FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Transparently Migrating Java Objects at Runtime in an Infrastructure-as-a-Service Cloud

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Fritz Schrogl, BSc
Matrikelnummer 0325746

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.-Prof. Mag. Dr. Schahram Dustdar
Mitwirkung: Mag. Dr. Philipp Leitner (Universität Zürich)

Wien, 11.04.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Fritz Schrogl, BSc
Lavaterstraße 7/2/5, 1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11.04.2014

(Unterschrift Verfasser)

Abstract

In the last couple of years Cloud Computing has become an important topic in industry and computer science. In particular, this applies to clouds following *Infrastructure-as-a-Service (IaaS)* paradigm, due to their maturity and high flexibility in terms of supported software stacks. Cloud Computing promises on-demand provisioning of IT resources, making easy scaling of applications during runtime possible and also relieves software developers of dealing with hardware, so that they can focus entirely on their software engineering. But there are also some downsides. To gain full advantage of Cloud Computing's scaling abilities developers have to write distributed software, which is a complex and error prone task. To minimize these hurdles *Distributed Systems Group (DSG)* developed jCloudScale, a Java-based middleware using *Aspect-oriented Programming (AOP)* and Bytecode manipulation to transparently deploy Java objects to common IaaS clouds.

This master's thesis describes and presents an extension to jCloudScale, that enables transparent migration of already deployed Java objects during runtime, while preserving their state and function. The migration mechanism further supports migration triggering using business rules and uses automated planning to choose an optimal migration strategy. First this thesis details the state of the art in the area of Cloud Computing, gives an overview of relevant related work and an introduction to jCloudScale's architecture. Afterwards the migration mechanism is described and evaluated in detail. Finally a conclusion about the work done and possible tasks for future development are given.

Kurzfassung

In den letzten Jahren wurde Cloud Computing zu einem wichtigen Thema in Industrie und Wissenschaft. Dies gilt im Speziellen für Clouds die dem *Infrastructure-as-a-Service (IaaS)* Paradigma folgen, aufgrund ihrer Ausgereiftheit und hohen Flexibilität in Bezug auf den unterstützten Softwarestack. Cloud Computing verspricht die sofortige zur Verfügungstellung von IT-Ressourcen, wodurch einfaches Skalieren von Applikationen während der Laufzeit möglich wird und außerdem Softwareentwickler vom managen von Hardware befreit werden. Dadurch können sich diese ganz der Softwareentwicklung widmen. Es gibt jedoch auch Nachteile. Damit Entwickler alle Vorteile der Skalierungsfähigkeit von Cloud Computing nutzen können müssen sie so genannte "Verteilte Software" schreiben, was kompliziert und fehleranfällig ist. Um diese Probleme zu minimieren hat die *Distributed Systems Group (DSG)* jCloudScale entwickelt, eine Java-basierte Middleware, welche *Aspect-oriented Programming (AOP)* und Bytecodemanipulation nutzt, um Java-Objekte transparent in IaaS Clouds zu verteilen.

Die vorliegende Diplomarbeit beschreibt und präsentiert eine Erweiterung von jCloudScale, welche die transparente Migration von bereits verteilten Java-Objekten zur Laufzeit ermöglicht und dabei deren Zustand und Funktion bewahrt. Dieser Migrationsmechanismus unterstützt weiters das Auslösen von Migrationen anhand von Geschäftsregeln und verwendet automatisiertes Planen, um eine optimale Migrationsstrategie auszuwählen. Die vorliegende Arbeit beschreibt zunächst den Stand der Technik im Bereich des Cloud Computings und gibt anschließend einen Überblick über relevante vergleichbare Forschung und jCloudScales Architektur. Danach werden der Migrationsmechanismus selbst und dessen Evaluierung im Detail beschrieben. Abschließend wird ein Fazit zur durchgeführten Arbeit gezogen und mögliche zukünftige Entwicklungsaufgaben vorgestellt.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1 Introduction

Since 1969, when the first four nodes of the ARPANET were deployed, the Internet and thus computer science as a whole have come a long way [64, 65]. In the beginnings network computing was complicated and expensive. Therefore only universities and governmental institutions participated, but this changed in the early nineties. In 1989 Tim Berners-Lee, a scientist working at CERN, composed a paper titled *"Information Management: A Proposal"* [13] describing a new approach on managing information using linked information systems and hypertext documents, nowadays better known as *World Wide Web (WWW)*. A year later he developed the first *What You See Is What You Get (WYSIWYG)* web browser with direct inline creation of links [101], making it possible to "surf" through hypertext documents across multiple hosts. In the same year the first commercial Internet service providers were founded, providing the general public the opportunity to connect to and use the Internet. From this time on the number of Internet hosts increased in logarithmic scale to an approximate total of 850 million hosts by July 2011. This tremendous growth was not limited to the number of computers connected to the Internet, but also the number of people using the Internet increased remarkable to an estimated total of about 2.1 billion users worldwide by June 2011. The growth of both values is illustrated in Figure 1.1.

As the number of Internet hosts and users increased the economical importance of the Internet increased as well. At first existing enterprises used the Internet to stay in touch with their customers by providing new and convenient services (e.g., electronic banking) to them. Later new business were started offering novel services to the users for free, covering their expenses by displaying advertisements to them. With the technology becoming more mature new business models evolved

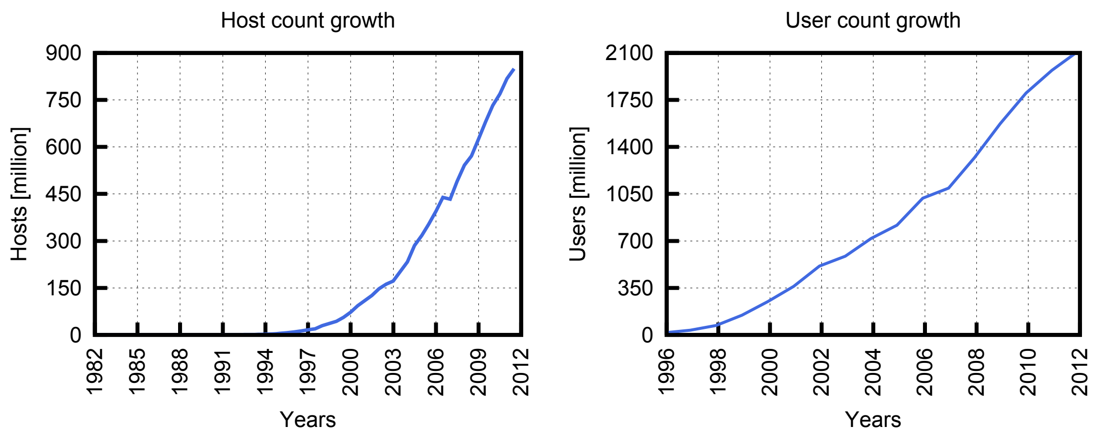

Figure 1.1: Internet host and user count history, data taken from [56] and [57]

as well and e-commerce became an important part of the newly emerged Internet economy. As stated in [10] the estimated revenues of the U.S. Internet economy in 1998 were about $300 billion with a total of 1.3 million attributed jobs. Due to these numbers it was not that hard for companies like Google, Facebook or Amazon, which were nearly unknown a decade ago, to collect enough venture capital to build up numerous data centers across the globe, each one large enough to house ten-thousands of servers [50, 53, 55]. The computational power these servers provide is needed by the aforementioned companies to offer their services fast and reliable to their millions of customers worldwide, regardless of time and date.

After the dot-com bubble burst in 2001 venture capital givers became much more conservative regarding the investments they made [95, 103]. An Internet start-up needed to have a solid business model with the promise to become profitable as fast as possible to convince investors. This resulted in the fact, that many start-ups with completely new and unusual business models could not obtain enough money to expand their computing infrastructure to met their customers demand or even worse to start their business at all. At the same time established companies with big data centers sought for possibilities to utilize their infrastructure more efficiently, so they came up with the idea of renting unused computing resources to other businesses. The term "Cloud Computing" [5, 6, 75] was born. With Cloud Computing there is barely the need for a start-up to buy its own infrastructure, which reduces the upfront costs to a minimum, making it possible to start new businesses without big amounts of venture capital.

Jinesh Varia from Amazon puts it like this: “[...] In the past, if you got famous and your systems or your infrastructure did not scale you became a victim of your own success. Conversely, if you invested heavily and did not get famous, you became a victim of your failure. [...]” [96]. But the benefits of Cloud Computing are not limited to start-ups, also well established companies can use it to operate their services more cost-efficient, because IT costs can then be seen as expenses instead of investments [6]. This is also valid for Cloud Computing providers themselves. Many of them have provisioned their IT resources to sustain occasional spikes in demand, but using only a few percent of their capacity at any one time. For instance Amazon, the world’s largest online store, uses only 10% of its IT resources on average, as stated in [1]. This whole situation made Cloud Computing an important topic in nowadays IT industry and computer science research.

1.1 Motivation

There are three different types of Cloud Computing, *Infrastructure-as-a-Service (IaaS)*, *Platform-as-a-Service (PaaS)* and *Software-as-a-Service (SaaS)* [45, 70], which will be discussed in detail in Chapter 2. To date IaaS provides customers the greatest flexibility in terms of the software stack they can use to implement their service(s). IaaS achieves this great flexibility by providing only relatively low-level services (*Virtual Machines (VMs)* and distributed storage in general) to the developer, burying less to nothing of the complexity of the underlying distributed system to her. Thus software developers have to deal on their own with all the complex details of developing and running a distributed application inside an IaaS cloud. They have to write software capable of running simultaneously on multiple hosts, create a *Virtual Disk Image (VDI)*, deploy it to the cloud and start as much host instances necessary to suit their application’s needs. For (cost-)efficient operation they will also have to implement a sophisticated scaling mechanism to match the amount of provisioned IT resources with actual demand.

To mitigate these hurdles the *Distributed Systems Group (DSG)* at Vienna University of Technology has developed jCloudScale [69], a novel Java-based middleware for *Amazon Web Services (AWS)*- and OpenStack-based [34,49] IaaS clouds. With jCloudScale an application developer just has to specify scaling requirements and policies of her application using code annotations. jCloudScale will then transparently deploy and scale the application to an IaaS cloud using *Aspect-oriented Programming (AOP)* and Bytecode manipulation. An in-depth description of jCloudScale will be given in Chapter 4.

Currently jCloudScale is in an early development state and thus has some functional limitations. One of them concerns its inability of migrating an already deployed *Cloud Object (CO)* to another *Cloud Host (CH)* during the CO's life cycle. This hinders jCloudScale from effectively using the computing resources of an IaaS cloud it is deployed to. During runtime of an application deployed through jCloudScale many COs are created and destroyed again. Therefore it is most likely that their distribution among the CHs used will be very biased. So it is possible that the system load of some CHs is be pretty low, while the load of others is so high that these hosts reduce the overall performance of the distributed application. It is also possible that all CHs are close to their idle state but have COs deployed to them and thus cannot be shut down by jCloudScale. Hence more cloud resources than necessary will be used, resulting in increased costs for the cloud user.

A fully transparent migration mechanism for COs which monitors system states of CHs and decides autonomous which CO to migration would resolve these problems. Thus the goal of this thesis is to design and implement such an mechanism for jCloudScale, making it an even more compelling tool for IaaS cloud application development.

1.2 Contribution

Before examining the jCloudScale framework in detail this thesis will give a definition and overview of Cloud Computing, its benefits and challenges. Also the *Message Queue (MQ)* paradigm and AOP will be discussed, as booth techniques are used by jCloudScale.

The main goal of this thesis is to address jCloudScale's inability to migrate COs from one CH to another after they have been created. Therefore in course of this thesis an appropriate migration mechanism for COs will be developed and implemented into jCloudScale. The migration mechanism must be fully transparent for objects interacting with COs going to be migrated and thus must preserve a CO's state during its migration.

Besides the bare migration functionality jCloudScale will also be extended to support some kind of migration policies. A migration policy is used to determine when to and where to migrate a CO. Such policies will have to consider several key figures of the distributed application and the cloud infrastructure (both managed by jCloudScale) to work efficiently. Hence reasonable key figures have to be identified and an additional mechanism to measure these figures without seriously lowering jCloudScale's performance has to be developed.

To achieve the goals mentioned above various software components have to interplay which each other. Because jCloudScale is a distributed software system these components will be spread above multiple hosts. This makes the use of distributed locking techniques a must, to avoid data corruption and ensure the software's stability. Good design of the migration process will be necessary to keep the performance penalty on jCloudScale, caused by these locking modes, as low as possible.

Finally the achieved work will be evaluated, using DSG's OpenStack [34] cloud. OpenStack is an IaaS cloud, fully compatible with AWS, which will make the evaluation results also applicable to AWS. The evaluation itself will be done to prove faultless working of the implemented migration feature and compare jCloudScale's scaling-capabilities with and without migration enabled.

1.3 Organization of this Thesis

The structure of the remainder of this thesis will be outlined as follows:

- Chapter 2 will give an overview of Cloud Computing, by discussing its different types and deployment models used today. Also common benefits and concerns of Cloud Computing will be addressed. Then *Amazon Web Services (AWS)* will be described, as it is currently the dominating IaaS cloud provider, making it very likely the cloud platform of choice when using jCloudScale. After that an introduction to the MQ [73, 91] paradigm and AOP will be given, because jCloudScale uses these technologies for communication and object invocation purposes.
- In Chapter 3 relevant related work will be presented and where applicable compared with jCloudScale. This related work will include *Enterprise Java Beans (EJBs)* [87], an industry-proven *Java Enterprise Edition (Java EE)* technology, as well as more academic frameworks like Cafe [78] and Aneka [98].
- The main goal of this thesis is to extend jCloudScale with a transparent migration mechanism for COs, therefore Chapter 4 describes jCloudScale's design and features in detail.
- Chapter 5 will focus on the design and implementation details of the migration mechanism added to jCloudScale. Each step of the implementation work will be shown and decisions made throughout this process will be discussed.

- In Chapter 6 the now extended jCloudScale middleware will be evaluated. Therefore appropriate use cases will be introduced to test and document jCloudScale's ability to efficiently utilize the underlying cloud infrastructure, both with migration mechanism enabled and disabled.
- Chapter 7 concludes this thesis by recapitulating the changes made to the jCloudScale middleware and the results from Chapter 6. Some final remarks and an overview of possible future work will be given too.

CHAPTER 2

State of the Art Review

The following chapter will give an comprehensive outline of state-the-art technologies used by jCloudScale and thus build the basis for the subject of this thesis. At first the fundamentals of Cloud Computing, its benefits and challenges will be covered. Then *Amazon Web Services (AWS)* will be discussed, as it will be the most likely choice when choosing a cloud platform to run jCloudScale applications. The end of this chapter will be devoted to the *Message Queue (MQ)* paradigm and *Aspect-oriented Programming (AOP)*.

2.1 Cloud Computing

In Chapter 1 it was mentioned, that jCloudScale is a middleware for building transparently scaling applications for IaaS clouds, but a detailed explanation of the term “cloud”, and therefore Cloud Computing, was missing. This issue will be addressed now, by giving a definition of Cloud Computing, describing its taxonomy and discussing its benefits and challenges for providers and users.

Once Larry Ellison, CEO and founder of Oracle, said about the IT industry: “[...] The computer industry is the only industry that is more fashion-driven than women’s fashion. [...]” [27], which is true to some extent. Technologies and their accompanied terms appear all the time and only few of them remain for more than a couple of years. This makes it hard for enterprises and customers to figure out if and when they should move to a new technology or better stick with their current infrastructure. Because of that advisory firm Gartner publishes its “Hype Cycle” [28], starting in 1995, to assist CEOs and CIOs with their decision making. Gartner’s hype cycle is a graphic representation of the maturity and adoption of new technologies. It consists of five phases ("Technology Trigger", "Peak of Inflated Expectations", "Trough of Disillusionment", Slope of Enlightenment“, ”Plateau of Productivity“) each technology must go through to become adopted sustainable by the market. Cloud Computing appeared first in Gartner's 2008 hype cycle, located in the ”Technology Trigger“ phase. As seen in Figure 2.1 Cloud Computing is still present in the 2013 hype cycle, now in phase 3 with an estimation of 2 to 5 years till mainstream adoption.

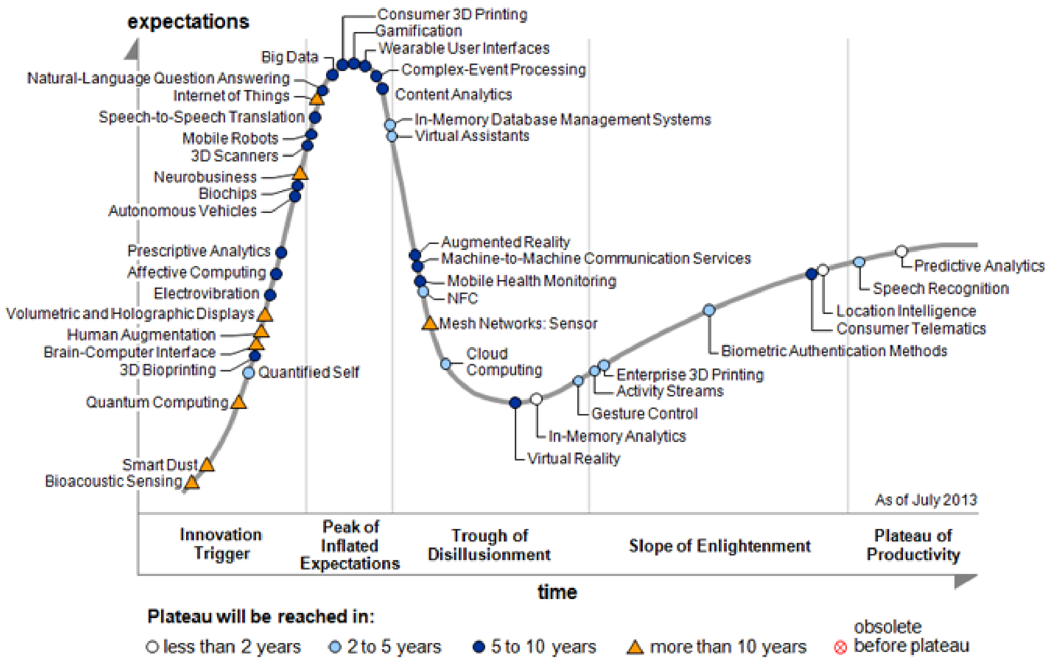

Figure 2.1: Hype Cycle for Emerging Technologies 2013, from Gartner [41]

The progress of Cloud Computing on the hype cycle's path during the last 5 years and the tangible estimation till mainstream adoption make it very likely that Cloud Computing will be a sustainable and important technology for the computer industry. This is also underpinned by data from IDC, saying that "[...] worldwide revenue from public IT cloud services exceeded $16 billion in 2009 and is forecast to reach $55 billion in 2014, representing a compound annual growth rate of 27.4%. [...]" [21]. Beside economical predictions people also believe in the success of Cloud Computing as revealed by a survey of the Pew Research Center's Internet & American Life Project and Elon University's Imagining the Internet Center, were about 71% of the survey's participants agreed with the opinion, that by 2020 most people will do their work in Internet-based applications instead of applications running on a PC operating system and that most developers will program for companies that provide cloud-based applications, because the most innovative work will be done in that domain [3].

2.1.1 Definition of Cloud Computing

It is a necessary but also a hard task to give a definition of Cloud Computing. The reason for this can be explained by a quote of Andy Isherwood:

> A lot of people are jumping on the bandwagon of cloud, but I have not heard two people say the same thing about it. There are multiple definitions out there of 'the cloud'.
>
> — Andy Isherwood, HP VP Software Services (EMEA) [8]

Nevertheless there recently has been a growing number of Cloud Computing-related publications referring to "The NIST Definition of Cloud Computing" [75] to accomplish this task. The *National Institute of Standards and Technology (NIST)* is an US federal agency, responsible for developing standards and guidelines for other federal agencies. Vivek Kundra, CIO of the US government, estimates in [66], that combined spending of federal agencies on Cloud Computing could reach up to $20 billion and thus let them become important customers for cloud providers. Because of this it is likely that the aforementioned NIST paper will become a de-facto standard when defining Cloud Computing and is therefore refereed in this thesis too.

In NIST's paper the Cloud Computing model is composed of five characteristics, three service models and four deployment models. Because the author of this thesis thinks, that service- and deployment models are not part of an essential definition of Cloud Computing they are omitted here and discussed later on (see 2.1.3). According to NIST the cloud model is defined by the following five characteristics:

- *On-demand self-service:* A cloud must give it's customer the ability to provision computing capabilities as needed, without the need of approval or interaction with one of the cloud provider's employees.
- *Broad network access:* The cloud is reachable through the network, using standard mechanisms, regardless of the access technology used by a user (e.g., mobile devices, thin or thick client platforms).

- *Resource pooling:* A cloud provider utilizes it's computer resources in a way that multiple customers can be served at the same time. These resources are assigned and reassigned dynamically the meet each customer's actual demand. To achieve this cloud providers use computer virtualization technology (see Section 2.1.2) to separate computing resources from actual hardware. Hence customers have in general no knowledge nor control over the exact location of the resources provided to them.

- *Rapid elasticity:* The computing capabilities a cloud provides to it's customer must be elastically and rapidly scalable. Meaning that the customer's demand has to be fulfilled nearly immediately, regardless of requested quantity and time of request. This can be done automatically or manually by the customer herself. For the customer the resources available for provisioning often appear to be unlimited.

- *Measured service:* Pay-per-use is an often used payment model in Cloud Computing. Therefore a cloud provides appropriate monitoring services for customers and providers to display each customer's resource consumption. There can be different levels of abstraction used by the monitoring services to best fit each resource type (e.g., storage, processing, bandwidth, etc.).

Given these characteristics the distinction between Cloud Computing and Grid Computing [29] is often not clear, but both paradigms provide quite a different experience for the user. The aim of Cloud Computing is to deliver each user his/her own operating environment, isolated from the environments of other users of the cloud. On the contrary Grid Computing tries to present a uniform resource pool to it's users, so all users face the same operating environment. Figure 2.2 illustrates both paradigms. A more in-depth comparison of Grid and Cloud Computing is given in [4].

2.1.2 Virtualization for Cloud Computing

As listed in the previous section one characteristic of Cloud Computing is resource pooling. Besides convenient on-demand provisioning provided to the users, efficient resource utilization is also crucial for cloud platform providers for operating their infrastructure profitable. To achieve this efficiency it is necessary to separate computational resources from actual hardware resources, by creating *Virtual Execution Environments (VEEs)*. VEEs are "[...] fully isolated runtime environments that abstract away the physical characteristics of the resource and enable sharing [...]", as stated in [94]. To enable VEEs virtualization technologies are used.

