle is started only once for the whole system. The server bundle can be started at any location.
In summary, the adapted game is composed out of multiple server-side services. Server side services can use other server services. A main server service is registered once in the beginning on the Cirrostratus Platform. Each client use this main service as connection to the server part. Every nodes is free to run a client and some server services.

Observations

As a general observation the most difficult adaptations were the ones dealing with concurrent aspects to the Cirrostratus Platform, namely communication and durability. Since the selected application was already dealing with distribution, it was first necessary to remove those concurring parts. Indeed distribution is an already embedded ability of Cirrostratus Platform. Hence, adapting to

the Cirrostratus Platform an application not designed to cope with distribution might surprisingly be less time consuming than adapting an already distributed application. Further research in this area is needed to confirm this hypothesis. Likewise, the same might be true for durability since all the explicit persistence features had be removed. Since every service is implicitly persistent in the Cirrostratus Platform, durability needs no special attention.
The easiest task of the adaptation was the separation of concerns since it was largely already given by the structure of the original application. For instance a separation between client and server code also makes sense in the Cirrostratus case, since in this way not all the participating nodes have to know about the server code. Such separation of concerns might be more difficult to identify in other applications.

6.2 Experiment

The evaluation of this work is functionality oriented. With an experiment the practicality of the Cirrostratus Platform and of the modularized Stendhal game is demonstrated.
During the experiment the following operations are performed.
First a node starts. Since it is the first in the network, it creates a new DHT. Other nodes joining later the DHT are given an entry node. The Stendhal server bundle is deployed on the node creating the DHT. However any other node could have been selected as well. When the Stendahl server bundle is started, the different services composing the server are registered. From this point the game can start. Some additional nodes are launched and start clients. The node on which the server is currently residing also holds a client. Every client will do a lookup for the main server service and proxy services will be created. Through proxies, clients will obtain new information about the game state and are able communicate any player actions to the server. Clients can leave the game at any point in time.
Figure 6.2 represents the setup of the experiment. Ten nodes are used. Every node has an intance of the Cirrostratus Platform running and will either install a Stendhal server bundle or a Stendhal client bundle on the top of it.
The experiment was realized on a single computer, a HP EliteBook 6930. Local sockets are opened for communication between the nodes. Figure 6.3 is a screenshot of the game running with ten players. Changes are equally visible for every client and the functionalities of the original game are preserved.

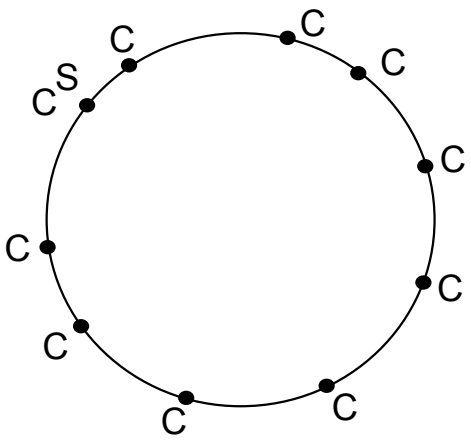


Figure 6.2: C = Stendhal Client, S = Stendhal Server

Figure 6.3: Stendhal game running on the Cirrostratus Platform with 10 players

Chapter 7

Conclusions

This thesis presented a platform giving the possibility to have a distributed, modular and dynamic architecture of Information Systems involving zero code adaptation. The final goal of this platform is to, over the time, maximally benefit from the advantages offered by each architecture. Three main pillars were identified as necessary elements of Cirrostratus Platform, namely: Distributed System, Modularity and Location transparency.
The distributed system requirement is based on the observation that information have inherently to be distributed. Therefore, nowadays, machines are widely used as entry point to some network and, in some way, they do take part to a network wide computing. Everyday the network is becoming more and more an entity for itself. For the Cirrostratus Platform a decentralized P2P architecture was selected since connectivity everywhere, implying large number of participants, and non-reliable nodes were assumed. A DHT implementation, namely the Chord protocol, was selected as a suitable solution for that purpose.
The modularity pillar is driven by the fact that a monolithic piece of software can hardly be meaningfully distributed. Besides other benefits like reusability and maintainability, modularity gives the possibility to distribute modules such that each node is only aware of a subpart of the system. On the other side, every module in the system is potentially available to every node if required. Moreover performance can be improved across the network by redistributing load. In the Cirrostratus Platform, modules can interact dynamically through a SOA. In this way the coupling between modules is reduced.
Location transparency provides the decoupling between the functions and the distributed architecture. To achieve dynamically any distributed architecture, modules should be able to move freely across the network. Therefore nothing should bind a module to a specific node. The access to a module should also be transparent since no one can guess where a module will be located in a dynamic and distributed architecture.
A prototype implementation is presented based on two well-established components: a DHT and the OSGi Framework. The DHT provides the necessary elements to cope with distribution. The OSGi Framework offers the needed modularity aspects. The implementation of the location transparency requirement is a custom solution and was the topic of this work and of a previous work [22].
From a realization perspective this work concentrated on implementing a scal-