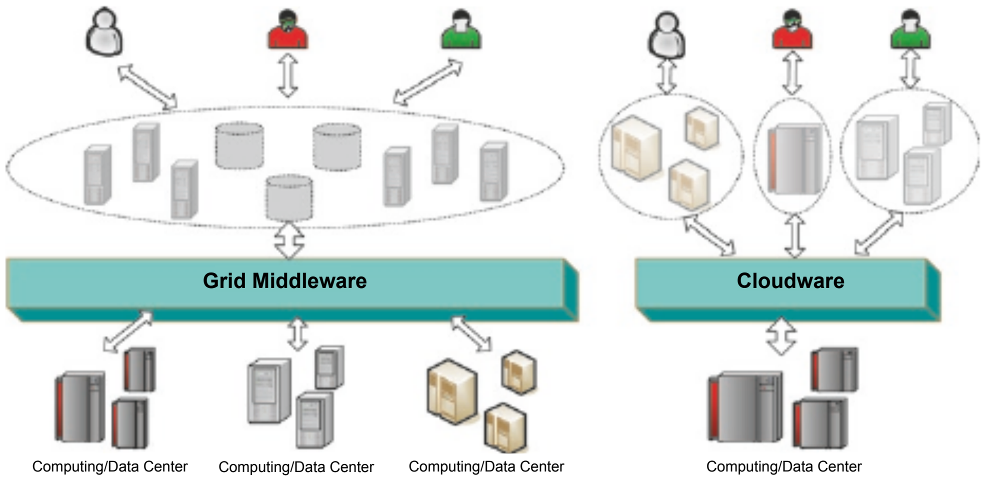

Figure 2.2: Comparison: Grid and Cloud Computing, taken from [4]

There are various computer resources that can be virtualized, like storage, network, memory and hardware. Especially for IaaS-based clouds virtualization of hardware, also called system virtualization [84], is most important. With system virtualization it is possible to run multiple heterogeneous operating systems on the same physical computer simultaneously [94]. The *Operating Systems (OSes)*, and thus the applications running on them, are isolated from each other, so they can not interfere. A thin software layer, called *Virtual Machine Monitor (VMM)* or hypervisor, performs this isolation by creating a virtual environment for each OS running on the physical machine. The physical server is also called "host" and the OSes, executed in their virtual environments, "guests" or *Virtual Machines (VMs)*. Because the virtual environment a VM is executed in is completely abstracted from actual hardware it is possible to move VMs between physical hosts, even while they are executed. This so called "live migration" [19] also helps cloud platform providers to achieve efficient utilization of their infrastructure.

There are two types of VMMs in use. The first one is the so called "hardware-level VMM" and has been around since the 1960s [86]. A hardware-level VMM runs directly on top of the physical machine and maintains access of the Virtual Machines to the actual hardware. A schematic illustration of such an VMM is shown on the left side of Figure 2.3. The second VMM-type is the "OS-level VMM". It was introduced in the late 1990s by VMWare and is installed on top of an existing OS. The goal of an OS-level VMM is to extend the OS, it is installed on, with VMM-capabilities, so that it can run VMs on its own [86]. The illustration of an OS-level VMM is shown on the right side of Figure 2.3. Due to there different approaches both VMM types target different use cases. A hardware-level VMM focuses on efficiency by minimizing the overhead it introduces, whereas an OS-level VMM focuses on ease of use by mostly providing a convenient *User Interface (UI)* to the user. Thus hardware-level VMMs are often used on server systems, while OS-level VMMs are in general used in desktop computing. An example for the former is XEN [7,104], which is also used by AWS to virtualize their hardware and provide VEEs for their customers (see Section 2.2), and an example for the latter is VirtualBox [81,102].

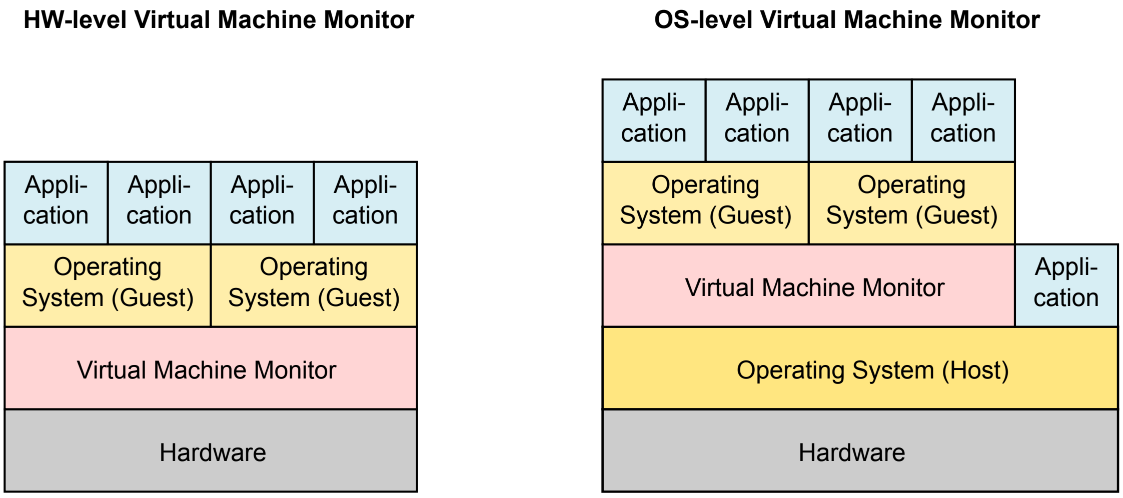

Figure 2.3: Hardware-level VMM vs. OS-level VMM

Apart from the type of VMM used in system virtualization there is also another distinction concerning the ”completeness“ of the virtualization achieved:

- *Full Virtualization:* Full Virtualization provides the highest degree of virtualization that can be achieved. A guest OS cannot determine if it is running on physical hardware or inside a virtual environment provided by a VMM. Hence every OS can be executed without the need to modify it. [4]
- *Para-Virtualization:* In Para-Virtualization a guest OS is aware that it is not running on actual hardware and thus has to be modified to be able to run on the VMM. Communication between the guest OS and the VMM is done through a special *Application Programming Interface (API)*, called *Virtual Machine Interface (VMI)*. [7]

Both, Full and Para-Virtualization, have advantages and disadvantages over each other. Full Virtualization prevails in flexibility, because it does not require modification of the guest OS kernel. This can especially be an issue when running proprietary Operating Systems, where source code is not available publicly. Para-Virtualization on the other hand prevails when comparing execution performance of the guest OSes. Because guest OSes are aware of the VMM and thus communication is done through the VMI instead of binary code translation or binary code rewriting performance of guest OSes are close to native execution [4]. Figure 2.4 shows the schematic differences between Full and Para-Virtualization.

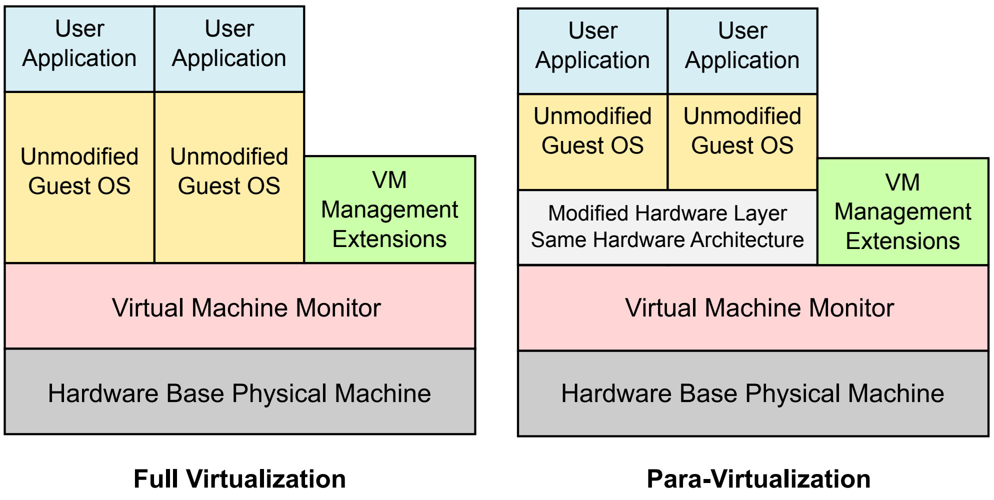

Figure 2.4: Para-Virtualization vs. Full Virtualization, redrawn from [4]

Until recently the nowadays so important x86 architecture did not support Full Virtualization in hardware, hence all necessary functions had to be implemented in software into the VMM. This changed in 2005, when Intel and AMD introduced their virtualization technologies. Surprisingly hardware-supported Full Virtualization was not significantly faster then software-only Full Virtualization, as revealed in [2]. As the authors state this may change over time, when hardware support for virtualization enhances and lead to another boost in virtualization and thus Cloud Computing infrastructure technique.

2.1.3 Taxonomy of Cloud Computing

After previous section briefly digressed into virtualization core Cloud Computing topics will be taken up again, by giving an overview of its taxonomy used today. The previously cited ”NIST Definition on Cloud Computing“ [75] states that a cloud model is composed of five essential characteristics, three service models and four deployment models. The characteristics were discussed in Section 2.1.1, service and deployment models will be explained in the following.

A well-known quote by Butler Lampson is: “[...] All problems in computer science can be solved by another level of indirection [...]” [82]. It describes the common practice in computer science to use multiple layers of abstraction to hide a system’s complexity from the user and providing a homogeneous environment to her. It is obvious that abstraction layers are also used in Cloud Computing. A service model categorizes a cloud offering by the number of abstraction layers it reveals to the user, hence the user has to manage by herself when using the cloud. The three service models, defined by NIST [75], are called *Infrastructure-as-a-Service (IaaS)*, *Platform-as-a-Service (PaaS)* and *Software-as-a-Service (SaaS)*. IaaS provides the lowest level of abstraction, and SaaS the highest level of abstraction to the user:

- *Infrastructure-as-a-Service (IaaS):* IaaS provisions basic computing resources, like processing, storage, networking and others, to the customer. The cloud provider manages basic infrastructure, like hardware and the virtualization technique used, but starting at the operating system level (inside the VM) everything can be controlled by the user. Compared to both other service models, IaaS has the disadvantage to leave most part of the administrative work to the user. On the other hand this is also an advantage, because the user keeps full control over the software stack she wants to use.

  The ”pay as you go“-model is considered to be the best pricing model for IaaS-based clouds [72]. The customer is charged by the amount of resources (e.g., processing, storage, bandwidth) she consumed during a certain time span (e.g., month). Examples of known IaaS providers are Amazon (see Section 2.2) or Rackspace [85].

- *Platform-as-a-Service (PaaS):* PaaS adds more abstraction layers to a cloud and thus is easier to manage for a customer, compared to an IaaS cloud installation. The cloud provider manages basic infrastructure, VMs and the middleware installed on each VM. The middleware provides a common execution environment to the customer, letting the cloud appear as one single computer with nearly infinite performance. The user manages deployment of applications to the cloud and settings for the application-hosting environment [75].

  Yet PaaS clouds are not as mature as IaaS clouds, due to lack of standardized cloud application development, but this may change in the future [72]. Two examples for PaaS providers are Google App Engine [54] and Microsoft Azure [23].

- *Software-as-a-Service (SaaS):* SaaS provides the highest level of abstraction for the customer. Everything up to the actual application is managed by the cloud provider. The user can access the application through a web browser or a program interface [75]. SaaS takes away near to all administrative work for the user, but also provides the least flexibility to her.

  Examples for SaaS are Microsoft Office 365 [76], Google Maps API [40], or Salesforce's online-CRM solution [88].

A visual comparison of the three service models is given in Figure 2.5. The stacked rectangles represent layers of abstraction and their color indicates if they are managed by the user or the cloud provider. In traditional IT all layers have to be managed by the user herself, so even IaaS, which provides the lowest level of abstraction, takes some tedious work away from the user.

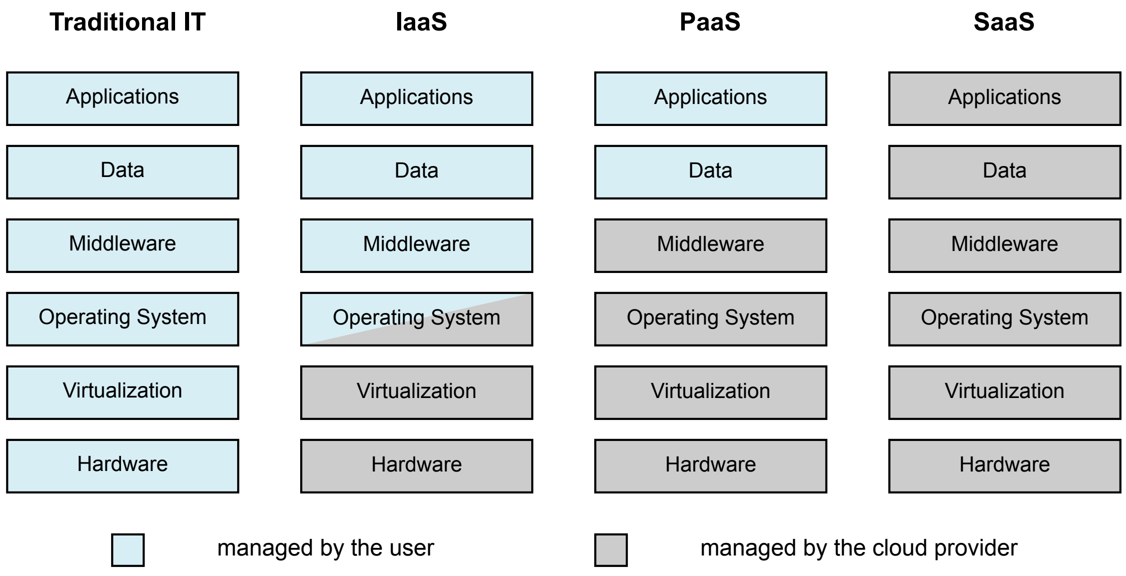

Figure 2.5: Comparison of IaaS, PaaS, and SaaS; inspired by [44]

The last part a cloud model is composed of is the deployment model. A deployment model categorizes a cloud offering by the kind of audience it is provisioned for. The authors of NIST's definition of Cloud Computing [75] define four types of deployment models:

- *Private cloud:* A private cloud infrastructure is exclusively provisioned for only one customer (organization or individual). Therefore the cloud may be owned and managed by the customer herself, but outsourcing to a third party is also possible. Depending on who operates the cloud it may be located on or off premises [75]. Private clouds are suitable for customers having special requirements regarding security, privacy or legal regulations, which could not be fulfilled by another deployment model [72].

- *Public cloud:* A public cloud infrastructure is the opposite of a private cloud. It is provisioned for multiple customers, who share the cloud infrastructure but operate their applications in separated virtual environments. The services hosted in a public cloud can be accessed by the general public, unless the customer configures to the contrary. The cloud is located on the premises of the cloud provider [75]. Public clouds are best suited for non-business critical applications, individual users or small businesses [72].
- *Community cloud:* The community cloud deployment model is similar to the public cloud model, but is only provisioned for the members of a specific community instead for the general public. These members can be individuals or organizations and share the same requirements regarding clouds (e.g., security or compliance considerations). The cloud infrastructure may be owned by community members or a third party and can be located on or off premises [75].
- *Hybrid cloud:* A hybrid cloud incorporates characteristics of at least two of the three other deployment models. The parts of a hybrid cloud representing other deployment models remain unique inside the hybrid cloud, but are bound together, to enable data and application portability between them. The technology used for bounding can be standardized or proprietary and a hybrid cloud can be located on or off premises [75].

2.1.4 Opportunities and Advantages

There has been an important growth in Cloud Computing services during the last couple of years and predictions claim that this growth will continue, even accelerate, in the following years [38, 83]. Besides usual market hype this is also caused by the many benefits Cloud Computing promises for organization's IT. The authors of [17] list 13 benefits of Cloud Computing and rank them by their number of occurrence in literature. Some of these benefits have already been briefly mentioned in this thesis. For better overview and understanding the most important benefits will now be discussed in detail:

- *Cost efficiency:* Cost reduction is one of the most addressed benefits of Cloud Computing [17]. Jinesh Varia writes in [96] that "[...] if you have to build a large-scale system it may cost a fortune to invest in real estate, hardware (racks, machines, routers, backup power supplies), hardware management (power management, cooling), and operations personnel. [...] Now, with utility-style computing, there is no fixed cost or startup cost. [...]" Instead of startup and fixed costs, there are only usage-based costs in pay-as-you-go Cloud Computing [5]. Further these running costs are in general lower, than *Small- and medium-sized Enterprises (SMEs)* or start-ups would have to pay when running their own data centers. This is due to the fact, that large data centers have better economies of scale (Supply-side savings, Demand-side aggregation, Multi-tenancy efficiency) than smaller ones, resulting in a significant lower *Total Cost of Ownership (TCO)* [44].

- *Scalability/Flexibility:* Another benefit of Cloud Computing is its elasticity. Popular Internet services often face large variations in demand, which can occur periodic (e.g., daytime or seasonal related) or arbitrary (e.g., mentioned in news articles or social media). To handle such spikes in demand one has to provision in advance for them, because changing the number of servers in a data center can take weeks [5]. This makes it impossible to scale provisioned computation resources flexible according to current demand. On the other hand with Cloud Computing scaling normally takes place in minutes [96]. Due to its elasticity Cloud Computing has also the potential for shrinking processing time. Now one can rent a huge number of machines to compute a task in parallel and speed up processing, with nearly the same costs as when renting only one machine for an accordingly longer time [44].

- *Better business focus:* Nowadays the use of IT is inevitable for a business, meaning that even if IT is not the core business of a company a significant amount of money has to be spent for it. As described in [44] 53% of a company's IT budget is currently spent on infrastructure and 36% is spent for maintaining existing applications. With Cloud Computing the spending in both areas can be reduced, freeing up money for new application development, giving the company an advantage in market competition. Furthermore Cloud Computing can reduce Time-to-Market for new services, giving an enterprise an additional advantage over its competitors [43].

That the benefits of Cloud Computing can also be put into effect in real world business is proven by numerous success stories. Some of these stories are enumerated in [45], another one is described in [96]. Two success stories not noted in literature previously cited are "Animoto's Facebook scale-up" [46,100] and "Pulse's Kindle Fire scale-up" [12].

Animoto is an online video editing service, originated in 2006. It allows users to easily create videos with their own photos and music by just uploading them to the service. To run this service Animoto uses various services offered by AWS, like *Amazon Elastic Compute Cloud (EC2)*. EC2 is a service, providing resizable compute capacity to customers (see Section 2.2). On average Animoto used 50 EC2 instances simultaneously to serve its 25,000 users, but in April 2008 they started a Facebook app resulting in an enormous popularity growth of their service. In within 3 days the number of users had grown by 10 times, to a total of 250,000. To keep up with the demand Animoto had to increase the number of used EC2 instances to more then 3,500.

Pulse is a young Internet start-up, founded in 2010. It develops news reading applications for mobile devices, like smartphones and tablet computers, which aggregate and enrich news feeds to provide a better user experience than usual news applications do. For pre-processing news feeds Pulse runs its backend software on *Google App Engine (GAE)*. When Amazon launched its Kindle Fire tablet in late 2011 they decided to preload Pulse on every device sold. Due to that Pulse could nearly double its user-base, to about 10 millions, in a very short period of time. The number of requests per second Pulse's backend had to handle reached 10,000 at peak time. Because Pule's backend software is deployed to a PaaS cloud it scaled-up to met demand, without much intervention needed by Pule's developers.

Both success stories show the benefits Cloud Computing can bring real world business. Without the use of cloud services both companies would not have been able to scale-up their services that much in such short periods of time, given their personnel and financial limitations, resulting in a severe loss of reputation and new users.

2.1.5 Concerns and Challenges

Whenever a new technology emerges its rise is also accompanied by numerous concerns. This is especially true for technologies like Cloud Computing, which are considered to have serious impact on a whole industry. Richard Stallman, a Free Software activist and known for uncompromising statements, expressed his concerns about Cloud Computing in an interview with *The Guardian* like this:

> It's stupidity. It's worse than stupidity: it's a marketing hype campaign. Somebody is saying this is inevitable – and whenever you hear

somebody saying that, it's very likely to be a set of businesses campaigning to make it true.

– Richard Stallman, GNU Founder [60]

Stallman wanted to address concerns about Cloud Computing regarding issues like e.g., privacy, security, reliability, portability and sovereignty of cloud customer data and software, which are indeed main concerns about Cloud Computing, as stated in [17] and [44]. Besides Stallman's rant there is lot of qualified literature discussing these issues like [58, 61, 62] or more comprehensively [18].

The five previously mentioned main concerns about Cloud Computing will now be discussed in detail:

1. *Security:* Data security is an important issue for enterprises, because for most companies their data represent a big business value. As [62] states, in Cloud Computing a user will typically not know the exact location of her data stored, nor will she know the sources of data from other users collectively stored with hers. If one of these other users conflicts with law, storage hardware can be confiscated, which may result in data loss for other users too [18]. Also clouds can become a tempting target for cybercrime, which can increase security risks for their users further. As mentioned in [43] encrypting data stored in the cloud does not solve security problems completely, because when this data should be processed in the cloud it has to be decrypted temporarily.

2. *Privacy:* With more personal data stored in the cloud privacy becomes an important topic in Cloud Computing. In many countries data protection laws are in effect, which may lead to civil or criminal sanctions for a company failing to comply with. The authors of [18] consider this to be a high risk for cloud consumers, because when they store data in the cloud they have to rely on the cloud provider to comply with these laws. Storing data in foreign countries or at cloud providers owned by US-based companies can also be a risk due to foreign jurisdiction. For instance the US Patriot Act grants US federal agencies access to data stored in data centers, even when they are located outside the United States but owned by US-based companies. This and further implications for Cloud Computing caused by the US Patriot Act are treated by law firm SNR Denton in [35] and german computer magazine "iX" in [15].

3. *Availability:* When an enterprise decides to shift toward the cloud by moving more and more of its IT infrastructure into the cloud availability and reliability of a cloud provider becomes crucial for the enterprise's business success. Due to this fact cloud providers offer in general *Service Level Agreements (SLAs)* to their customers. A SLA is a contract specifying various parameters the cloud provider's services have to fulfill, as described in detail in [61]. When a cloud provider cannot met its SLAs cloud customers can expect compensation payments for their loses. The problem with most Cloud Computing SLAs today is, that they are weak worded on purpose to protect cloud providers instead of customers, as stated by the authors of [43].

4. *Interoperability:* Another concern about nowadays Cloud Computing is the limited interoperability between the services of different cloud providers. Currently standardized interfaces shared among cloud providers, like common APIs or file formats for VM images are missing. This issue affects all three cloud types (IaaS, PaaS, SaaS), as described in [18]. This vendor lock-in limits customers ability to switch their cloud provider and thus is an disadvantage for them. On the other hand this is an advantage for cloud providers, because now customers cannot leave that easily when increasing prices or reducing reliability, as outlined in [5].

5. *Control/Governance:* Using a cloud provider's infrastructure to offer services to customers or employees implicates that a company gives up (some) control about how to manage used IT infrastructure. In [18] this is called ”Loss of Governance“ and assessed to be a high risk for cloud consumers. Examples for this loss in governance are ”Reputation Fate Sharing“ [5] and unpredictable service performance [45]. Reputation fate sharing means that numerous customers' reputation can be affected if only one cloud user behaves bad, e.g., by sending spam mails, so that multiple *Internet Protocol (IP)* addresses of the cloud become blacklisted by spam-prevention services. Unpredictable service performance occurs when a cloud provider poorly handles over-subscription of the physical computer hardware. In both cases end-users will blame the affected cloud user, despite the fact that the cloud provider or another cloud customer is responsible for the obstacles.

Some of the concerns treated above are not inevitable. According to the opinion of the authors of [4], there is no technical reason why a cloud service cannot be as secure as a self-hosted service. They also think that privacy concerns about Cloud Computing arise from the fact, that users are not used to the Cloud Computing paradigm yet. But when social habits change there might be no concerns about privacy any more. Reliability and availability concerns can be eliminated by diversification. The authors of [58] recommend the use of several non-related cloud providers to achieve superior availability of one's service(s). The vendor lock-in problem may be mitigated in future, as there are many standardization activities in progress. Examples for these activities are the *Cloud Security Alliance (CSA)* or the *Open Cloud Computing Interface Working Group (OCCI-WG)*, mentioned in [4]. There are also efforts to create open-source cloud stacks, comprehensively discussed in Chapter 6 of [11].

Concluding it can be said, that with Cloud Computing maturing the quality of cloud offerings and the benefits they provide users will rise and outweigh user's concerns.

2.2 Amazon Web Services (AWS)

Amazon Web Services is a product and also a subsidiary of e-commerce company Amazon. In 2006 AWS started to offer IT infrastructure services to the general public in form of Web Services accessible over *Hypertext Transfer Protocol (HTTP)* [31], using *Representational State Transfer (REST)* [30] and SOAP [42, 90] protocols [49]. Since then market adoption of AWS increased steady and by February 25th, 2009 about 490,000 developers had registered for AWS as mentioned by Jeff Barr in an AWS blog post [9].

By running one of the world's biggest e-commerce websites Amazon is compelled to maintain big IT facilities around the world to bring its services fast and reliable to its millions of customers. These services face seasonal peak periods in demand especially during the weeks before Christmas or Thanksgiving. To not hinder its own business during these periods of time Amazon has to provision its IT resources to cope with these peaks. As a result a large part of these resources remain unused for the rest of the year, causing costs but no revenues [11]. To change this Amazon started AWS. Services provided through AWS are billed in a pay-as-you-go manner with "[...] no up-front expenses or long-term commitments (necessary), making AWS a cost-effective way to deliver applications. [...]" as described by Amazon in [51].

Since its start in 2006 the number of services offered by AWS has increased continuously. Now AWS's offerings are not limited to IaaS solutions any more, also PaaS, SaaS and even *Human-as-a-Service (HaaS)* solutions are available. For instance AWS Elastic Beanstalk, currently in beta stage, is a PaaS solution for Java web applications. A developer just needs to package her application's code into a Java *Web Application Archive (WAR)* file, upload it to Elastic Beanstalk and everything else (deployment, scaling etc.) is managed by the service. *Amazon Relational Database Service (RDS)* is an example for a SaaS solution offered by AWS. With RDS a relational database can be managed and operated through a Web Service. Scaling, backup etc. is done automatically by RDS. HaaS describes a service requiring human interaction for operation. Amazon Mechanical Turk enables companies to establish such services. A company has to submit a task it needs to be done to the Mechanical Turk service, which then offers this task to a human workforce. When the task is done AWS sends the result back to the company requested the task. The human workforce is paid by AWS which bills the requesting companies per task accomplished.

The remainder of this section will only discuss three core services of AWS: *Amazon Elastic Compute Cloud (EC2)*, *Amazon Elastic Block Store (EBS)*, and *Amazon Simple Storage Service (S3)*. Similar services are also provided by OpenStack, which will be used to evaluate jCloudScale's migration mechanism, which design and implementation is the goal of this thesis (see Chapter 6).

An overview of AWS's services including also ones not covered in this section can be found in [51] and [52].

2.2.1 Amazon Elastic Compute Cloud (EC2)

EC2 is the very core of Amazon Web Services. It is not just a service used by AWS customers, but also by AWS itself to run other services like Amazon Elastic MapReduce or Elastic Beanstalk on top of it. EC2 virtualizes the thousands of physical servers hosted in Amazon's data centers across the world and provides them as general purpose *Virtual Machines (VMs)*. As described in [52] Amazon uses a highly customized version of Xen [104] as *Virtual Machine Monitor (VMM)* to achieve virtualization in EC2. Further Unix guests are executed in Para Virtualization mode while for other OSes Full Virtualization is used. Booth virtualization techniques have already been discussed in Section 2.1.2 of this thesis. Because of the VMM Virtual Machines are isolated from each other so that they cannot access another VM's data. This also applies to network traffic, even if a VM's network card is – intentionally or not – set to promiscuous mode. These measures and various security certificates attested to AWS guarantee confidentiality of customer's data in EC2, as noted in [52].

The *Virtual Machines (VMs)* provided by EC2 are available in different configurations regarding computational resources they provide. These configurations are called instance types and currently there are 13 different instance types available.

An instance type is defined by the amount of storage, memory and computation power it comes with. AWS further categorizes instance types in usage classes like "high memory" or "high CPU", which does not limit for what an instance type can be used but is a recommendation for what usage the given instance type's configuration fits best. The designation, API name, storage, memory, and computation power of each EC2 instance type currently available is shown in Table 2.1.

Instance Type	**API Name**	**Storage**	**Memory**	**Compute**
Micro	t1.micro	—	613 MB	2 ECU
Small	m1.small	160 GB	1.7 GB	1 ECU
Medium	m1.medium	410 GB	3.75 GB	2 ECU
Large	m1.large	850 GB	7.5 GB	4 ECU
Extra Large	m1.xlarge	1,690 GB	15 GB	8 ECU
HiMem Extra Large	m2.xlarge	420 GB	17.1 GB	6.5 ECU
HiMem 2x Extra Large	m2.2xlarge	850 GB	34.2 GB	13 ECU
HiMem 4x Extra Large	m2.4xlarge	1,690 GB	68.4 GB	26 ECU
HiCPU Medium	c1.medium	350 GB	1.7 GB	5 ECU
HiCPU Extra Large	c1.xlarge	1,690 GB	7 GB	20 ECU
Cluster 4x Extra Large	cc1.4xlarge	1,690 GB	23 GB	33.5 ECU
Cluster 8x Extra Large	cc2.8xlarge	3,370 GB	60.5 GB	88 ECU
Cluster GPU 4x ExLarge	cg1.4xlarge	1,690 GB	22 GB	33.5 ECU

Table 2.1: Overview of Amazon EC2 instance types, data from [47]

Amazon measures computation power provided by an EC2 instance type in *EC2 Compute Units (ECUs)*. An ECU is a standardized unit, defined by AWS to be equivalent to the performance a 2007 AMD Opteron or Intel Xeon processor at 1.0 to 1.2 GHz clock speed delivers. This performance abstraction enables to keep the compute power of an instance type constant despite the capabilities of actual hardware it runs on.

An EC2 instance can be managed (created, restarted, etc.) programmatically by using Web Services. It is also possible to control the geographic region an EC2 instance is deployed to. Currently Amazon maintains data centers in five different regions, including United States (Virginia, California), Europe (Ireland), and Asia (Singapore, Tokyo) [52]. Stored data is not automatically replicated between regions and thus allows customers to fulfill local-dependent privacy and jurisdiction requirements by choosing the appropriate data center location for their EC2 instances [52]. Regions are further separated in so called "Availability Zones". Availability Zones are physically separated from each other, but located within the same metropolitan area. Each Availability Zone has its own independent hardware facilities, so that it cannot suffer from failure of hardware in another Availability Zone. Amazon encourages its customers to deploy their AWS applications to multiple Availability Zones and across regions to achieve best service reliability even when facing serious service interruptions due to natural disasters or system failures [97]. The downside of deploying to different geographic regions is that data traffic between EC2 instances running in different regions is charged separately by AWS. Also appropriate encryption methods should be used when sending data between regions, because AWS uses public Internet infrastructure for transmitting data across regions [52].

Like other virtualization software EC2 uses *Virtual Disk Images (VDIs)* to create virtual servers. In AWS these images are called *Amazon Maschine Images (AMIs)*. AWS provides various pre-built AMIs, created by Amazon itself or third-party vendors so that users do not have to build their own ones. Provided AMIs differ by the OS and software packages installed, hence each AMI is built for a certain purpose like hosting Web applications, database servers, etc. One big difference between ordinary VDIs and AMIs is that AMIs are read-only images. If an EC2 instance is terminated changes made to the AMI used are not saved persistently, meaning that if the instance is started again all changes previously made are lost [4]. Therefore multiple EC2 instances can be run in parallel sharing the same AMI. To save data beyond the runtime of a VM other AWS services, like Amazon Elastic Block Store and Amazon Simple Storage Service must be used.

2.2.2 Amazon Elastic Block Store (EBS)

The circumstance that AMIs are read-only and thus changes to the file system of an EC2 instance remain temporary (i.e. till it is terminated) makes EC2 unsuitable for various applications. For example installing a database server to an EC2 instance would be pointless, because database changes would be lost when the instance gets terminated. The same applies to log files, file uploads, etc. To overcome this limitation AWS offers *Amazon Elastic Block Store (EBS)*. EBS is a standalone service but integrates seamless into EC2. It offers raw block devices with sizes from 1 GB to 1 TB, which can be attached to an EC2 instance. For the OS running on the instance an EBS volume appears and behaves like an additional hard disk drive, regardless of its real location in the cloud. Opposite to AMIs EBS volumes store data durable when the EC2 instance is terminated and/or they are detached from the instance. [4, 47]

When creating an EBS volume the user has to specify the Availability Zone the volume should be stored in. AWS then transparently replicates the volume multiple times inside this Availability Zone to prevent data loss due to failure of a single hardware component. EBS volumes are not replicated across Availability Zones and thus EBS is not an as reliable storage service as S3. Therefore Amazon recommends in its best practices guide for AWS [97] to regularly make point-in-time snapshots of EBS volumes and store them to S3. A point-in-time snapshot is an incremental backup of an EBS volume, meaning that only data blocks changed since a previous snapshot made will be added to the snapshot. Another best practice is to encrypt data stored on EBS volumes by using e.g., encrypted file systems when building security sensitive applications [97].

EBS has also some limitations. Besides that EBS volumes are not replicated across Availability Zones it is not possible to make point-in-time snapshots of a volume if connected to a running EC2 instance [51]. Further it is not possible to attach an EBS volume to multiple EC2 instances at the same time. On the contrary connecting multiple volumes to an instance is possible. As described in [97] this offers the possibility to increase EBS's I/O performance by configuring the volumes as *Redundant Array of Independent Disks (RAID)* device.

Amazon charges a customer's EBS usage on a monthly basis by the amount of storage space used and the number of I/O operations performed.

2.2.3 Amazon Simple Storage Service (S3)

Beyond EBS Amazon offers another storage solution in AWS called *Amazon Simple Storage Service (S3)*. S3 is designed as storage for the Internet. It can be accessed through a REST- or SOAP-based Web Service interface and can store "[...] any amount of data, at any time, from anywhere on the web. [...]" as claimed by [48]. In contrast to EBS, which provides a raw block device for storing data, S3 is more structured. It defines *buckets* and *objects*. A bucket is a container which can store an arbitrary number of objects. Objects are fundamental entities stored in S3. An object consists of object data and metadata describing the object data. Usual metadata is the object's last modification date or its HTTP content type. [52]

S3 can handle objects with up to 5 TB in size and the number of objects a customer can store is not limited. When creating a bucket a customer can choose the region the bucket should be stored in. If objects are stored to a bucket AWS synchronously replicates the data to multiple Availability Zones within the bucket's region. Therefore S3 can sustain concurrent loss of data in two data center facilities, which translates into 99.999999999% durability and 99.99% availability over a given year [48]. There is an option in AWS called *Reduced Redundancy Store (RRS)*, allowing a customer to specify buckets which store data non-critical for her. RRS then reduces the level of redundancy S3 uses to store this data, leading to a reduced durability of 99.99% and the ability to only sustain the loss of data in a single data center facility. The benefit for the customer comes in reduced storage costs for data stored with the RRS option enabled. [48]

Additionally to RRS there are other advanced storage features in S3 like data versioning and *Access Control Lists (ACLs)*. Data versioning allows to store miscellaneous version of an object in S3. By default object requests will deliver the most recent version of an object stored, but particular versions of an object can be retrieved by specifying a version in the request [48]. ACLs allow to apply fine grained access restrictions on S3 buckets and objects. By default only the bucket and/or object creator has access to her buckets and objects, but this can be changed to also permit access for other AWS customers or even anonymous users. Access permissions can be configured on bucket and object level, but ACLs of buckets are not inherited to the objects they contain. Furthermore an Amazon S3 bucket can be configured to log access to it and the objects it contains. If this so called *access logging* is enabled information about each access request will be gathered, periodically aggregated into log files and finally delivered to a specified S3 bucket [52].

The monthly price a customer has to pay for using S3 depends mainly on 3 usage values: amount of data stored, number of requests made, and data transferred out of S3. Costs can slightly change when using RRS or changing regions.

2.3 Message Queues (MQ)

Distributed software systems have an inherent need for communication, to exchange information and synchronize with each other. There are various techniques available for communication in distributed systems, such as *Remote Method Invocation (RMI)*, *Remote Procedure Calls (RPCs)* or Web Services, but these are not sufficient when systems grow too large and/or become to complex. Therefore another type of software services was created, named *Message-oriented Middleware (MOM)*. [73]

Message Queues (MQs) are a fundamental concept within MOM. The big difference between RMI/RPC on one hand and MQs on the other hand is that former provides transient synchronous communication, while latter provides persistent asynchronous communication [91]. Before discussing MQs any further it is suitable to define the previously used terms:

- *Transient Communication:* Transient communication requires that all applications, taking part in a particular communication, are reachable and executing, otherwise all communication is lost. For instance, most transport-level communication is transient. If a target host cannot be reached messages will simply be discarded. [91]

- *Persistent Communication:* As its name suggests persistent communication does not discard messages, when a target system cannot be reached. Hence it is not required that sender and receiver applications are executed at the same time. The Internet e-mail System is a prominent example for a persistent communication system. [91]
- *Synchronous Communication:* When RPC is used to call a method on a remote host the caller method is blocked till the remote host has completed its computation. This behaviour is called synchronous communication. A disadvantage of synchronous communication is, that systems do not have processing control independence, as control is handed over to the called system for the time being. An advantage is, that synchronous communication guarantees the sequential order of messages. [73]
- *Asynchronous Communication:* Asynchronous communication negates the properties of synchronous communication. The caller retains processing control, as it does not need to block and wait for the callee's response, but sequential order of messages is not guaranteed any more. Further asynchronous communication requires an intermediary between sender and receiver, so that messages can be cached, if the receiver cannot be reached. [73]

As stated in the beginning of this section a MQ, or MOM in general, is capable of handling communication in large and complex systems. This is due to the use of persistent asynchronous communication and the system architecture it enables. There are four main advantages a MQ brings to communication for big distributed software systems: [73]

- *Coupling:* MQs provide asynchronous communication between sender and receiver by acting as intermediary between them. Due to that a MQ can be seen as independent layer between distributed software systems, that looses their coupling with each other, resulting in a highly cohesive, decoupled system deployment.
- *Reliability:* Besides asynchronism a MQ also provides persistency. This means that it safely stores a message as long as it takes to deliver the message to its recipient and get his acknowledgement in return. Because of this

a MQ is able to guarantee a message's delivery, which improves the communication's reliability.

- *Scalability:* By decoupling the interaction of distributed software systems, as acting as intermediary between them a MQ also decouples their performance characteristics. This is because when using a MQ a sender does not know anything about the message's receiver and vice versa. Hence both systems can be scaled independently with no disruption to one another or other systems.
- *Availability:* The loose coupling of software systems, enabled by use of a MQ, does not only increase the whole systems scalability, but also its availability. Because communication between systems is persistent and asynchronous, the failure of one subsystem does not necessarily result in failure of another subsystem, as messages can still be handed of to the MQ, if the messages' receiver is down.

MQs support different types of messaging models. The two most important ones are the *Point-to-Point* and the *Publish/Subscribe* models, which are outlined in Figure 2.6. As illustrated in the figure both models are based on the exchange of messages through a channel. Typically the channel is called "queue" in the *Point-to-Point* model and "topic" in the *Publish/Subscribe* model. In the following enumeration both models will be discussed in detail:

- *Point-to-Point Model:* The Point-to-Point model can be seen as a logical 1-to-1 connection, connecting one sender with one receiver through a queue. As mentioned earlier sender and receiver do not know each other, they connect to the queue, which is identified by an unique name, and hand over message delivery to it. A MQ can host multiple queues, and thus communication channels, simultaneously. Typically the order a queue sorts the messages it receives can be configured. The most common configuration used is a *First-In-First-Out (FIFO)* queue, where messages are sorted in the order they where received. As shown in Figure 2.6 a queue is not strictly limited to one sender and one receiver. Despite that the queues logical behaviour stays the same, meaning that even if multiple receivers are connected to a queue a message will only be forwarded to one of them. This can be used as an easy way to introduce load balancing into a system. [73]
- *Publish/Subscribe Model:* The Publish/Subscribe model can be seen as a logical 1-to-n association, connecting one sender (publisher) to multiple receivers (subscribers), through a so called topic. As in the Point-to-Point

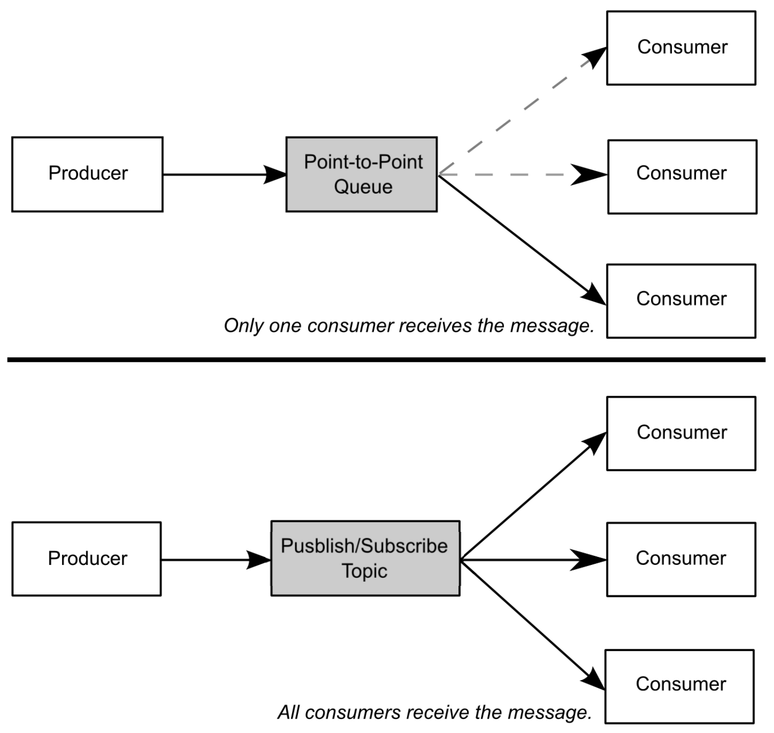

Figure 2.6: Messaging models compared, inspired by [73]

model sender and receiver do not know each other. They connect to a topic and hand over message delivery to it. Like queues topics are identified through an unique name and a MQ is also capable of hosting multiple topics simultaneously. In contrast to queues topics replicate a received message as often as necessary to provide each connected receiver with it. Therefore it is possible that a sender can reach thousands of receivers by only sending a single message. In practical application a topics 1-to-n association is not a strict limitation. It is also possible that multiple senders are connected to the same topic, so that they effectively share the same set of receivers. [73]

2.4 Aspect-oriented Programming (AOP)

AOP is a new programming paradigm which helps achieving better modularization in software development. It does not replace other design methodologies, such as *Object-oriented Programming (OOP)*, but enhances them to overcome their shortcomings [67]. While much of the theoretical founding of AOP was done at universities all over the world a research team at Xerox PARC coined the term AOP in a paper in 1996 [63] and also created AspectJ [92, 93], one of the first practical implementations of AOP [67]. In 2002 Xerox PARC handed the AspectJ project over to the Eclipse Foundation, a non-profit open-source community, which now actively develops AspectJ [79].

2.4.1 Definition of Aspect Orientation

AOP introduces a couple of new technical terms, like *Aspects*, *Pointcuts* and *Code Weaving.* To understand the concepts of AOP the meaning of these terms has to be known. An explanation is given in the following:

- *Cross-Cutting Concerns:* As mentioned before AOP helps to overcome shortcomings of current programming methodologies. For OOP this means handling of a software's cross-cutting concerns. Each application consists of core concerns and system-wide concerns. In OOP concerns are mapped to modules, so core concerns represent the software's main modules whereas system-wide concerns represent its supportive modules. Supportive modules are in general utilized by main modules to cover common tasks. Such system-wide concerns, which span multiple modules are called "Cross-Cutting Concerns" [67]. The most common example of a Cross-Cutting Concern is that of logging. Logging is a system-wide concern that in general affects every other concern in a system [79].
- *Aspects:* An aspect is a Cross-Cutting Concern that is packed into a separate module and called by the AOP runtime whenever necessary. This means that with AOP cross-cutting concerns can be implemented as complete independent modules without the need of fusing invocation code in the software's main modules [67]. As defined in [79] an aspect has to fulfill four requirements: it must be definable in a modular fashion, it must be dynamically applicable, its application must accord to a set of rules, and it must provide a mechanism and a context for specifying the code it will be executing.
- *Join Points:* Join points are points within an application which can trigger the execution of an aspect. What points in an application can act as a join point depends on the capabilities of the used programming language and AOP tools. Common examples for join points are the invocation of a constructor or the initialization of an object. [79]

- *Pointcuts:* A pointcut actually decides if an aspect is executed when a join point is reached. It encapsulates logic that is evaluated when the pointcut's corresponding join point is triggered and invokes the aspect if that logic evaluates to true. [79]
- *Code Weaving:* Code weaving is the name of the process that "weaves" AOP code (i.e., aspects) and ordinary program code (i.e., the software's core modules) together. The actual weaving process is done by an AOP tool usually called "weaver". A weaver can be implemented in various ways, which affects how code is woven together. One way is source-to-source translation. Here source code of core modules and aspects are preprocessed by the weaver and woven source code is produced, which is than compiled by the language compiler to executables. Another approach used by AspectJ is to use a special Java class loader, which loads the Bytecode of aspects and ordinary Java classes, weaves them together and than hands the new Bytecode over to the virtual machine. All approaches have in common that the original source code is not modified by the weaver. [67]

To better understand how the use of AOP changes the logical execution order of an application a comparison is sketched out in Figure 2.7. The sketch at the top of the figure shows how a logging module, which is a Cross-Cutting Concern, is used by core modules, such as accounting and persistence. The grey bars inside the core modules indicate code that is used to invoke the logging module. The bottom of the figure shows the same scene but now implemented using AOP. The core modules now do not implement any code to invoke the logging module. Instead the logging aspect is automatically invoked when the appropriate join points inside the core modules are reached. The logging aspect then calls the logging module if the pointcut's decision logic evaluates to true.