able and reliable DHT, a virtual layer giving a global semantic to the distributed OSGi Frameworks and on modularizing an application for the evaluation of the Cirrostratus Platform. The DHT was extended with multiple features making it suitable for the Cirrostratus requirements. The modularized application is a multiplayer game and its design makes it specially suited for the evaluation of the Cirrostratus Platform.
The resulting constructed prototype should be a reliable basis for a complete solution.
I would like to conclude this thesis by a vision. A vision in which users can from every computer access transparently every data and every applications. Instead of having the same identical programs replicated on million of machines, only the currently needed elements are present on each computer. Programs don't need to be installed anymore. Like in a search engine, users just indicate the name of the application to run it. People don't have to constantly buy new more powerful hardware. Resources can easily be traded and differences of performances are blurred by the network. This is the vision of a flexible distributed world. The Cirrostratus Platform is the spirit of this vision and should help to make it real.

7.1 Future Work

7.1.1 Replication Strategies

To guarantee a total indepedence from a specific node, the failure of a node should not affect the availability of a module. For that purpose modules and services should be replicated. Future work may help to answer the questions when and where a replication is meaningful. Depending on the type of network, like for intance the Internet or Sensor Networks, and on different indicators, like for instance the type of devices or previous failure patterns, the strategies might need to be customized. A second aspect besides availability is performance. Placing replicas at strategic locations based on for instance the load of the network, the load of each node and its capacity, or other sensors is an interesting problem.

7.1.2 Performance Analysis

Due to time constraint this work did not evaluate some interesting aspects related to performance.

Modularity The influence of modularity on performance is. An interesting problematic is for instance to which extend modularity can reduce communication. On one hand too monolithic modules probably don't exploit the power of distribution. On the other hand too small modules maybe introduce unnecessary overhead. From this point it might be attractive to investigate the existence of optimal module sizes.

Scalability In the Cirrostratus Platform every participating node should bring some resources to the Platform. Therefore as the number of participant increases, the capacity of the system should increase as well. It would be attractive to show that if in average the available computing resource are sufficient at

each node, the overall system scales as well. This would prove that, in contrast to a rigid server-client architecture, an adaptive computing platform scales with the number of participants.

Heterogeneity If some nodes are more powerful than others, a desired feature would the possibility to distribute the load proportionally to the capacity of each device. For that purpose the semantic of a capacity has to be defined and concrete strategies have to be established. However the realization of this approach might be restricted by different factors. For instance a possible limitation could be the overhead introduced by the reallocation of modules. A future work might focus on showing the limits of automatic resources handling in the context of the Cirrostratus Platform.

7.1.3 Open Issues

Network Virtualization In fact the presented approach will not fulfill completely the location transparency goal if modules communicate directly through the network to the outside world. This is for example the case if a service opens Java Sockets. If this service gets migrated, an endpoint of the communication is changed and the communication is broken. One solution might be to extend the already existing bytecode instrumentation to handle any network communication. However no precise solution was designed yet.

Security This work has not discussed the issue of security. However, my personal opinion is that security has to be integrated part of the Cirrostratus Platform. In a real environment, some modules and services should not be accessible from everyone. Possibly the Cirrostratus Platform should allow the definition of security policies. Therefore security aspects need to be elaborated.

A taxonomy of tradeoffs Most importantly a future work should evaluate how the different parameters, like modularity or the amount of interactions, influence the global picture of performance. More precisely an overview could be constructed illustrating how the parameters impacts on each others. I view the Cirrostratus Platform as an excellent place to perform such experiments. In a later stage, a cost-benefit analysis could be performed on the basis of a real case scenario might. It this way the economics of the Cirrostratus Platform could be evaluated.