2.4.2 Benefits of AOP

AOP is often criticized for being to difficult to understand and also to make it harder to understand a program's execution behaviour because of its join points and pointcuts which become active just at runtime. But as stated in [67] these acceptance issues are mostly base to the fact that AOP is a new programming methodology which demands a new thinking about system design and implementation. Besides that the authors of [67] also state that the benefits of AOP far outweigh the perceived costs and therefore its worth using AOP. Some of the benefits claimed by AOP are listed in the following:

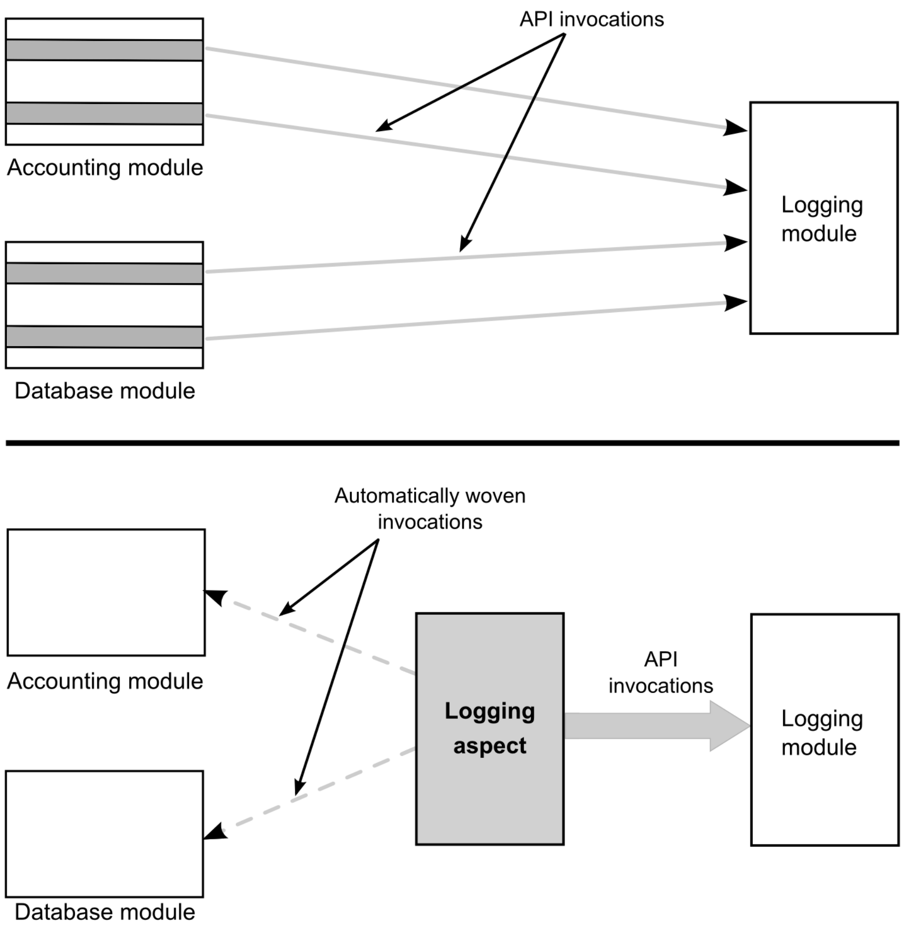

Figure 2.7: Top: Logging w/o AOP / Bottom: Logging with AOP, see [67]

- *Separation of Concerns:* Because AOP improves handling of Cross-Cutting Concerns it becomes much easier to design a software's core modules according to the Separation of Concerns principle. As an example see Figure 2.7. Both core modules do not need to implement logging code, when AOP is used. This results in cleaner assignment of responsibilities and better traceability. [67]
- *Modularization:* Besides better implementation of the Separation of Concerns principles AOP also encourages higher modularization of software. With AOP each concern can be addressed separately which leads to reduced coupling between objects and lesser duplicated code. Both results make software maintenance easier. [67]
- *Late binding of design decisions:* A common dilemma for software architects

is that of under- or overdesigning a software. Overdesigning occurs when an architect designs a software so that it can easily be extended to support functionalities which may be needed in the future. Underdesigning obviously describes the opposite behaviour. The problem is that future requirements of software are rarely foreseeable, so that such provisions are often result in wasted effort. With AOP under- and overdesigning is not necessary any more because due to the use of aspects new features can easily be added without the need for system wide modifications. [67]

- *Better code reuse:* With AOP each aspect is implemented as a separate module, leading to more loosely coupling between objects, which is the basis for a higher degree of code reuse. One just has to change the weaving specification instead of various core modules to change a system's configuration. [67]
- *Improved Time-to-Market:* All aforementioned benefits result to a better software development process. Improved Separation of Concerns allows better assignment of a core module to the developer's skills. Better modularization make software maintenance easier. Late binding of design decisions reduces the amount of wasted development effort and better code reuse allows faster development. All of these lead to a significant reduction of time-to-market. [67]

CHAPTER 3

Related Work

This chapter will discuss selected technologies and frameworks which, like jCloud-Scale, also enable the distribution of an application's objects among several computer nodes. Hence these technologies can be used for programming distributed applications and therefore also cloud applications. After each technology's description a comparison with jCloudScale will be given.

3.1 Enterprise Java Beans (EJBs)

In 1999 Sun Microsystems introduced the *Java Enterprise Edition (Java EE)*, which is an umbrella specification for building enterprise-class distributed applications using the Java programming language. Java EE applications are executed in a special runtime environment – called application server – which consist of several logical domains called containers. Each container has a specific role and provides various services, like database access, transaction handling, security etc., to the components it supports. *Enterprise Java Beans (EJBs)* are components handled by the EJB container of an application server and also build the core of the Java EE specification. [39]

Components executed in different containers of an application server can interact with each other using protocols like *Remote Method Invocation (RMI)* and *Hypertext Transfer Protocol (HTTP)* (for SOAP and RESTful Web Services). This also applies to components running inside application servers hosted on different computers and connected through a network. EJBs are used for processing the transactional business logic of an application and to map database tables which persistently store an application's business data. [39] There are different types of

EJBs, each one fitting best for a particular use case. First EJBs can be categorized in session beans, message-driven beans and entity beans. Session beans can be further distinguished in stateful session beans, stateless session beans, and singleton session beans. [87]

As already mentioned at the beginning of this section Java EE is a collection of numerous specifications. This is a great benefit for the platform because it led to various implementations of Java EE runtimes from different vendors. Any Java EE enterprise application can be executed on any Java EE runtime if both artifacts (i.e., the runtime and the application) correspond to these specifications [87]. This compatibility guarantee and the component model architecture of EJBs make them a perfect technology for writing cloud-based applications. Indeed lately there have emerged numerous *Platform-as-a-Service (PaaS)* solutions leveraging EJBs to execute and scale applications inside a Cloud Computing environment. A description of some selected services of these cloud offerings is given in the following:

- *Amazon Elastic Beanstalk*: Elastic Beanstalk is executed on top of *Amazon Elastic Compute Cloud (EC2)*, transforming the *Infrastructure-as-a-Service (IaaS)* solution into an PaaS one. For Java developers it uses Apache's Tomcat server as execution environment, so that a developer only has to upload her application as a *Web Application Archive (WAR)*. Everything else (deployment, scaling, etc.) is than automatically handled. Apache's Tomcat does not implement an EJB container, meaning that a developer has to use third-party libraries if she wants to use EJBs. [49]
- *Oracle Java Cloud Service*: The Java PaaS solution from Oracle provides full Java EE support by using Oracle WebLogic application server as execution environment. Besides that the service also provides the Oracle database cloud service and integrated identity management for enterprise customers. It is possible to use the service in a public or private cloud configuration or migrate one's application completely to an on-premise cloud infrastructure. [24]
- *CloudBees*: CloudBees is a PaaS solution providing support for separated development and production environments. Developers can deploy their application to a development environment, where they can build and test their application using the Jenkins continuous integration server. Afterwards the applications can seamlessly be moved to a production environment inside the CloudBees cloud. CloudBees provides full Java EE support and can be used with various IaaS providers. [20]

- *Jelastic*: Jelastic is a PaaS solution that can be used with various IaaS cloud provider located in different countries to meet an application's performance and/or legal requirements. A developer can configure the application's execution environment by choosing between different application servers and database servers and by setting limits for horizontal and vertical scaling. Jelastic features full support for a Java EE execution environment. [59]

3.1.1 Comparing EJBs with jCloudScale

EJBs and jCloudScale have very different approaches and also aim to achieve different goals. EJBs are part of the Java EE specification, implement a component model and are not meant to be executed standalone. They are executed inside an EJB container, which also controls their life cycles and provides additional services such as transactions and naming. The container decides when to create, destroy, passivate or activate an EJB. Therefore debugging EJBs is hard and needs to be supported by the container. EJBs were designed for executing business logic in multi-layered, multi-user enterprise applications.

jCloudScale on the other hand is a lightweight middleware using *Aspect-oriented Programming (AOP)* to deploy plain Java objects into an IaaS cloud. Applications decide when to create and destroy *Cloud Objects (COs)*. With AOP disabled an application acts like an ordinary local Java application and thus can be as easily debugged. jCloudScale works best for applications, which can split their workload several independent packages, so that each one can be executed by a CO.

3.2 Uni4Cloud

Uni4Cloud [89] is developed at the Department of Applied Informatics at University of Fortaleza. Its goals are to provide automated deployment of applications to IaaS clouds independent of used cloud provider and also to deploy applications composed of multi-cloud components. To achieve this Uni4Cloud relies on open Cloud Computing standards: *Open Virtualization Format (OVF)* and *Open Cloud Computing Interface (OCCI)*. OVF provides a vendor and platform neutral solution for specifying *Virtual Machine Interface (VMI)*. OCCI provides a common *Application Programming Interface (API)* specification for common IaaS cloud services, such as storage and *Virtual Machine (VM)* management.

The Uni4Cloud approach is composed of three main services [89]:

- *Service Modeler*: The service modeler is a visual tool for building and configuring OVF images. Developers have to provide pre-built templates for VMs and define parameters for their deployment. The service modeler then produces an OVF virtual appliance.

- *Service Manager*: The service manager takes OVF virtual appliances produces by the service modeler and actually deploys them to one or multiple IaaS clouds. It does this by using a OCCI-compliant API provided by the cloud adapter component.
- *Cloud Adapter*: The cloud adapter basically translates OCCI-complaint API calls to specific API calls for the IaaS providers used. Therefore the cloud adapter component provides a plugin interface. Wrappers, encapsulating an IaaS cloud's API, can be connected through this interface and used by the cloud adapter.

3.2.1 Comparing Uni4Cloud with jCloudScale

Uni4Cloud and jCloudScale share the same goal: decoupling applications from the underlying IaaS cloud infrastructure and thus eliminating vendor lock-in. But the approaches to achieve this are quiet different for both projects. Currently Uni4Cloud only focuses on application deployment. A developer has to pack her application alongside with the whole execution stack needed in an OVF image and configure the number of instances etc. Uni4Cloud will then deploy the application, but will not provide any further services. For instance, as for now, Uni4cloud does not support auto-scaling.

jCloudScale also uses VMIs to create cloud nodes, but a developer does not have to provide them for each application, instead jCloudScale uses a generic image for any kind of application. When deploying an application's objects to cloud nodes jCloudScale copies all necessary binaries and libraries of that application automatically to the cloud nodes. jCloudScale also takes care of auto-scaling, demanding developers to only specify a scaling policy for their application.

3.3 Mobile Cloud Middleware (MCM)

Mobile Cloud Middleware (MCM) [32] was developed at the Distributed System Group of University of Tartu. As its name suggests its target is to be a middleware for mobile devices, like smartphones, trying to connect to cloud services. Nowadays the hardware of mobile devices is powerful enough to execute a variety of applications, ranging from simple calculators to more complex map applications and even graphical sophisticated games. These applications often use cloud services to enrich their user experience or to outsource computational complex tasks, yet to demanding for mobile hardware. Because nearly every cloud service comes with its own proprietary API mobile apps can become extensive to develop when using multiple services. MCM tries to overcome this problem by providing common API for different cloud services.

MCM consists of three main components. When using MCM a mobile app connects to MCM's transportation handler component. The transportation handler than forwards the request to the MCM manager, which decides which cloud service to use and requests a matching servlet from the adapter servlet component. The adapter servlet component looks up a servlet containing a set of functions for consuming the selected cloud service and provides it to the MCM manager. Finally the MCM manager creates a cloud service adapter, which it subsequently uses to communicate with the selected cloud service. The result of each cloud transaction is sent back to the mobile device in *JavaScript Object Notation (JSON)* format.

MCM is implemented in Java using servlet technology and therefore can be executed on any application server featuring a servlet container. Currently MCM supports cloud services from Amazon, including EC2 and *Amazon Simple Storage Service (S3)*, Google, Eucalyptus and *Software-as-a-Service (SaaS)* offerings from Facebook, Face.com, etc.

3.3.1 Comparing MCM with jCloudScale

jCloudScale and MCM are quite different. MCM is a unified gateway to various cloud services, especially SaaS ones. It focuses on providing its services to mobile platforms and therefore also implements the asynchronous notification mechanisms for these platforms. MCM is not transparent in any way for applications using it and does not provide any scaling mechanism but the capabilities of the application server it is executed on.

jCloudScale can be used with any program written in Java, tries to be as transparent for the developer as possible and provides an automated scaling mechanism through scaling policies. In contrast to MCM jCloudScale is limited to IaaS clouds, but can be used in a much wider application context.

3.4 Aneka

Aneka [98] is a framework for writing applications that execute and scale inside an IaaS cloud infrastructure. It was originally developed at the Department of Computer Science & Software Engineering at the University of Melbourne within their Gridbus project [16], but is now a commercial software at Manjrasoft. Aneka is based on the .NET framework [22], but written with portability in mind and therefore also compatible with the Mono framework [25]. This makes it possible to execute Aneka on IaaS nodes running Windows- as well as Unix-based *Operating Systems (OSes)*. Further distributed applications developed with Aneka can be written in any programming language supported by the .NET runtime.

Aneka is logically located between the physical and virtual resources of the cloud infrastructure it runs on and the distributed applications developed with it. Aneka provides four main services to an application developer. These services are executed inside the Aneka container [98]:

- *Fabric Services:* These services provide hardware profiling and dynamic resource provisioning. Profiling collects runtime information about nodes Aneka runs on, while provisioning acquires and integrates new nodes dynamically. Both services are accessible through vendor independent APIs, so that Aneka applications do not depend on proprietary APIs of the underlying IaaS provider used. Aneka supports multiple IaaS provider, such as EC2.
- *Foundation Services:* These are Aneka's core services. They address four issues: maintaining a list of all nodes inside an Aneka cloud, reserving nodes exclusively for executing Aneka applications with special requirements, providing persistent storage management and data transfers for applications, and keeping track of resources used by users so that they can be billed afterwards.
- *Execution Services:* Aneka supports multiple programming models. At default Aneka comes with support for the *Task model*, the *Thread model*, and the *MapReduce model* but support for other models can easily be added. For each supported programming model at least two services have to be provided: the *Scheduling Service* and the *Execution Service*. The scheduling service coordinates execution of applications and dispatches collection of jobs generated by these applications to the compute nodes. The execution service is in charge of retrieving all files required for execution, the actual application execution itself, monitoring and collecting the results.
- *Transversal Services:* Persistence and security are services used by all other services of the Aneka container. Persistence keeps track of information such as the applications running in the Aneka cloud and their status, the status of the storage, topology information of the cloud etc. All this is saved to

persistence storage, so that restoring after a system crash or partial failure is possible. The security layer consists of an authentication and authorization service, which are used to identify users and check what they are allowed to do in the Aneka cloud.

The biggest benefits of Aneka are that it acts as an abstraction layer for the underlying cloud infrastructure and that it simplifies scaling of distributed applications. An application developed with Aneka can be executed and scaled on any Cloud Computing infrastructure supported by Aneka without needing to modify it. Future development of Aneka will focus on full support for elastic scaling of Aneka clouds and support for additional third-party Cloud Computing providers. [98]

3.4.1 Comparing Aneka with jCloudScale

Aneka and jCloudScale are both middlewares for IaaS cloud offerings, trying to decouple applications from the underlying cloud infrastructure. Aneka is written using the .NET framework and thus only supports applications written in .NET-compatible languages. The framework provides various services to applications, like persistent storage management and security, making it more convenient for developers writing distributed applications.

jCloudScale is written in Java, supporting only Java applications. Beside an API for retrieving some statistical data about jCloudScale's state and several annotations jCloudScale provides no visible services to application developers. This is because in contrast to Aneka jCloudScale tries to be as transparent for developers as possible by not forcing them to use special APIs and thus programming for a particular framework.

3.5 Composite Application Framework (CAFE)

The *Composite Application Framework (CAFE)* [77, 78] was developed by Ralph Mietzner at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart during his phd work. A composite application describes a software that is composed out of different components. These components are in general Web application and/or Web Services. Thus the goals of CAFE are "[...] to describe configurable composite service-oriented applications and to automatically provision them across different providers. [...]" [78].

With the advent of Cloud Computing and the emerging of cloud service providers different aspects of an application can be outsourced. As described in Chapter 2 the SaaS model allows to outsource whole applications, while the PaaS and IaaS models allow to outsource platform support respectively hardware infrastructure. To take full advantage of these outsourcing possibilities a new software architectural style called *Service-oriented Architecture (SOA)* became popular. SOA designed applications are assembled out of services which may run on the infrastructure of different cloud service providers [78]. As mentioned above CAFE now tries to describe how these SOA applications are configured and provisioned. In a typical CAFE scenario a user selects an application at any given CAFE application provider, defines the desired *Service Level Agreement (SLA)* values for it and customizes the application to her needs. CAFE then automatically deploys the application across different providers, so that the user's selections are met. [77]

To achieve this there are three main roles defined in the CAFE process: *Application Vendor, Application Provider*, and *Application Customer.* [77]

1. *Application Vendor:* An application vendor's task is to build CAFE applications. These applications can be composed out of existing services provided by third-party providers, self-developed from scratch, or even a mixture of both. To become a CAFE application all artifacts of an application together with an application descriptor file have to be packed into a CAFE application archive file.
2. *Application Provider:* An application provider hosts one or more CAFE applications, so that third-parties can use them. It's possible that the components of a CAFE application are hosted at different application providers, hence an application provider can also host only parts of a CAFE application. The delivery model (IaaS, PaaS, SaaS) used by the application provider does not matter to CAFE.
3. *Application Customer:* The third role in the CAFE process is called application customer. An application customer wants to use one or more CAFE applications and may not necessarily be an actual person. A customer may be an enterprise and therefore consist of multiple users.

Some of CAFE's main research issues are to separate application vendors and providers, to allow "[...] customers to follow a best-of-breed strategy [...]" [77], to enable consumer- and process-driven customization of applications, to define an architecture and methodology of provisioning of applications across different providers and to optimize the distribution of these applications across different providers.

3.5.1 Comparing CAFE with jCloudScale

CAFE and jCloudScale follow very different goals. CAFE tries to orchestrate different services hosted at different providers to achieve a unified experience for the user. It does this by using provided meta-information for each service. CAFE focuses on the end user and does not provide any abstraction for application developers.

jCloudScale in contrast focuses on the application developer. It tries to make developing distributed applications easier and decouple applications from underlying IaaS cloud offerings.

CHAPTER 4

Background

The jCloudScale middleware is the cornerstone of this thesis, which main goal is to design and implement a migration mechanism for objects managed by jCloudScale – so called *Cloud Objects (COs)*. Therefore and as introduction for Chapter 5, where the design and implementation details of this migration process will be described, jCloudScale's architecture and basic concepts are presented in the following.

4.1 jCloudScale

jCloudScale [68,69] is a middleware which aim is to ease development of distributed applications on top of *Infrastructure-as-a-Service (IaaS)* cloud offerings. It was developed by *Distributed Systems Group (DSG)* at Vienna University of Technology, is entirely written in Java and therefore currently only supports Java-based client applications. It's code is licensed under Apache License 2.0 (*Apache Software Licenses (ASLs)* 2.0, [33]), a free software license, that allows use for any purpose, to modify and redistribute again. jCloudScale is available through it's project website [68], alongside with some documentation on how to use it.

jCloudScale implements a declarative deployment model, which means that an application developer only has to specify some scaling policies for her application – using code annotations – to make use of the middleware. jCloudScale's code is than loaded automatically at application startup time, using *Aspect-oriented Programming (AOP)* [63, 67] and Bytecode manipulation techniques. The main advantage of this approach is that a client application does not have to be implemented towards a particular execution environment, instead an application using jCloudScale can be executed as ordinary local Java application by just disabling AOP code weaving. Common *Platform-as-a-Service (PaaS)* cloud offerings like *Google App Engine (GAE)* [54] lack such flexibility and therefore impose tight vendor lock-in on a developer's application. On the other hand IaaS offerings are as flexible as jCloudScale but demand more programming, deployment and monitoring effort. Therefore in the larger taxonomy of Cloud Computing, jCloudScale fills the space between PaaS and IaaS, as stated by its authors in [69]. Because jCloudScale acts as a middleware between an application and an IaaS cloud provider's infrastructure, only jCloudScale needs to be reconfigured when changing IaaS providers.

Due to jCloudScale's declarative deployment model a developer can decide (by using code annotations) which ones of her application's objects should be managed by jCloudScale. Such objects are called *Cloud Objects (COs)* and executed on *Cloud Hosts (CHs)* inside an IaaS cloud. Because COs are distributed among different CHs jCloudScale is best used for computing-intense scientific applications, where tasks can be divided and carried out simultaneously by multiple COs.

4.1.1 Architecture

It can be said, that jCloudScale has to handle two main tasks: hiding the distributed nature of COs from client applications and managing CHs and the COs executed on them. Indeed jCloudScale's architecture pictures this. The client part of jCloudScale and the developer's application are both executed on a local machine, while jCloudScale's server part runs on each CH created by jCloudScale. For the server part jCloudScale uses a custom tailored *Virtual Disk Image (VDI)*, which consists of a trimmed down Linux *Operating System (OS)* and all binaries necessary to execute jCloudScale's server part. An architectural overview of jCloudScale showing it's client and server part and their main components is shown in Figure 4.1.

The client part consists of five main components: `CloudScaleClient`, `CloudManager`, `CloudScaleReferenceManager`, Host- and Monitoring repositories. `CloudScaleClient` ensures that all components necessary for running jCloudScale are set up and started before any jCloudScale functionality is actually used. This includes starting a *Message Queue (MQ)* service, that will handle all communication between jCloudScale's client and multiple server sides and distributing a common jCloudScale configuration used by all components. The `CloudManager` implements methods for managing CHs and COs. It creates and destroys COs, uses a scaling policy to decide when to start and stop CHs and invokes CO methods. The `CloudScaleReferenceManager` manages data han-

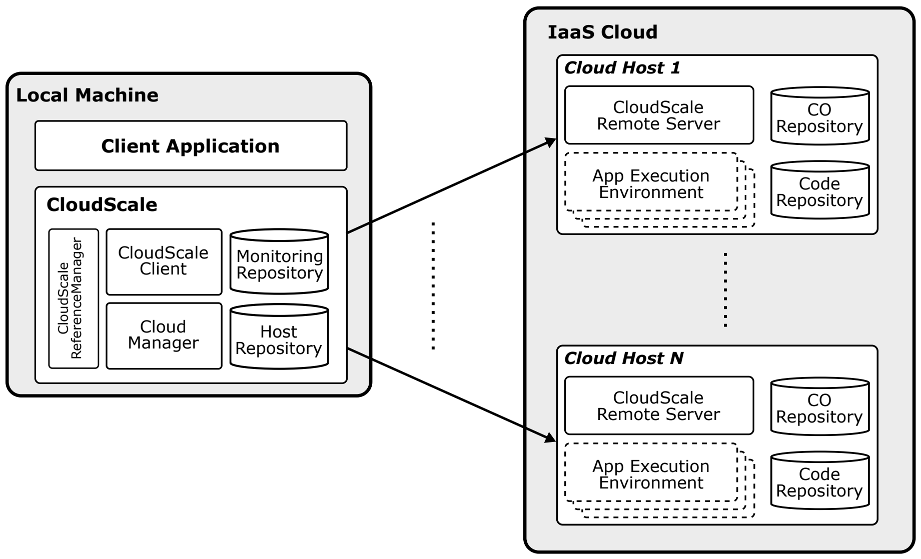

Figure 4.1: Architectural Overview of jCloudScale, based on [69]

dling between ordinary Java objects and COs. When a client application invokes a CO's method with a complex type as parameter the parameter's value is by default not serialized and sent to the managing CH but replaced by a `CloudScaleReference` object. If the reference object is then invoked at the CH the `CloudScaleReferenceManager` transparently resolves the reference and invokes the original object at the client. This removes the need of making all complex types of a client application serializable when interacting with COs. Host- and Monitoring repositories are used to keep track of managed CHs respectively of monitoring events created by these CH.

A server part consists of four major components: a jCloudScale remote server, App execution environments, CO- and code repositories. The remote server component listens for commands from the client's `CloudManager` and executes them. App execution environments are created for each CO executed on the CH so that they cannot interfere with each other. The CO repository is used to store an index of all COs hosted on the CH, so that they can easily be looked up and accessed when requested. The code repository is used to store the COs' class files and optionally other binaries necessary to execute them.

A typical jCloudScale interaction consists of several steps. It is started when the client application tries to invoke a method of a CO. First the invocation is intercepted by an AOP aspect and redirected to the `CloudManager` component. The `CloudManager` then looks up the CH that actually manages the CO. Afterwards the invocation command is forwarded to the remote server component of this CH and the CO is looked up in the Cloud Host's CO repository. At last the remote server component invokes the requested method on the CO and sends the methods return value – if any – back to the client application. This interaction is basically an implementation of the Broker remoting pattern, as shown in [99].

4.1.2 Eventing in jCloudScale

jCloudScale provides the ability to monitor the state of CHs and COs through an event-based system. Events are ordinary serializable Java objects carrying payload regarding the event they represent. jCloudScale currently comes with seven predefined event types and three event categories, ordered hierarchically as shown in Figure 4.2.

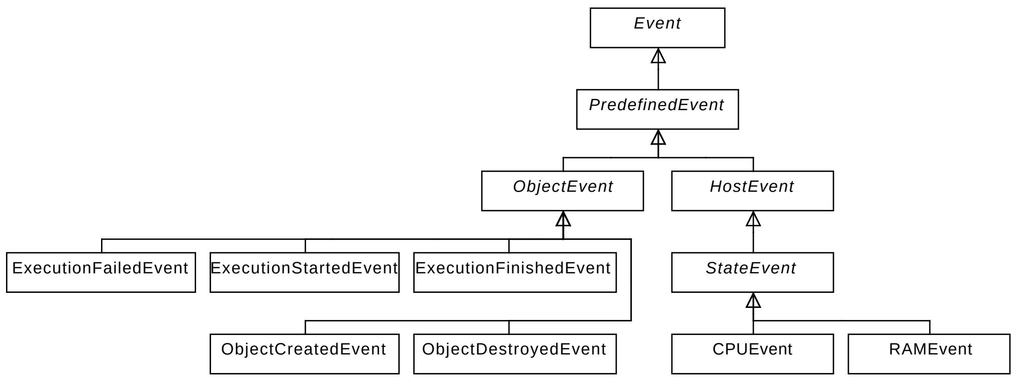

Figure 4.2: Event class hierarchy in jCloudScale

Requiring events to inherit from abstract class `Event` ensures that all events implement the `Serializable` interface and provide variables for holding their unique identification value and timestamp of their creation. Further the deep inheritance structure of events allows easy filtering by category in event receivers.

Events can be issued by all three main components of jCloudScale: CHs, COs and the `CloudScaleClient` itself. After their creation events are distributed to interested consumers using a message queue service and a dedicated topic specified in jCloudScale's `MonitoringConfiguration`. Events can be triggered in three different ways:

- *Periodically:* Events are periodically triggered after a certain time interval has passed. For instance, events of category `StateEvent` are created this way. The `MonitoringConfiguration` provides a time interval value that should be used by all events triggered this way.

- *On certain events:* The usual way of triggering events. Since jCloudScale supports AOP, aspects can be used to create event objects on occurrence of certain events, such as deployment of a CO or invocation of a CO method. All predefined events of category `ObjectEvent` are created this way.
- *Manually:* Events can also be triggered programmatically during the execution of a CO method. For this to be possible there needs to be an interface to the MQ accessible from within COs. jCloudScale provides this interface through dependency injection. When a CO is deployed jCloudScale injects an appropriate object into the CO's variables annotated with `@EventSink`, which can be used by the CO for sending event messages.

The generic approach and overall architecture of jCloudScale's eventing system permits extension with new event types and categories easily. Therefore the underlying concept of this system appears to be a good staring point for designing a decision mechanism for migrating COs inside a jCloudScale cloud, which is vital for a capable migration system. The next chapter describes the design and implementation of such a migration system for jCloudScale.

CHAPTER 5

Design and Implementation

Chapter 1 has given an elaborate introduction to the motivation and desired contribution of this thesis. Chapter 2 and Chapter 3 were used to discuss the foundations of Cloud Computing and similar distributed technologies. In the previous chapter an overview of jCloudScale's architecture and eventing system was given to make it more comprehensible how jCloudScale was extended to support the migration of *Cloud Objects (COs)* between *Cloud Hosts (CHs)*. Finally this chapter will discuss at length all work done and design decisions made to achieve the goal of migrating COs.

In the introduction of this thesis some desired key features of a migration mechanism for jCloudScale were stated. Then a list of contributions this thesis' work should make to jCloudScale was derived from these features. Extracting these contributions declared in Chapter 1 results to the following list of properties the migration mechanism should incorporate:

- migrations should be transparent for COs
- migrations must be transparent for objects interacting with COs
- a CO's state must be preserved during migration
- migrations must not interfere with jCloudScale's normal operation
- triggering of migrations should be policy-based and configurable by the user
- the decision which CO and CH taking part in a migration should be based on provided metrics and configurable by the user

This chapter is structured into three parts. First, Section 5.1 discusses the considerations and design decisions made while implementing the previously enumerated features into the migration mechanism. Afterwards Section 5.2 describes the migration of a CO in detail, using a simple fictional use case to aid the reader's understanding. Finally Section 5.3 ends this chapter by giving an architectural overview of jCloudScale's migration system.

5.1 Design considerations and decisions made

Some of the desired features of jCloudScale's migration mechanism stated in the introduction of this chapter are non-trivial and therefore must be discussed comprehensively. Therefore this section focuses on describing the considerations made regarding these non-trivial features.

5.1.1 Code mobility in distributed systems

In object-oriented programming an object is characterized by three properties: identity, state, and behaviour. Its identity distinguishes the object from other objects, its state describes the values currently assigned to its member variables, and its behaviour is defined by the methods the object implements. Migrating an object from one execution environment to another one is also called Code Mobility. As described in [74], or to more extent in [36], Code Mobility can be differentiated in *strong* Code Mobility and *weak* Code Mobility. Both Code Mobility techniques migrate an object's code, i.e. its behaviour, to the new execution environment, but only strong Code Mobility also migrates the object's state. As enumerated in the introduction of this chapter, there are two mandatory requirements for the migration process. First, the state of a CO must be preserved during migration, and, second, its migration must be transparent for other objects interacting with it. To achieve this, the migration process must at least transfer two of an object's properties – its state and its behaviour – to the new Cloud Host. Hence only strong Code Mobility is suitable for jCloudScale's migration mechanism.

As further stated in [74], solutions implementing strong Code Mobility can be classified into five categories. These categories are derived from their implementation level. The following list shows these categories in ascending order, starting with the lowest-implementation level:

- *Operating system level:* Enables the migration of *Operating System (OS)* processes between *Computational Environments (CEs)*. This kind of mobility requires support by the OS and is limited to process migrations between the same Operating Systems, which decreases this solution's portability. Both properties make this solution inapplicable for jCloudScale.

- *Virtual machine level:* Presupposes that the CE is executed inside a *Virtual Machine (VM)*. A migration will migrate the whole VM from one physical host to another one. Works only if the source and destination physical host have the same machine configuration. This solution is also inapplicable for jCloudScale, as it does not allow to migrate a specific CO from one host to another.

- *Middleware/Engine level:* Extends the employed middleware or engine to achieve strong mobility. In jCloudScale's case this means modifying the Java VM executing jCloudScale. Consequently this would reduce jCloudScale's feature set on unmodified VMs and thus diminish its portability severely. Therefore, this solution cannot be used either.

- *Bytecode level:* This solution is specific to programming languages compiled to Bytecode instead of compiled to native machine code. Java is such a language. It provides the ability to serialize objects and deserialize them later again. This approach implies some limitations, but nonetheless seems to be suitable for jCloudScale.

- *Source code level:* This solution provides programming libraries for enabling Code Mobility. It requires the developer to instrument their code by adding explicit calls to these libraries. Because of this, the source code level solution is not transparent for the developer's code and thus is not suitable for jCloudScale either.

Of these five solutions that can be used for implementing strong Code Mobility the Bytecode level solution was chosen as most suitable for jCloudScale's migration mechanism. This solution draws some limitations on the migration process but they are outweighed by the fact that it does not require any modifications be made to the Java VM or the underlying OS. Therefore, jCloudScale's compatibility with standard Java runtimes and its portability across different OSes stays intact. jCloudScale's usage of this Code Mobility solution is described at length in Section 5.2. Figure 5.1 shows an overview comparison of the execution models of programs written in Java and programs written in programming languages that compile directly to native machine code, like C++. As one can see instead of compiling directly to executable machine code Java introduces an additional layer of abstraction. This layer consists of Bytecode and the Java VM, which interprets and executes the Bytecode. The introduction of additional abstraction layers does

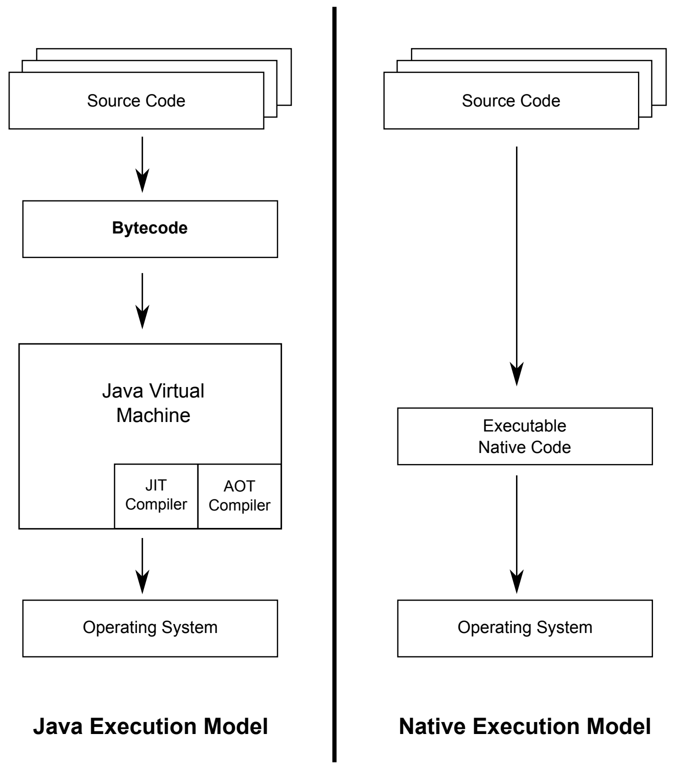

Figure 5.1: Java Execution Model vs. Native Execution Model

in general reduce computation performance, due to additional overhead, which is also true for the Java execution model. To mitigate this disadvantage the Java VM uses techniques, such as *Just-in-Time (JIT)*- and *Ahead-of-Time (AOT)*-compilers. With JIT Bytecode is compiled into native machine code by the VM during execution, while with AOT Bytecode is compiled before its execution. With these techniques Java's execution model is performance-wise nearly on par with the native execution model [26, 80].

5.1.2 Checkpointing and synchronization

Before a Cloud Object's code and its state can be transferred from one CH to another CH the object's state and execution has to be frozen at some point. This process is called Checkpointing. After the object was successfully migrated to its new CH, execution will be resumed at the previously saved checkpoint. As for Code Mobility, there are also different common approaches available for Checkpointing. Some of these techniques, like Checkpointing at natural synchronization barriers, coordinated Checkpointing, or uncoordinated Checkpointing with message logging, are briefly discussed in [74]. Each Checkpointing technique provides a different trade off between the number of checkpoints per time interval made and the amount of overhead they add to the application. Which approach to chose depends on the application. Applications with low *Mean Time Between Failure (MTBF)* will be best suited with uncoordinated Checkpointing with message logging which ensures application progress despite the low MTBF due to the high amount of checkpoints per time interval made and will therefore gladly accept the significant overhead of this solution. Applications with high MTBF will better chose Checkpointing at natural synchronization barriers, which doesn't add much overhead.

All described Checkpointing techniques have a common limitation. They require full control over the execution state of all objects interacting with an application using these Checkpointing techniques. For instance if a failure occurs the whole system has to be rolled back to a consistent state, i.e. a previously created checkpoint. It would not be sufficient to only roll back the failed object and resume its processing because that could let all other objects interacting with it in a different state than they had when the checkpoint was created. This could result into unwanted side effects and in unanticipated application behaviour.

In jCloudScale full execution control over applications using the jCloudScale middleware cannot be achieved without either modifying the used Java VM severely to gain this control or by requiring applications to implement a sophisticated rollback/Checkpointing *Application Programming Interface (API)*. Both possibilities are not desired, former would reduce jCloudScale's portability because of its dependency on a non-standard Java VM, and the latter would add a significant amount of additional implementation work for developers of applications that want to be fully compatible with jCloudScale. So jCloudScale's migration mechanism has to use a more basic Checkpointing technique, available for all Java VM implementations and transparent for third-party applications. The lowest common denominator for achieving this is by using the state between method invocations of COs as checkpoints, which equals Checkpointing at natural synchronization barriers. That is exactly what jCloudScale's migration process does. The migration of a CO is only started if there are no method invocations for this CO currently performed. For that to work, and to prevent the migration mechanism to interfere with jCloudScale's normal operation, as requested in this chapter's introduction, an effective synchronization and serialization mechanism is required. A simple approach would be to lock a CO entirely as soon as a method invocation is initiated. This would prevent parallel invocations of this CO and therefore guarantee each invoker – migration process included – exclusive access to the CO. The downside of this approach is, that it would also considerably harm jCloudScale's execution performance. Hence the migration mechanism must use a more fine grained locking technique, such as *Two-phase locking (2PL)*, which is also used for concurrency control in database systems [14]. The two types of locks provided by 2PL are *read-locks* and *write-locks*. Each thread trying to invoke a CO must obtain such a lock for this particular CO before allowed to proceed. Read-locks are shared-locks and therefore are always available for requesting threads. Accordingly parallel invocations are still possible when using read-locks and jCloudScale's execution performance isn't significantly affected by them. Write-locks on the other hand are exclusive locks, so there is only one for each CO available. When the migration process wants to migrate a CO it requests a write-lock for this CO. After all locks currently in use for this CO have been returned to the 2PL mechanism the write-lock is granted to the migration process, guaranteeing it exclusive access to the CO till it returns the write-lock again.

5.1.3 Rule Engines and *Complex Event Processing (CEP)*

Another requested feature of the migration mechanism, noted in the introduction and not discussed yet, is the policy-based triggering of migrations. Additionally it should be possible, for the user, to declare and alter these policies, so that they can be adapted to meet requirements set by the actual environment jCloudScale is used in. For practical reasons it is also necessary that these policies can be changed without the need to recompile jCloudScale. To satisfy these requirements it is necessary to decouple evaluation of these policies from jCloudScale and its migration mechanism. The separation of business rules from actual application code is not uncommon, because business rules tend to change more quickly than application code, and therefore would increase the application's maintenance costs accordingly, if not done so [71]. This separation can be achieved by using a rules engine.

By default jCloudScale's migration mechanism uses the Drools Rules Engine[1] for evaluating the rules, that specify when to trigger a CO migration. Due to the migration mechanism's architecture the rules engine used can be changed easily by the user, as noted in 5.3. Because the rules engine is a separate process with its own lifecycle it is also possible to change business rules without the need to restart jCloudScale, which should benefit large jCloudScale installations, that cannot be shut down easily.

A rules engine, as stated in the Drools Rules Engine's documentation, is a program, that delivers *Knowledge Representation and Reasoning (KRR)* functionality to the user. KRR is a research field, that tries to find ways for representing knowledge

[1]http://www.jboss.org/drools/, accessed 2013-11-28

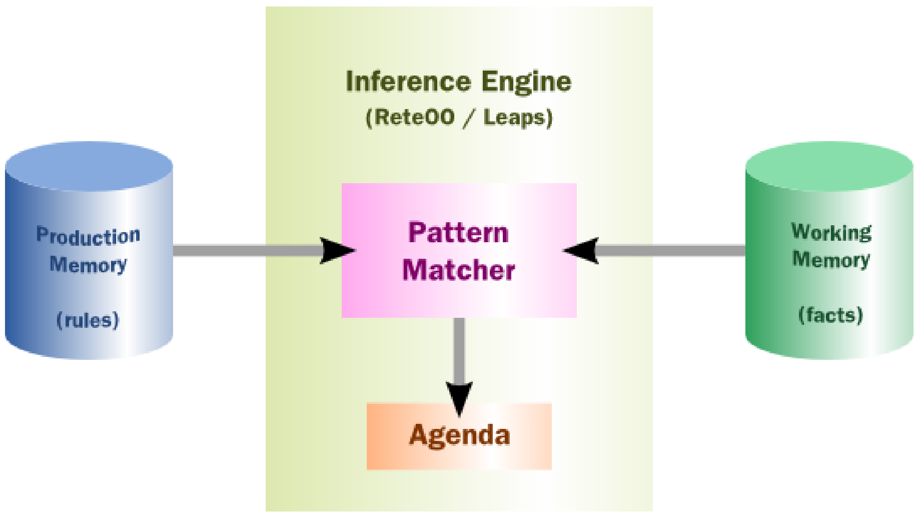

Figure 5.2: Overview of Drools Rules Engine's workflow (Source: Drools HP)

in a formal form, so that algorithms can utilize this knowledge to solve problems using deduction. This also explains why using rules to solve a problem is that different from using a common imperative programming language like Java. Rules are created by following a declarative approach, where one focuses on "what should be done", rather than on "how something is done". At a high level, a rules engine needs three things to work: *Ontology, rules* and *data*, where an Ontology describes knowledge within a domain in a formal way. The rules engine's task is to match data and facts (from the Ontology) against rules to infer conclusions which result into actions. This workflow is illustrated in Figure 5.2 A possible action in jCloudScale's migration system would be to trigger a migration, but that is not mandatory. It is also possible that actions themselves change data stored by the rules engine, resulting in other rules to be triggered. The triggering of rules through actions created by other rules is called *forward chaining* and supported by the Drools Rules Engine.

The actual Drools product used by the migration mechanism is called Drools Fusion, which is a rules engine that is also capably of *Complex Event Processing (CEP)*. As noted in the Drools documentation there is not a broadly accepted definition of CEP yet, but it can be said, that CEP focuses not on single events, but on event streams and applies techniques such as pattern recognition and event correlation on them to detect relationships between events. This enables rules to issue actions not only on occurrence of single events but also on group of events that, for instance, occur within a given time interval or are absent for a given period of time.

5.1.4 Choosing solutions with Automated Planning

Before an actual CO migration can be started two decisions must be made:

1. Which CO, out of all COs currently managed by jCloudScale, should be chosen for migration?
2. Which CH, out of all CHs available, should be the migration's destination host?

These questions can be answered by enumerating all possible answers, rating them by a predefined scheme and finally choosing highest rated answer. This proceeding is also called *Automated Planning* an can be executed by so called planning engines. jCloudScale's migration mechanism uses a planning engine to answer the questions stated above. By default jCloudScale uses the Drools OptaPlanner[2] engine as planing engine, which can be easily changed by the user, as described in Section 5.3.