Bibliography

[1] Distributed hash table definition, 2009. www.answers.com.

[2] Distributed system definition, 2009. http://dictionary.die.net/.

[3] GSM Association, 2009. http://www.gsmworld.com/.

[4] Java servlet technology, 2009. http://java.sun.com/products/servlet/.

[5] Modular architecture definition, 2009. http://www.webopedia.com/TERM/M/modular_architecture.html.

[6] OSGi Alliance, 2009. http://www.osgi.org.

[7] OSGi Release 4 Documentation, 2009. http://www.osgi.org/javadoc/r4v41/.

[8] R-OSGi web page, 2009. http://r-osgi.sourceforge.net/.

[9] Stendhal Website, 2009. http://stendhal.game-host.org.

[10] Terracotta, 2009. http://www.terracotta.org.

[11] H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. Wiley-Interscience, 2004.

[12] N. Bartlett. A comparison of eclipse extensions and OSGi services, 2009. http://www.eclipsezone.com/articles/extensions-vs-services/.

[13] Bundesamt für Statistik (BFS). *Hochgeschwindigkeits-Internet, Internetnutzung*, 2008. http://www.bfs.admin.ch/.

[14] J. Dijk. *The network society*. Sage Publications London, 1999.

[15] S. El-Ansary, L. Alima, P. Brand, and S. Haridi. A Framework for Peer-To-Peer Lookup Services based on k-ary search. 2002.

[16] S. El-Ansary, L. Alima, P. Brand, and S. Haridi. Efficient Broadcast in Structured P2P Networks. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 304–314, 2003.

[17] A. Gustavo, F. Casati, H. Kuno, and V. Machiraju. Web services: concepts, architectures and applications, 2004.

[18] H. He. What is service-oriented architecture. *Publicação eletrônica em*, 30, 2003.

[19] Internet World Stats. *Internet Worldwide Usage*, 2008. http://www.internetworldstats.com/.

[20] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, and L. Iftode. Smart messages: A distributed computing platform for networks of embedded systems. *The Computer Journal*, 47(4):475–494, 2004.

[21] N. Liogkas, B. MacIntyre, E. Mynatt, Y. Smaragdakis, E. Tilevich, and S. Voida. Automatic partitioning for prototyping ubiquitous computing applications. *IEEE Pervasive Computing*, 3(3):40–47, 2004.

[22] D. Maragkos. Replication and migration of osgi bundles in the virtual osgi framework. Master's thesis, ETH Zürich, 2008.

[23] M. Milenkovic, S. Robinson, R. Knauerhase, D. Barkai, S. Garg, A. Tewari, T. Anderson, and M. Bowman. Toward internet distributed computing. *Computer*, 36(5):38–46, 2003.

[24] D. Papageorgiou. The virtual osgi framework. Master's thesis, ETH Zürich, 2008.

[25] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[26] J. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed applications through software modularization. *Lecture Notes in Computer Science*, 4834:1, 2007.

[27] A. Singh. An introduction to virtualization, 2009. http://www.kernelthread.com/publications/virtualization/.

[28] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM New York, NY, USA, 2001.

[29] Target Search. Distributed computing, loose coupling definitions, 2009. http://whatis.techtarget.com.

[30] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic java application partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, 2002.

[31] M. van Steen. Distributed systems principles and paradigms, 2008. http://www.cs.vu.nl/~steen/courses/ds-slides/notes.01.pdf.

[32] R. Wattenhofer. Principles of distributed computing - chapter 10, 2008. http://dcg.ethz.ch.

[33] W. Weck. Independently extensible component frameworks. *in Special Issues in Object-Oriented Programming*, 1997.

[34] C. A. Werner Vogels. Expanding the cloud, 2008. http://www.allthingsdistributed.com/2008/09/expanding_the_cloud.html.

[35] Wikipedia. Centralized computing, 2009. http://en.wikipedia.org/wiki/Centralized_computing.

[36] Wikipedia. John gage, 2009. http://en.wikipedia.org/wiki/John_Gage.

[37] W. Zhu, C. Wang, and F. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support.