As noted in OptaPlanner's documentation Automated Planning tries to find efficient solutions for all kinds of planning problems in an automated way. A planning problem describes the need of allocating a limited set of constrained resources to fulfil a task. A planning engine uses scores to rate possible planning solutions and compare them with each other, and sophisticated heuristics to reduce the time needed to find a good solution for a given planning problem. Solutions can be categorized, into:

- *Possible solutions:* Any solution is a possible solution, whether or not it has a high or low score. Many possible solutions are worthless. For instance, choosing the CH, managing the CO to be migrated, also as destination host is a possible solution, although it would be pointless to do so.
- *Feasible solutions:* Every feasible solution is also a possible solution. In general the number of feasible solutions tends to be relative to that of possible solutions. To be feasible a solution must not break a planning problem's hard constraints.
- *Optimal solutions:* An optimal solution is a solution with the best score possible. Each planning problem has at least one optimal solution. It is not required for an optimal solution to be feasible, hence it is possible that an optimal solution breaks a planning problem's hard constraints.

[2]http://www.optaplanner.org/, accessed 2013-11-28

- *Best solutions:* A solution with the highest score, found by the planning engine in a given amount of time. If the planning engine can search for a solution for an infinite amount of time the best solution is an optimal solution.

The score of a solution is calculated based on the number of constraints it breaks. Constraints are defined by the planning problem to solve. In Drools OptaPlanner, there are two kind of constraints:

- *Hard constraints:* Hard constraints are most important constraints. The planning solution chosen to solve the particular planning problem must not break any hard constraints, unless not possible otherwise.
- *Soft constraints:* Soft constraints are far less important than hard constraints. Hard constraints always outweigh soft constraints, meaning, that a solution breaking fewer hard constraints than another solution is always considered "better", despite the number of soft constraints broken by it. Usually a planning problem defines more soft constraints than hard constraints.

jCloudScale's migration mechanism uses Java classes for calculating a problem solution's score. It is up to the concrete implementation of these classes to decide when a solution breaks a constraint and if this constraint is a hard or a soft one and to rate the solution accordingly. Therefore it is imperative for the user to write her own calculator classes, so that they are in line with the inherent constraints of the executed jCloudScale application.

5.2 Migrating Cloud Objects in jCloudScale

After discussing the general concepts and technologies used by the migration mechanism in the previous section, in the following an actual CO migration will be described in detail. A CO migration can be logically separated into three parts: the triggering of the migration, finding an appropriate solution for the migration's planning problem, and eventually migrating the CO to its new CH. All parts, except for the second one, are mandatory. The search for a migration solution can be omitted, if the CO to migrate and its destination CH are already specified when triggering the migration.

Figure 5.3 illustrates the workflow of a CO migration, using UML activity diagram notation. Each logical part of the migration process is recognizable in the activity diagram, including the optionality of the second part. The first and second part of the process are entirely executed on jCloudScale's client-side, while the third part's execution is distributed between the client-side and the source and destination CHs of the migration. Further details on each step of the migration process will be given in the appropriate sections.

To provide a better illustration of the migration mechanism in general and its logical parts in particular a small example application will be introduced. As the focus of this section lies on jCloudScale's migration mechanism and to keep code listings brief non-essential code will be omitted from them. The example application is called `CloudWorker` and consists of a single CO named identically. It is shown in Listing 5.1. The `CloudWorker` CO connects to a server with IP address 10.0.1.2 on port 12345, when created and disconnects again, when destroyed. Its single purpose is to process workload data, provided by the caller, when invoking the `processWorkload()` method. This method does three things, first it sends metadata about the workload to the server, the `CloudWorker` CO is connected to, then it processes the workload, and finally injects an event, storing the methods execution time, into jCloudScale's event system. The `RespondTimeEvent` class is a custom event and part of the `CloudWorker` application. It is a direct subclass of jCloudScale's abstract `Event` class (see Figure 4.2) and therefore inherits the `id` and `timestamp` variables, which are initialized by the event's constructor.

5.2.1 Triggering a migration

As illustrated in Figure 5.3 and already mentioned earlier, the task of the migration mechanism's first logical part is to decide when to trigger CO migrations. It accomplishes this by utilizing jCloudScale's event system, described in 4.1.2, and an external rules engine, described in 5.1.3. Whenever the triggering mechanism receives an event object it forwards it to the rules engine, where it is used to evaluate user-generated business rules. These rules can then be used to trigger a CO migration, using a callback interface, provided by the migration mechanism. The callback interface is shown in Listing 5.2. It offers five different methods for invoking a CO migration, to give the user greater control over the migration's outcome. Each migration method accepts an optional `MigrationReason` object as parameter, which will be stored together with some statistical data about the migration, to enable traceability. This information is stored by jCloudScale and can be evaluated by applications. The `initTrigger` method is meant for custom implementations of the callback interface and is invoked by jCloudScale when initializing the callback interface.

The `CloudWorker` example provides its own rules file, containing one trigger-

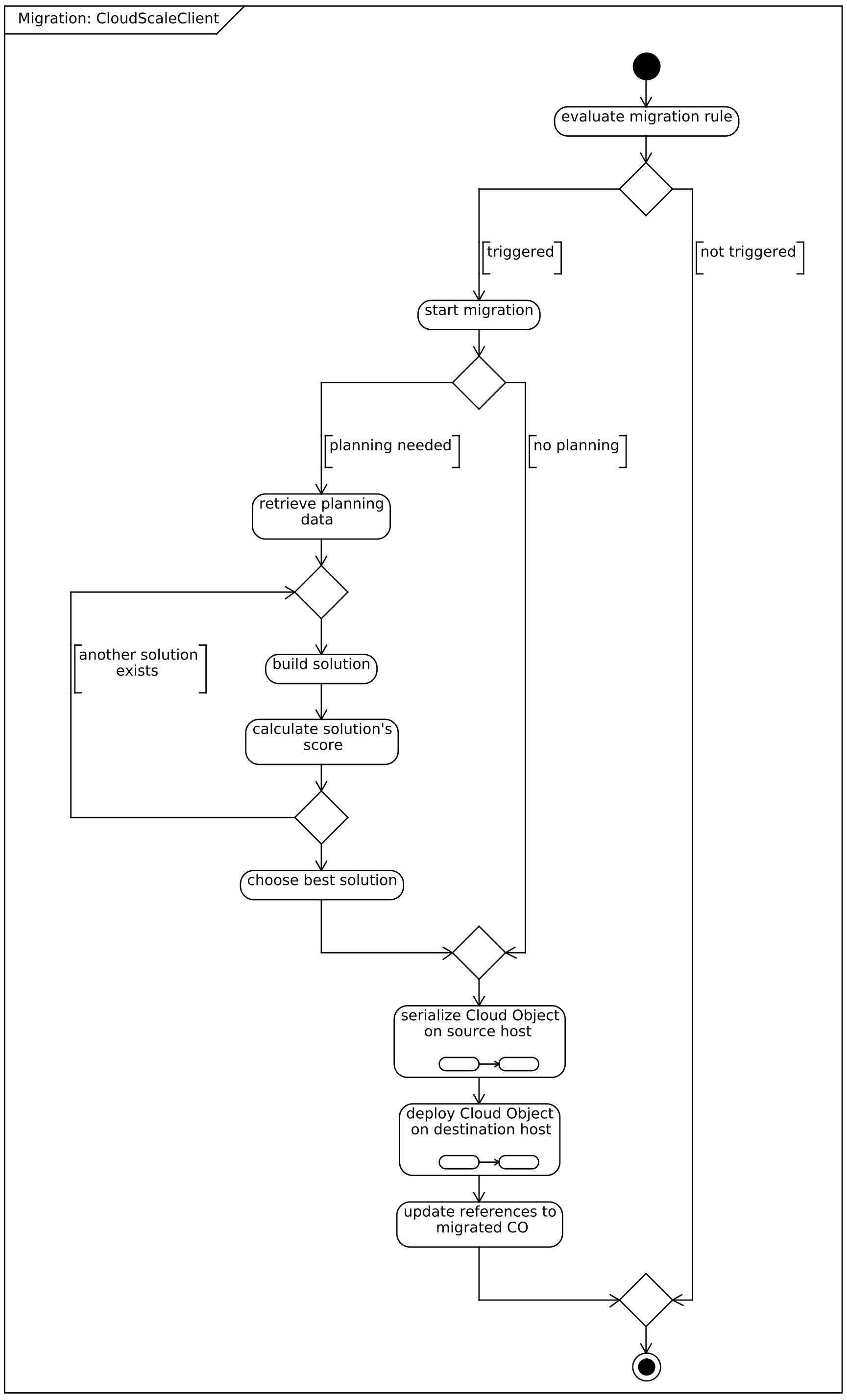

Figure 5.3: Workflow of a migration on jCloudScale's client side

```java
@CloudObject
public class CloudWorker extends Worker implements Serializable {

  @MigrationTransient    private Socket outbound;
  @EventSink             private IEventSink eventQueue;
  @CloudObjectId         private UUID coId;

  public CloudWorker() throws Exception {
    setupConnection();
  }

  @PostMigration
  private void setupConnection() throws Exception {
    this.outbound = SocketFactory.getDefault()
                    .createSocket("10.0.1.2", 12345);
  }

  @PreMigration
  private void shutdownConnection() throws Exception {
    this.outbound.close();
  }

  public void processWorkload(@ByValueParameter Workload workload)
      throws JMSException {
    long startTime = System.currentTimeMillis();

    sendData(outbound, workload.getMetadata());
    processData(workload.getData());

    long respTime = startTime - System.currentTimeMillis();
    eventQueue.trigger(new RespondTimeEvent(respTime, coId));
  }

  @DestructCloudObject
  public void destroy() throws Exception {
    shutdownConnection();
  }
}
```

Listing 5.1: The `CloudWorker` example

ing rule, which is shown in Listing 5.3. This rule enforces `CloudWorker`'s *Service Level Agreement (SLA)*, by evaluating the `RespondTimeEvents`, created by each of its instances. The rule collects the five most recent events, received by each instance, to calculate the average respond time of each instance's `processWorkload()` method. If the calculated respond time for an instance exceeds 15000ms

```java
public interface IMigrationTrigger {

  public UUID migrateFromHost(UUID sourceHostId,
                 MigrationReason reason);

  public UUID migrateObject(UUID cloudObjectId,
                 MigrationReason reason);

  public UUID migrateObjectToHost(UUID cloudObjectId,
                 UUID destinationHostId,
                 MigrationReason reason);

  public UUID migrateObjectToNewHost(UUID cloudObjectId,
                 MigrationReason reason);

  public UUID migrateFromHostToNewHost(UUID sourceHostId,
                 MigrationReason reason);

  public void initTrigger(CloudScaleConfiguration config,
                 MigrationConfiguration mconfig);
}
```

Listing 5.2: All migration methods supported by jCloudScale

a migration for this `CloudWorker` instance is triggered. The rule uses the callback's `migrateObject()` method, to enforce the migration of the particular `CloudWorker` CO that violated the SLA.

5.2.2 Finding an appropriate migration solution

After a migration has been triggered the migration mechanism's second logical part takes command. This part's task is to gather all information necessary to start an actual CO migration. A migration requires three pieces of information to take place: the CO to migrate, the source CH the CO is currently managed by, and the destination CH to which the CO should be migrated. This information will further be referenced to as migration solution. Depending on the specific migration method invoked by the triggering mechanism none, some, or all pieces of the migration solution are already known. Hence this step of the migration process is skipped, if the invoked migration method provided a complete migration solution. If the provided migration solution is incomplete the migration mechanism uses Automated Planning, as described in 5.1.4, to retrieve the missing pieces. The migration method, invoked by the triggering mechanism, also defines the number of possible migration solutions the planning engine can choose from, to retrieve the migration's optimal solution. For instance, the `migrateObject()` method,

```
package cloudscale.migration.rules

import cloudscale.migration.statistic.MigrationReason
import cloudscale.migration.object.*

global cloudscale.migration.IMigrationTrigger callback

declare RespondTimeEvent
  @role( event )
  @timestamp( timestamp )
end

rule "CloudWorker SLA"
  when
    event : RespondTimeEvent( $coId : objectId)
    Number( longValue > 15000 ) from accumulate(
        RespondTimeEvent( objectId == $coId, $time : respondTime )
        over window:length( 5 ), average( $time ) )
  then
    MigrationReason reason =
      new MigrationReason( "SLA001", "SLA violation", event );
    callback.migrateObject( event.getObjectId(), reason );
end
```

Listing 5.3: Trigger rule for `CloudWorker`

invoked by `CloudWorker`'s triggering rule, provides the identity of the CO to migrate, i.e. the number of possible migration solutions is limited to solutions where this CO will be migrated to another CH.

To choose the optimal solution the planning engine has to calculate a score for each solution, so that they can be rated and compared. The score calculation is based on statistical data of the solutions's CO and CHs. This data is provided through the migration mechanism's `MigrationDataCollector` class, which gathers this information from events received through jCloudScale's event system. These events are generated by various event aspects, as shown in Figure 5.4. These event aspects are part of jCloudScale and issue event objects either periodically or on occurrence of specific events. For instance, state events, which contain information about a CH's utilization are issued periodically, while e.g. event objects about a CO's deployment and destruction are issued when these specific events happen.

The `CloudWorker` example provides its own score calculator class, shown in Listing 5.4. The planning engine first uses the calculator to initialize the score of each solution and afterwards to calculate the solution's final score. As shown in the

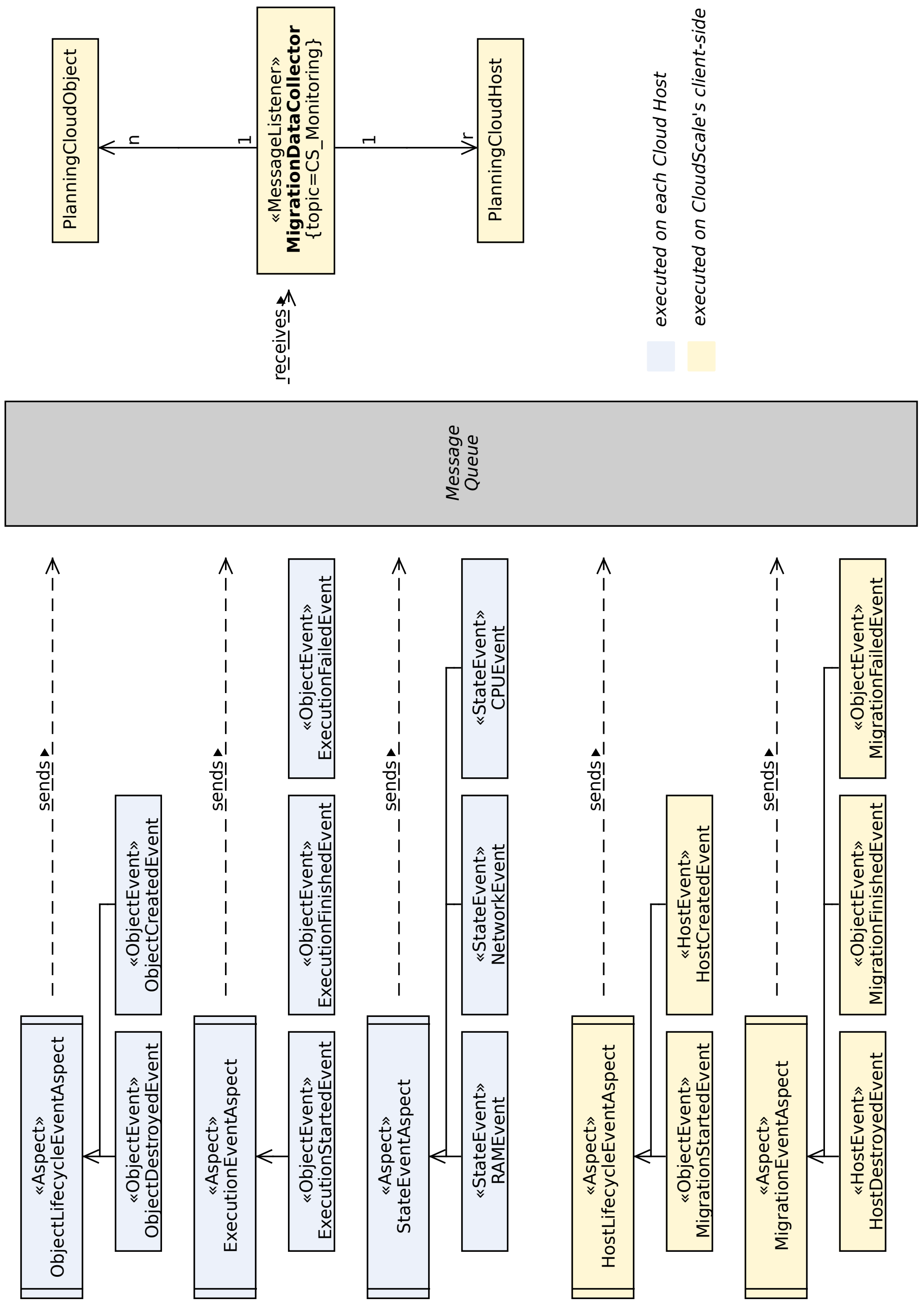

Figure 5.4: jCloudScale's eventing is used to generate planning data

calculators listing each solution is initialized with a default score of (0, -1000), if no score is set. If a score is already set, the solutions final score is calculated, using the number of COs managed by the solution's CH as hard constraint and the number of times the solution's CO was invoked as soft constraint. Therefore the `CloudWorkerScoreCalculator` favours migration solutions which feature a destination CH that manages as few COs as possible and a migration CO that was invoked as quite often. Because the migration CO of all possible migration solutions for the `CloudWorker` example is already set by the `migrateObject()` method the optimal solution will be the one which features the destination CH which manages the fewest amount of COs.

```java
public class CloudWorkerScoreCalculator implements
        SimpleScoreCalculator<DroolsMigrationSolution> {

  @Override
  public Score<HardAndSoftScore> calculateScore(
        DroolsMigrationSolution solution) {
    DroolsMigrationPlanningEntity entity =
                solution.getPlanningEntity();

    if (solution.getScore() == null)
      return DefaultHardAndSoftScore.valueOf(0, -1000);

    int hardScore = 0;
    if (entity.getCloudHost() != null) {
      PlanningCloudHost host = entity.getCloudHost();
      hardScore -= host.getManagedCloudObjects() * 10;
    }

    int softScore = -1000;
    if (entity.getCloudObject() != null) {
      PlanningCloudObject object = entity.getCloudObject();
      softScore += object.getTimesInvocated() * 10;
    }

    return DefaultHardAndSoftScore.valueOf(hardScore, softScore);
  }
}
```

Listing 5.4: Solution calculator for `CloudWorker`

5.2.3 Migrating a Cloud Object

After a migration was triggered and a valid migration solution was found the actual migration of a CO can be executed. As described in 5.2 the execution of the third logical part of the migration mechanism is distributed across the jCloudScale-Client and the migration's source and destination CHs. It is the task of the jCloudScale-Client to orchestrate the whole process, working as a middleman between the source CH and the destination CH, to ensure successful migration of the CO. The class that implements the jCloudScale-Client's task during a CO migration is called `MigrationExecutor`.

Before the CO can actually be migrated access to it must be restricted, so that other threads are unable to interfere. This is done using 2PL technique, as described in 5.1.2. While the executor has exclusive access to the CO, invocation requests of other threads are not rejected but queued and resumed again after the executor has returned its lock. Hence a CO's migration is fully transparent for other objects interacting with it. After that the `MigrationExecutor` instructs the CO's source host to serialize the CO and transmit the serialized data back to the executor. The serialization of Java objects is not without hurdles. There are several requirements[3] an object must fulfil to be serializable:

- the object's class or one of its superclasses must implement the `Serializable`-Interface
- the first superclass in its hierarchy not implementing this interface must have a no-argument constructor
- all member variables that can't be serialized must be marked transient
- the values of static fields will not be serialized

It's the developer's responsibility to ensure that a CO satisfies these requirements, so that it can be migrated. jCloudScale's migration mechanism tries to help developers to do so by providing four custom Java annotations. These annotations, when used inside a CO, are hints for the migration mechanism on how to serialize and deserialize the object. After the executor has received the serialized CO from the source CH the executor forwards it to the destination CH, where it is deserialized and initialized. The whole workflow of serialization and deserialization, including the handling of the migration specific annotations is illustrated in Figure 5.5.

Processing of the migration-specific annotations is done right before serialization respectively after deserialization. When the migration mechanism searches for

[3]http://www.oracle.com/technetwork/articles/java/javaserial-1536170.html, accessed 2013-11-27

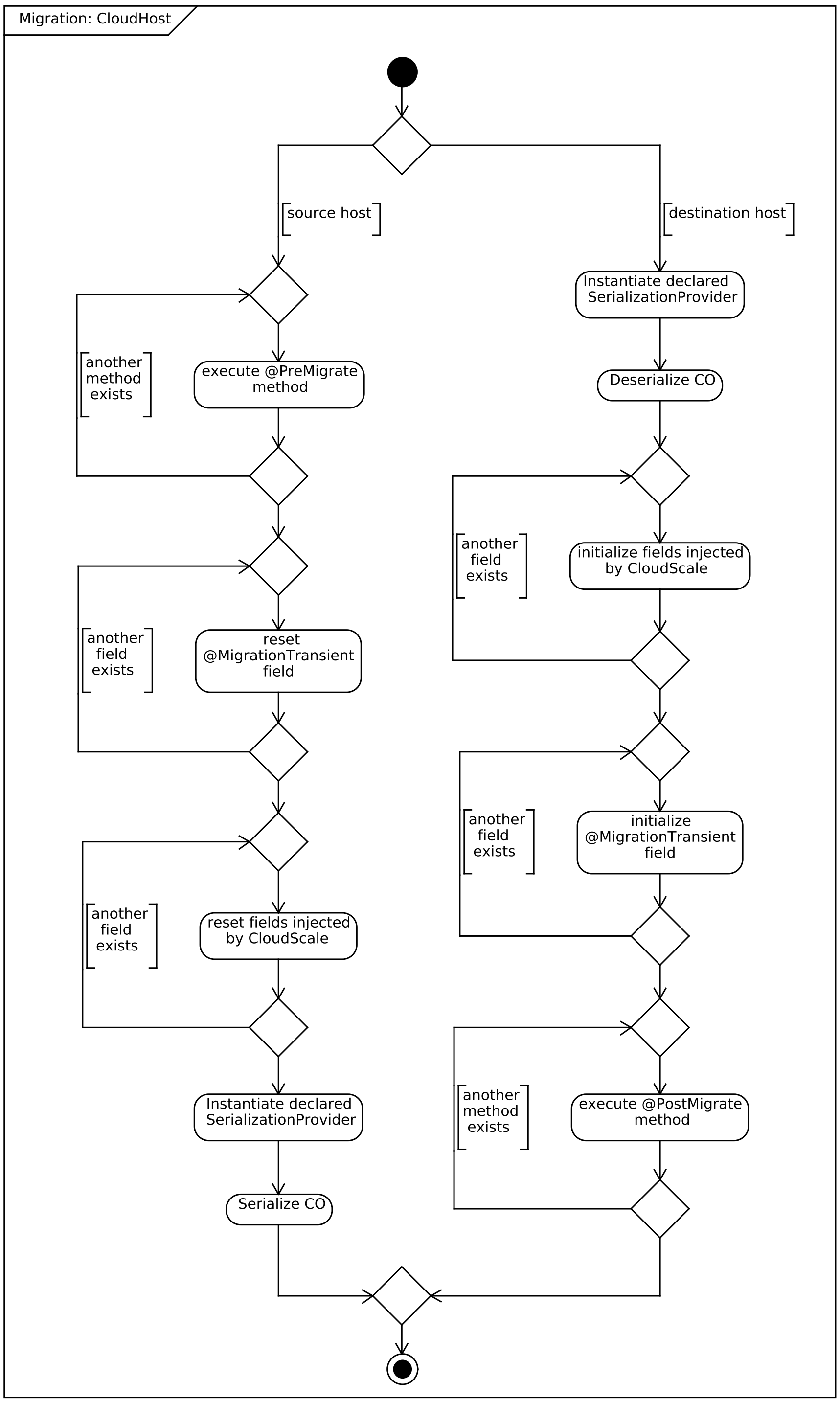
Migration: CloudHost
source host
destination host
execute @PreMigrate method
another method exists
reset @MigrationTransient field
another field exists
reset fields injected by CloudScale
another field exists
Instantiate declared SerializationProvider
Serialize CO
Instantiate declared SerializationProvider
Deserialize CO
initialize fields injected by CloudScale
another field exists
initialize @MigrationTransient field
another field exists
execute @PostMigrate method
another method exists

Figure 5.5: Workflow of a migration on a *Cloud Host (CH)*

these annotations, it starts at the top of the CO's class hierarchy, hence annotated variables or methods in the CO's super classes are handled first. The purpose of the migration-specific annotations are:

- *@MigrationTransient:* Can be used to declare member variables of a CO as not serializable. Before a migration takes place these variables will be set to null value, to not hinder a CO's serialization. The annotation accepts an optional parameter, which will be used as initialization value for the variable, when the CO is deserialized again.
- *@PreMigrate/@PostMigrate:* These annotations are used to mark methods of a CO to be executed prior its serialization respectively after its deserialization. This enables COs that use external resources to close them properly before they are migrated, and reopen them again when the migration has finished.
- *@SerializationProvider:* If the previously described annotations are not sufficient to enable serialization and deserialization of a CO, this annotation can be used. It enables developers to write their custom serialization and deserialization mechanisms and specify which one of them should be used for which types of COs.

Although the migration-specific annotations help developers to reduce the reasons why a particular CO can not be serialized and, therefore, migrated there might still be circumstances that forbid to migrate a CO. To address these occasions the migration mechanism provides a *@NoMigrate* annotation. COs annotated in this way will not be taken into consideration by the `MigrationExecutor` when creating possible migration solutions. Hence, they will never be migrated automatically by the migration mechanism.

After serialization of the CO on the source CH, and its deserialization on the destination CH, there is one step left for the migration to be completed. For a CO migration to be fully transparent for other objects interacting with the CO, all references to the CO must be preserved, but redirected to its new managing CH. Because jCloudScale uses AOP to intercept method invocations of COs to redirect these invocations into the cloud, the `MigrationExecutor` can update these references at a central point inside jCloudScale and release the write-lock it holds for the migrated CO afterwards.

The `CloudWorker` CO in Listing 5.1 fulfils all serialization requirements listed above. It implements the `Serializable` interface and uses migration-specific annotations to handle its non-serializable member variable named `outbound`. Before the CO is serialized its *@PreMigrate*-annotated method `shutdownConnection()` is executed and its `outbound`, `eventQueue`, and `coId` variables are set to `null`. After that the `CloudWorker` CO can be serialized. When deserialized at the destination CH the CO's `eventQueue` and `coId` variables are initialized and its *@PostMigrate*-annotated `setupConnection()` method is executed, which creates a new `Socket` object and assigns it to the `outbound` variable. The `CloudWorker` CO has been migrated successfully.

5.3 Architectural overview of the migration mechanism

As stated in the introduction of this chapter, in this section a complete overview of the migration mechanism's architecture will be given. The UML class diagram notation is used to picture this overview. For clarity, components not part of the core process, like rules and planning engines, message queue and CHs, are omitted from the diagram. The class diagram is shown in Figure 5.6.

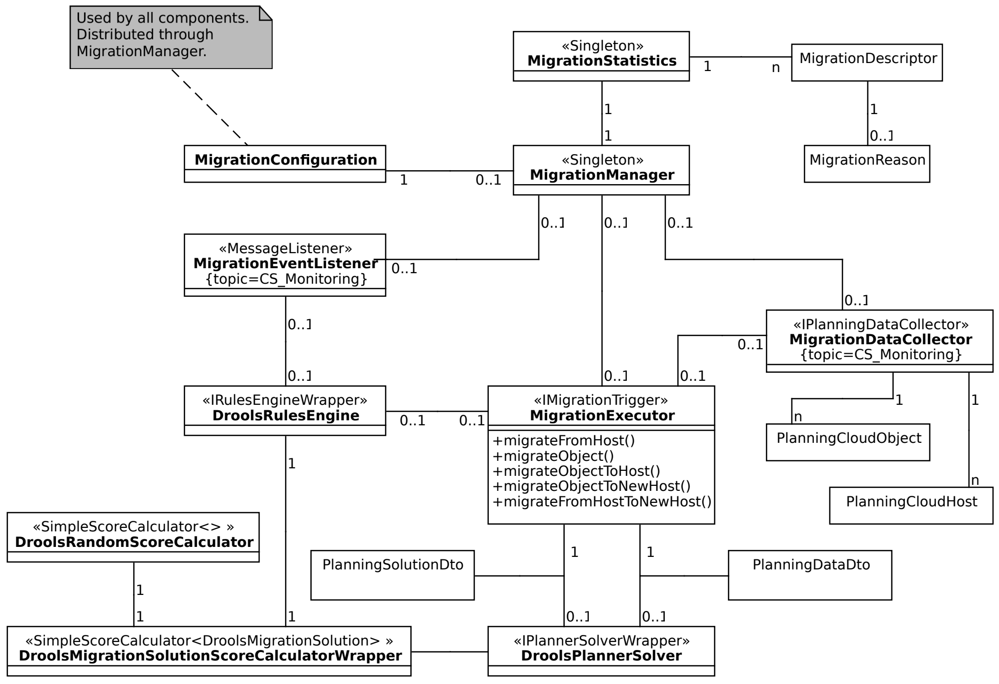
Used by all components. Distributed through MigrationManager.
«Singleton» MigrationStatistics
MigrationDescriptor
MigrationReason
MigrationConfiguration
«Singleton» MigrationManager
«MessageListener» MigrationEventListener {topic=CS_Monitoring}
«IPlanningDataCollector» MigrationDataCollector {topic=CS_Monitoring}
«IRulesEngineWrapper» DroolsRulesEngine
«IMigrationTrigger» MigrationExecutor
+migrateFromHost()
+migrateObject()
+migrateObjectToHost()
+migrateObjectToNewHost()
+migrateFromHostToNewHost()
PlanningCloudObject
PlanningCloudHost
«SimpleScoreCalculator<> » DroolsRandomScoreCalculator
PlanningSolutionDto
PlanningDataDto
«SimpleScoreCalculator<DroolsMigrationSolution> » DroolsMigrationSolutionScoreCalculatorWrapper
«IPlannerSolverWrapper» DroolsPlannerSolver
1
n
0..1

Figure 5.6: Overview of all classes involved in CO migration in jCloudScale

As shown in the class diagram, the `MigrationExecutor` is the central point of the mechanism and orchestrates the migration once initiated by the rules engine.

The executor uses two services, first the `MigrationDataCollector` to retrieve planning data and second the `DroolsPlannerSolver` to determine the optimal migration solution. Both services are accessed through interfaces to decouple the executor from concrete implementations of these services. The same approach is applied when the `MigrationEventListener` accesses the rules engine to hand over received events. The use of interfaces was deliberately chosen to make it easy for developers to use their own implementations of those services.

The wrapper pattern [37] is used to decouple a solution score calculator class, that is used to rate a possible planning solution, from the planning engine's static configuration, to allow changing calculator classes during runtime. The `Drools-MigrationSolutionScoreCalculatorWrapper` acts as a wrapper for the specific calculator class that is configured in the `MigrationConfiguration`. The migration mechanism comes with two score calculator classes: `DroolsRan-domScoreCalculator` and `DroolsSimpleScoreCalculator`. Former generates a random score for each planning solution, while latter rates the planning solutions highest, that features the CO most often invoked and the CH managing the fewest COs.

The `MigrationManager` is a single interface to the migration subsystem. To some degree it implements the facade pattern described by Gamma et al. in [37]. Its main purpose is to start and stop all parts of the migration system in correct order and provide a common configuration to them. The common configuration is called `MigrationConfiguration` and stores migration-specific parameters like the location of rules files and message queue topics to use. An instance of the `MigrationStatistics` class is also provided through the manager. It stores information about the outcome of each migration executed alongside with additional information about the participating CHs and CO.

CHAPTER 6

Evaluation

The previous chapter described in detail the inner workings of jCloudScale's newly developed migration mechanism and how it can be used and configured by application developers. In the following chapter focus will be on evaluating the migration mechanism to examine its behaviour under different workloads and therefore its impact on jCloudScale's overall performance.

This chapter is structured into three sections. First the evaluation scenario, under which all tests where conducted, will be described. Then a detailed analysis of the migration mechanism's components, regarding their performance and memory consumption, will be made. Afterwards the chapter will be concluded by summarizing and commenting the evaluation results from the analysis section.

6.1 Evaluation Scenario

As stated in Chapter 4, jCloudScale is a middleware for distributed applications that run on top of *Infrastructure-as-a-Service (IaaS)* clouds. Although jCloudScale supports a *local-mode*, where all of its parts are executed on one host, jCloudScale is meant to be executed in a distributed environment. Therefore evaluation tests are in general conducted using *Distributed Systems Group (DSG)*'s OpenStack IaaS cloud for executing jCloudScale server instances and a dedicated *Message Queue (MQ)* server, that handles communication between the server instances and the jCloudScale client. The jCloudScale client is executed on a separated host, outside the OpenStack cloud, connected to it through a *Virtual Private Network (VPN)* connection.

A detailed description of the execution environment is given below. An overview of all available *Virtual Machine (VM)* flavors in DSG's OpenStack cloud is given in 6.1.1. The particular VM flavor used for a test will be given in the test's description. Unless otherwise stated all tests have been repeated 100 times and averaged, to reduce unpredictable fluctuations. Measurement of time and memory consumption during tests was done using Java's standard *Application Programming Interface (API)*, including classes in package `javax.management`. During test execution no other programs have been executed to prevent influence of the test results.

Environment for jCloudScale-Client

The jCloudScale-Client part was executed on a 64-bit GNU/Linux distribution using kernel version 3.12.6 and OpenJDK version 1.7.0_51. The test machine used was an Intel Core i5-3320M CPU with 2.6 GHz, which supports parallel execution of 2 processes or 4 threads, and 8096 MB RAM. The migration-enabled jCloudScale version used was 0.3.1. Further the Drools Fusion and Planner distributions in version 5.4.0 were used as rules respective planner engines.

Environment for Message Queue-Server

The dedicated MQ server was executed inside DSG's OpenStack cloud, using a VM of flavor *m1.small*. The VM was running Ubuntu 12.04.1 LTS 64-bit with Oracle's Java SE runtime version 1.7.0_17. Apache ActiveMQ version 5.5.0 was used as MQ server. According to the system's `/proc/cpuinfo` file the CPU used was an Intel Core2 Duo T7700 with 2.4 GHz.

Environment for jCloudScale-Server

The jCloudScale-Server instances where also executed inside DSG's OpenStack cloud. Which OpenStack flavor was used for each host will be stated separately in the description of each test. All server instances used Ubuntu 13.10 64-bit with OpenJDK version 1.7.0_25 and migration-enabled jCloudScale version 0.3.1. As for the MQ server, the CPU of each VM was also an Intel Core2 Duo T7700 with 2.4 GHz.

6.1.1 Flavors in DSG's OpenStack cloud

OpenStack is an IaaS cloud platform that provides computing resources, through VMs, on demand. To conduct this thesis' evaluation DSG dedicated a quota of the overall resources available in it's OpenStack cloud. This quota covered the ability to run a maximum of 10 VM instances in parallel. Further the quota provided a total amount of 51200 MB RAM and 20 virtual CPU cores that could be utilized. When started, each VM instance claims a certain fraction of these resources for itself. The exact configuration of resources claimed by a VM is called “flavor”. There are 13 different flavors available in DSG’s OpenStack installation. They are listed in table 6.1.

Flavor Name	Virtual CPUs	Memory (RAM)	Disk size
m1.tiny	1	512 MB	0 GB
m1.micro	1	960 MB	40 GB
m1.small	1	1 920 MB	60 GB
m1.medium	2	3 750 MB	80 GB
m2.medium	3	5 760 MB	80 GB
m1.large	4	7 680 MB	120 GB
m1.xlarge	8	15 360 MB	200 GB
m1.2xlarge	16	30 720 MB	70 GB
w1.tiny	1	960 MB	25 GB
w1.small	2	1 920 MB	30 GB
w1.medium	8	3 750 MB	40 GB
w1.large	4	7 680 MB	30 GB
w1.xlarge	8	15 360 MB	60 GB

Table 6.1: Overview of available VM-flavors at DSG’s OpenStack cloud

6.2 Detailed Analysis

The following section will examine the performance and scalability of the migration process’ discrete parts, meaning the `MigrationExecutor`, the `Migration-EventListener` and the `MigrationDataCollector`. Further it is evaluated if a jCloudScale-Server using Java Instrumentation has reduced execution performance in comparison to a server not using instrumentation.

All evaluation tests where conducted using one of the two Cloud Objects, shown in Listing 6.1, respectively Listing 6.2. The object `COEvalSize`, from Listing 6.1, provides methods for creating specific amounts of random data, either as primitive byte array or as linked list storing byte objects. By creating a specific payload size one can control the *Cloud Object (CO)*’s memory footprint and serialized size, thus this CO is used to evaluate the performance of jCloudScale’s server instances, in terms of code execution and memory management. Further it is used to measure the impact the size of a CO has on the time needed to migrate this object.

```java
@CloudObject
public class COEvalSize implements Serializable {

  private byte[] primitivePayload;
  private List<Byte> complexPayload;

  @CloudObjectId private UUID id;

  @ByValueParameter
  public UUID getId() { return id; }

  public void createPrimitivePayload(int size) {
    this.complexPayload = null;
    this.primitivePayload = new byte[size];
    new Random().nextBytes(this.primitivePayload);
  }

  public void createComplexPayload(int size) {
    this.primitivePayload = null;
    this.complexPayload = new LinkedList<>();
    Random rng = new Random();
    for (int i = 0; i < size; i++) {
      byte[] b = new byte[1];
      rng.nextBytes(b);
      this.complexPayload.add(b[0]);
    }
  }

  public long getSize() {
    if (primitivePayload != null)
      return CloudScaleServerInstrumentation
                  .getObjectSize(primitivePayload);
    else if (complexPayload != null)
      return CloudScaleServerInstrumentation
                  .getObjectSize(complexPayload);
    return 0;
  }

  @DestructCloudObject
  public void destroy() {
    this.primitivePayload = null;
    this.complexPayload = null;
  }
}
```

Listing 6.1: Evaluation Cloud Object 1

The second CO, shown in Listing 6.2, is named `COEvalEventSender`. It provides two methods for triggering the injection of jCloudScale event objects into jCloudScale's monitoring message topic. Consumers of this topic are the `MigrationEventListener` and `MigrationDataCollector`. Therefore, this CO is used to evaluate how these components scale to different message loads on the monitoring topic. The `sendEvent` method injects a single custom migration event object into the MQ, when called, which will be picked up and processed by the `MigrationEventListener`. The `sendPlEvents` method is more complicated. When called it repeatedly injects various event objects into the MQ. It does this at a fixed rate and for a curtain amount of time and thus simulates an event stream similar to that created by jCloudScale server instances during normal execution of a jCloudScale application. These events are picked up and processed to planning data by the `MigrationDataCollector`.

```java
@CloudObject
public class COEvalEventSender implements Serializable {

  @CloudObjectId private UUID id;
  @EventSink     private IEventSink sink;
  final Queue<UUID> idPool = new ConcurrentLinkedQueue<>();

  @ByValueParameter public UUID getId() { return id; }

  public void sendEvent(@ByValueParameter MyMigrationEvent event)
      throws JMSException {
    sink.trigger(event);
  }

  public void sendPlEvents(int eventsPerSecond, long duration,
      @ByValueParameter final Event[] events)
        throws InterruptedException, JMSException {
    final Random rng = new Random();
    int fallbackIdx = 0;
    while (!(events[fallbackIdx] instanceof ExecutionStartedEvent))
      fallbackIdx++;
    final ExecutionStartedEvent evStart =
              (ExecutionStartedEvent) events[fallbackIdx];

    Timer timer = new Timer();
    timer.scheduleAtFixedRate(new TimerTask() {
      public void run() {
        UUID reqId = null, eReqId = null;
        try {
          Event e = events[rng.nextInt(events.length)];
          if (e instanceof ExecutionStartedEvent) {
            reqId = UUID.randomUUID();
            ((ExecutionStartedEvent) e).setRequestId(reqId);
```

```
              } else if (e instanceof ExecutionFailedEvent) {
                eReqId = idPool.poll();
                if (eReqId != null) {
                  ((ExecutionFailedEvent) e).setRequestId(eReqId);
                } else {
                  reqId = UUID.randomUUID();
                  evStart.setRequestId(reqId);
                  e = evStart;
                }
              } else if (e instanceof ExecutionFinishedEvent) {
                eReqId = idPool.poll();
                if (eReqId != null) {
                 ((ExecutionFinishedEvent) e).setRequestId(eReqId);
                } else {
                  reqId = UUID.randomUUID();
                  evStart.setRequestId(reqId);
                  e = evStart;
                }
              }

              sink.trigger(e);
            } catch (JMSException ex) {
              ex.printStackTrace();
            } finally {
              if (reqId != null) idPool.add(reqId);
            }
        }
    }, 1000, 1000 / eventsPerSecond);
    TimeUnit.MILLISECONDS.sleep(duration);
    timer.cancel();
  }

  @DestructCloudObject void destroy() { this.idPool.clear(); }
}
```

Listing 6.2: Evaluation Cloud Object 2

The description of each evaluation test will state which evaluation CO was used, when conducted. Further it will be described which parameters where used when the CO's methods where invoked.

6.2.1 Evaluating Java Instrumentation

As described in Section 5.1.4, jCloudScale's migration mechanism has the ability to calculate an optimal solution for a migration request, that does not specify the actual CO to migrate or the migration's destination host. The necessary calculations are carried out by an automated planning engine that uses collected planning data of COs and *Cloud Hosts (CHs)*. Besides other metrics, planning data of COs includes the CO's size, which is regularly retrieved from the jCloudScale server instance managing the CO. For this to work the server instance has to be started with Java Instrumentation enabled, by using a Java agent library. An agent library is merely a wrapper for the Java Instrumentation object injected into it by the Java VM at startup. This subsection will evaluate if the use of Java Instrumentation degrades the execution performance a jCloudScale server instance. All evaluation tests in this subsection where executed using DSG's OpenStack cloud. The VM flavor used for the jCloudScale server instance was *m1.medium*, with a custom defined maximum Java memory heap size of 3300MB, to prevent `OutOfMemory` exceptions. Tests where conducted using the `COEvalSize` CO from Listing 6.1.

The left chart in Figure 6.1 shows an execution time comparison when filling a byte array sized from 1MB to 1GB with random data. Execution times for both server types are nearly identical, thus it does not appear as Java Instrumentation support degrades a Java VM's execution performance, when working with primitive type data. This assumption can also be verified, when applying a Student's t-test, as shown in Appendix B.1. The right chart of Figure 6.1 shows execution times of the CO's `getSize` method, also with the byte array sized from 1MB to 1GB and filled with random data. Because object size computation is only available if Java Instrumentation is enabled the method always returns a hardcoded value of -1 when disabled. As before it appears that Java Instrumentation support does not impact the server's execution performance. Further it is shown that computation of a CO's size is done in constant time, regardless of the CO's size.

The evaluation tests used for Figure 6.2 are similar to that used for Figure 6.1, but use a linked list of random byte objects as payload, instead of a byte array filled with primitive byte values. Because creating objects is in general much more time consuming than creating primitive data the payload's maximum size was limited to 128MB to reduce the tests overall execution time. The results shown in Figure 6.2 are similar to those from Figure 6.1, meaning that Java Instrumentation support does not appear to impact a jCloudScale server instance's performance. As before this assumption can also be verified, when applying a Student's t-test, as shown in Appendix B.2.

Due to these tests results all further evaluation tests will be conducted with Java Instrumentation enabled.

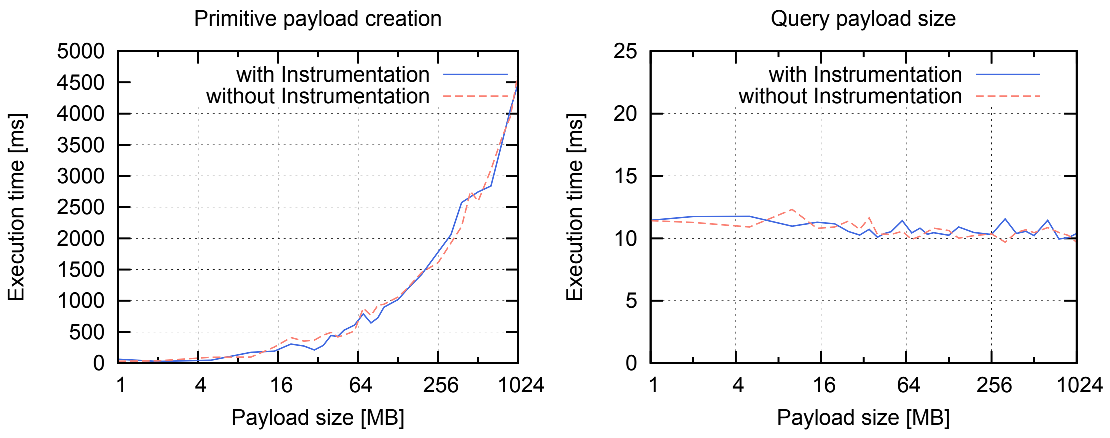

Figure 6.1: jCloudScale server computing primitive byte array

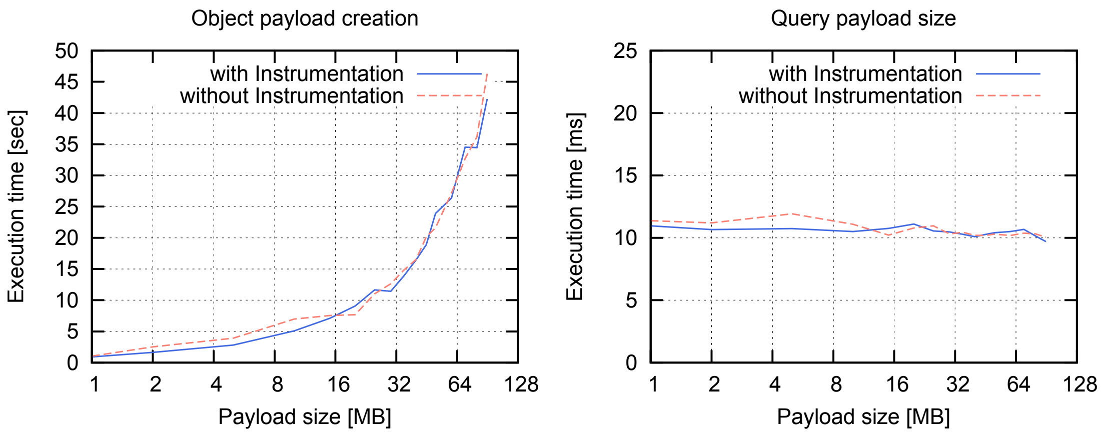

Figure 6.2: jCloudScale server computing linked list of byte objects

6.2.2 Evaluating the MigrationExecutor

The `MigrationExecutor` fulfils two main tasks: migrating a CO from its source host to its destination host and finding an optimal solution, if migration planning is necessary. Both tasks will be evaluated separately.

Figure 6.3 shows how plain (i.e., migration without prior planning) CO migration scales to the CO's size. The data shown was retrieved using DSG's OpenStack cloud with two jCloudScale server instances of flavor *m1.medium*, again with a custom defined maximum Java memory heap size of 3300MB, to prevent `OutOfMemory` exceptions. Both instances did not manage any other COs except the one being migrated. The `COEvalSize` CO from Listing 6.1 was used as evaluation CO.

The left chart of Figure 6.3 shows the duration of the migration, when a primitive type byte array is used as payload, while the right chart shows the duration when a linked list with byte objects is used. When discussing migration times one has to keep in mind, that the `MigrationExecutor` is executed on the jCloudScale-Client, which, in this evaluation setup, is not physically located inside DSG's network, but connected to it through a symmetric 10Mb/s VPN connection. This matters because when the serialized CO is migrated from its source host to its destination host it is piped through the `MigrationExecutor`. Nevertheless Figure 6.3 clearly shows that migrating a CO with references to a complex object tree takes considerable more time than migrating an object with a primitive type data structure. The evaluation showed further that migrating objects with a serialized size greater than 100MB cannot be handled by the MQ server without transmission errors.

Figure 6.4 shows how long it takes the planner to find a best migration solution within a given number of possible solutions. The left chart in Figure 6.4 gives a more detailed view of the results for up to 100000 possible solutions, as this will cover the problem range of most real-world migration planning problems. Both charts also show a speed comparison between Drools Planner's "brute-force" planning algorithm and its "first-fit" heuristic planning algorithm, with advantages for the "brute-force" algorithm, when the number of possible solutions exceeds 10000.

When comparing execution times from Figures 6.3 and 6.4 it can be said, that the time spend for finding a best migration solution does not add significant to the overall time needed when migrating a CO.

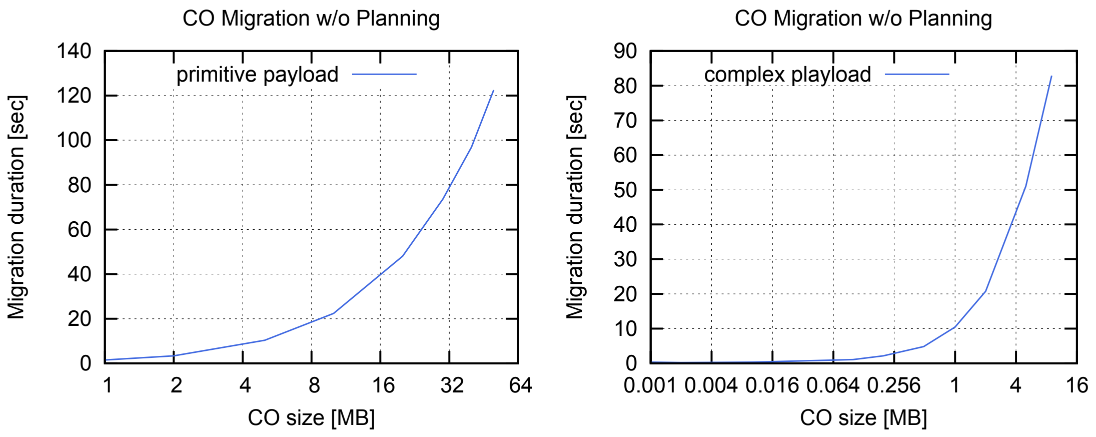

Figure 6.3: Migrating a CO without prior planning

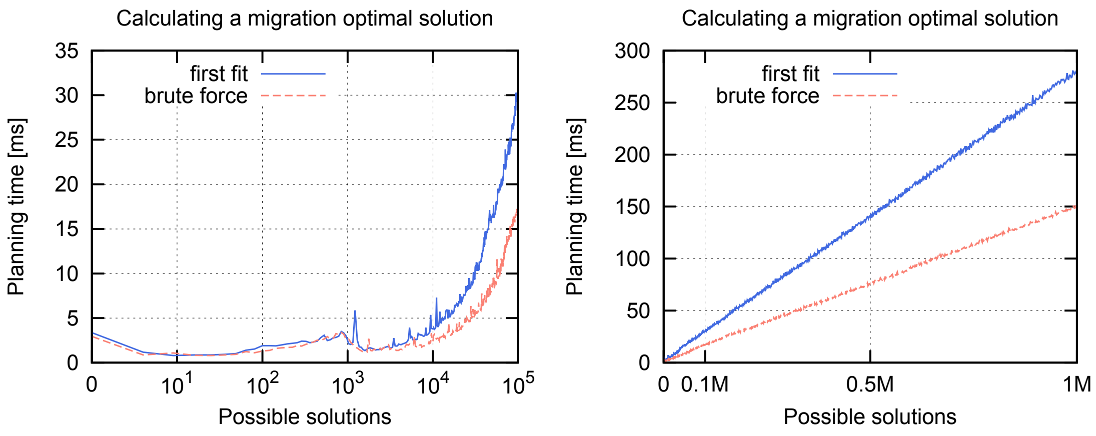

Figure 6.4: Searching a migration optimal solution

```
rule "MyMigrationEvent-$ruleNumber$"
  when
    event : MyMigrationEvent( customField == $customField$ )
  then
    MigrationReason reason =
        new MigrationReason($customField$, null, event);
    migration.migrateObject(event.getCloudObjectId(), reason);
end
```

Listing 6.3: Template rule

6.2.3 Evaluating the MigrationEventListener

The `MigrationEventListener` is basically an adapter for the rules engine to fit into jCloudScale's migration mechanism. It ensures that the rules engine is properly initialized, when the migration system starts, and forwards event objects, received through jCloudScale's monitoring topic, to it. Hence the overall performance and resource consumption of the `MigrationEventListener` is vastly determined by that of the rules engine. Therefore the following will evaluate how the rules engine performs, when loaded with different amounts of rule sets.

The `MigrationEventListener` and its rules engine are executed on the jCloudScale -Client and during the evaluation events where directly injected into the rules engine, without involvement of the MQ server. So no jCloudScale server instances where used in this evaluation. The rules engine was initialized with different amounts of rules, ranging from 1 rule to 20000 rules and an event, triggering one of the loaded rules, was injected afterwards. The rules files, containing the different amounts of rules, used during this evaluation where created programmatically by replicating the template rule shown in Listing 6.3. The template rule's variables `$ruleNumber$` and `$customField$` were replaced with unique values before a rule was added to a rules file.

Figure 6.5 displays the results of `MigrationEventListener`'s evaluation. The left chart shows the time needed by the rules engine to trigger a migration, by calling one of the methods defined in the `IMigrationTrigger` interface, after an event was injected. The three lines in the chart indicate which rule in the loaded rules file was trigger (first, last or random). The right chart in Figure 6.5 shows the heap- and non-heap-memory consumption of the JVM after the rules engine was initialized with different amounts of rules. Both charts show that the rules evaluation time respectively the JVM's memory consumption rise almost linear with the number of rules loaded.

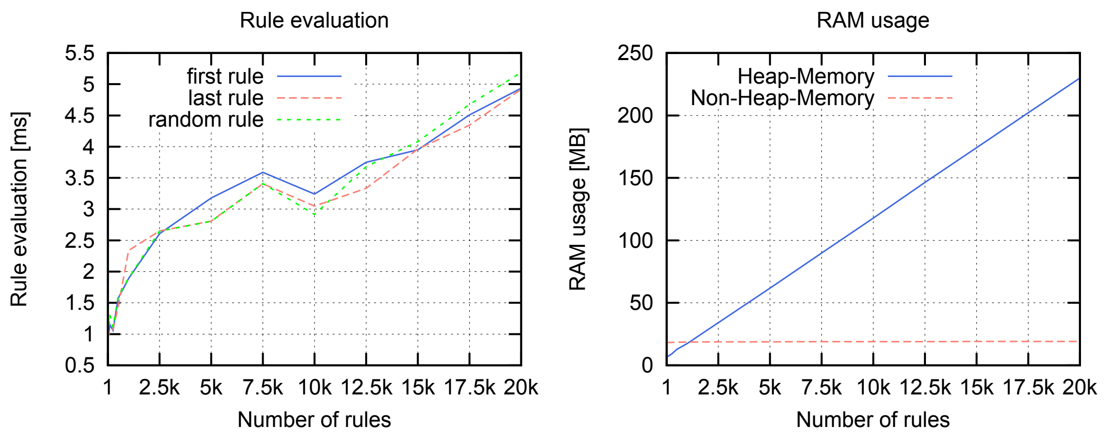

Figure 6.5: Trigger rules computation time and RAM usage

6.2.4 Evaluating the MigrationDataCollector

The last part of the migration mechanism to be evaluated is the `Migration-DataCollector`. Its task is to receive and process all events transmitted through jCloudScale's monitoring topic and generate migration planning data from the extracted information. The following will evaluate how the collector scales to different loads of monitoring events on the monitoring topic, to determine its impact on the jCloudScale-Client's performance and if its current architecture limits the number of concurrently active CHs and COs. That is because the number of monitoring events to process is directly related to the number of active CHs and COs.

The evaluation was conducted using DSG's OpenStack cloud with a jCloudScale-Server instance of flavor *m1.small*. The `COEvalEventSender` CO from Listing 6.2 was deployed to that instance and its `sendPlEvents` method was used to generate a steady stream of monitoring events. Before any performance indicators were measured the stream of monitoring events was kept constant for 1 minute, to allow the system to level off.

Figure 6.6 shows the results of the data collectors evaluation. The left chart shows the jCloudScale-Client's system load average during processing of different event loads. The right chart shows the percentage of CPU time consumed by the whole JVM process during processing of different event loads. It is important to note, that the consumed CPU time, displayed in the right chart, reflects the utilization of a single CPU core, but that the machine running the jCloudScale-Client provided

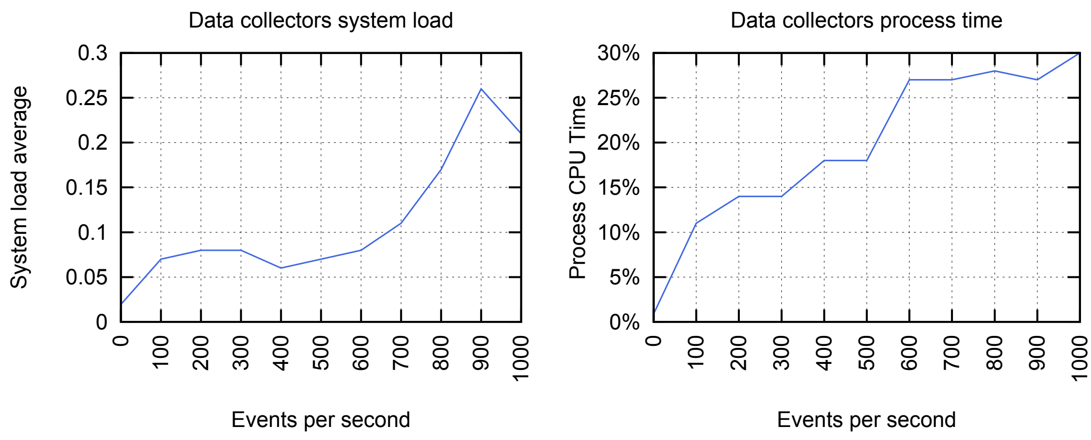

Figure 6.6: Data collectors system load and process CPU time

four CPU cores. This means that the total amount of consumable CPU time on this machine would be 400% and could be used by the `MigrationDataCollector`, if necessary, due to its multi-threaded architecture.

With that in mind it can be said, that the `MigrationDataCollector` scales sufficient to not harm jCloudScale's ability to operate large amounts of CHs and COs.

6.3 Evaluation summary

After evaluating the three distinguish parts of jCloudScale's migration mechanism the following can be said:

- *Performance:* The migration mechanism's overall performance impact on jCloudScale is not considerable. First, as shown in Section 6.2.1, the use of Java Instrumentation has no effect on a JVM's, and thus jCloudScale's, execution performance. Second, as shown in Section 6.2.2, finding an optimal migration solution using automated planning is fast (i.e. less than 0.5 seconds), even when problem spaces are large (i.e. about 1 million possible solutions). This is especially true when comparing time spent on automated planning with time spent for actually migrating a CO from one host to another. Third, as shown in Section 6.2.3, evaluating migration triggering rules is also very fast (i.e. less than 10ms), especially when assuming the use of a more realistic ruleset of less than 100 rules. Finally, as shown in Section

6.2.4, the data collector scales reasonable well to high event loads, although it bears the potential to overload the jCloudScale-Client.

- *Memory consumption:* As shown in Section 6.2.3 the migration mechanism's rules engine can occupy a significant amount of heap memory, when initialized with extensive rulesets. This can be a problem on systems with limited memory resources. However, when assuming that a realistic ruleset consists of not more than 100 rules memory consumption should be negligible.

The migration mechanism's evaluation showed also some potential problems. As already mentioned the current architecture of the `MigrationDataCollector` bears the potential to overload the jCloudScale-Client, especially if the client's machine does not support parallel execution of multiple application threads. Further the evaluation showed that when migrating COs with large memory footprints the client can become a bottleneck, slowing down the whole migration process significantly, if its network bandwidth to the MQ server is limited. Another source of a potential problem could be the MQ itself. As mentioned in Section 6.2.2 produces transmission errors, when trying to migrate COs larger than 100MB in serialized size. It is open for investigation if this issue is a configuration problem or a general limitation of the MQ software.

CHAPTER 7

Conclusion and Future Work

The introduction of this thesis stated the goals a migration mechanism for jCloudScale should achieve. Now it is time to conclude if these goals were met:

- The main goal of the migration mechanism was to be "[...] fully transparent for objects interacting with *Cloud Objects (COs)* going to be migrated and thus must preserve a COs state during its migration [...]".

  This goal was achieved. As described in Section 5.2 the migration mechanism uses *Two-phase locking (2PL)* technique to prevent other objects from accessing COs while they are migrated and references to them need to be updated. Further the values of a CO's member variables, and thus its internal state, are preserved during a migration.

- Another goal requested that the migration mechanism should "[...] support some kind of migration policies [...] to determine when to and where to migrate a CO [...]".

  This goal was also achieved. Migration "policies" are separated into migration triggering rules, which are evaluated by the rules engine and used to determine when to migrate a CO and into calculator classes for the planning engine to determine which migration solution is the optimal solution.

- Last, it was also a goal of the migration mechanism "[...] to keep the performance penalty on jCloudScale [...] as low as possible [...]".

  It can be said that this goal was also achieved, as shown and discussed in Chapter 6.

7.1 Future Work

During the migration mechanism's evaluation some issues and limitations, that can be addressed in future development, came to attention: When creating a new CO jCloudScale uses so called "scaling policies" to decide whether to deploy the object to an already existing *Cloud Host (CH)* or to better start a new one. Conceptually this is a very similar problem to finding a migration solution, when migrating a CO. Therefore it might be sensible to merge scaling policies and the finding of migration solutions into one unified policy mechanism.

Another issue revealed in the evaluation was the *Message Queue (MQ)*'s inability to transmit COs which serialized size exceeded 100MB. This limitation could be resolved by compressing and splitting serialized COs into several pieces, each one smaller than 100MB, before handing them over to the MQ for transmission. The receiving CH would than need to put all pieces back together again before deserializing the CO. Another possible solution would be to directly transmit serialized COs between CHs. This would not only circumvent the MQ, but also the jCloudScale client and might significantly speed up CO migrations, if the network connection to the client is considerable slower than between CHs.

Finally improving the type of code mobility used by the migration mechanism would be deservable. As described in Chapter 5 a CO must satisfy several requirements to be migratable. A type of code mobility that reduces these requirements and thus makes CO migration even more transparent for application developers would be a huge improvement. Unfortunately this kind of improvement is the most difficult one to achieve.

APPENDIX A

List of Abbreviations

2PL	Two-phase locking
ACL	Access Control List
ARPANET	Advanced Research Projects Agency Network
AMI	Amazon Maschine Image
AOP	Aspect-oriented Programming
AOT	Ahead-of-Time
API	Application Programming Interface
ASL	Apache Software License
AWS	Amazon Web Services
CAFE	Composite Application Framework
CE	Computational Environment
CEO	Chief Executive Officer
CEP	Complex Event Processing
CERN	Conseil Européen pour la Recherche Nucléaire
CH	Cloud Host

CIO	Chief Information Officer
CO	Cloud Object
CPU	Central Processing Unit
CRM	Customer Relationship Management
CSA	Cloud Security Alliance
DSG	Distributed Systems Group
EBS	Amazon Elastic Block Store
EC2	Amazon Elastic Compute Cloud
ECU	EC2 Compute Unit
EJB	Enterprise Java Bean
FIFO	First-In-First-Out
GAE	Google App Engine
HaaS	Human-as-a-Service
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
IP	Internet Protocol
Java EE	Java Enterprise Edition
JIT	Just-in-Time
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KRR	Knowledge Representation and Reasoning
MCM	Mobile Cloud Middleware
MOM	Message-oriented Middleware
MTBF	Mean Time Between Failure
MQ	Message Queue

NIST	National Institute of Standards and Technology
OCCI	Open Cloud Computing Interface
OCCI-WG	Open Cloud Computing Interface Working Group
OOP	Object-oriented Programming
OS	Operating System
OVF	Open Virtualization Format
PaaS	Platform-as-a-Service
RAID	Redundant Array of Independent Disks
RDS	Amazon Relational Database Service
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RRS	Reduced Redundancy Store
S3	Amazon Simple Storage Service
SaaS	Software-as-a-Service
SLA	Service Level Agreement
SME	Small- and medium-sized Enterprise
SOA	Service-oriented Architecture
TCO	Total Cost of Ownership
UI	User Interface
UML	Unified Modelling Language
VDI	Virtual Disk Image
VEE	Virtual Execution Environment
VM	Virtual Machine
VMI	Virtual Machine Interface

VMM	Virtual Machine Monitor
VPN	Virtual Private Network
WAR	Web Application Archive
WWW	World Wide Web
WYSIWYG	What You See Is What You Get

APPENDIX B

Evaluation: Statistical Analysis

In Chapter 6 it was stated, that running jCloudScale's-Server instances with Java Instrumentation enabled does not influence their execution performance significantly. In the following these claims will be verified using statistical hypothesis tests.

B.1 Computing a primitive byte array

The left chart in Figure 6.1 shows a comparison of averaged execution times, when filling a byte array, sized from 1MB to 1GB, with random data. To prove that Java Instrumentation has no influence on a jCloudScale-Server instance's execution performance the averaged execution times are analysed for significant differences, using a paired Student's t-test.

The averaged execution time for the computation of each measured array size, there differences and mean values are shown in Table B.1. The null hypothesis is specified as $\mu_0 = 0$, meaning that it is assumed that Java Instrumentation does not pose a significant performance impact. A confidence level of $\alpha = 0.1$ was chosen for the test.

$$Spec: \; H_0 : \mu_0 = 0; \; \alpha = 0.1; \; \bar{x} = -20303101 \tag{B.1}$$

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} = \sqrt{\frac{1}{30-1}\sum_{i=1}^{30}(x_i - \bar{x}^2)} = 132172886 \tag{B.2}$$

No.	Array size	with Instr.	without Instr.	Differences
#	[byte]	[ns]	[ns]	[ns]
1	0	45813442	40407367	5406075
2	1048576	62822942	32859173	29963769
3	2097152	27562011	37123665	-9561654
4	5242880	45943113	95275891	-49332778
5	10485760	174559127	97334800	77224327
6	15728640	191951369	259239627	-67288258
7	20971520	307741526	411927089	-104185563
8	26214400	274729331	353294342	-78565011
9	31457280	213201026	368005797	-154804771
10	36700160	284894164	447126903	-162232739
11	41943040	443480903	493765951	-50285048
12	47185920	429235423	421034746	8200677
13	52428800	528698355	448056737	80641618
14	62914560	608169041	514668620	93500421
15	73400320	789860659	881938503	-92077844
16	83886080	642773287	765649969	-122876682
17	94371840	725727585	926683646	-200956061
18	104857600	896169477	945777061	-49607584
19	134217728	1019963266	1062241873	-42278607
20	157286400	1183133977	1198702740	-15568763
21	201326592	1420277545	1452362984	-32085439
22	268435456	1782454634	1610295755	172158879
23	335544320	2058744829	1926888853	131855976
24	402653184	2573864077	2188323032	385541045
25	469762048	2666074812	2753999184	-87924372
26	536870912	2743401603	2595545671	147855932
27	671088640	2838408420	3103041228	-264632808
28	805306368	3507749839	3600584776	-92834937
29	939524096	4072294729	3955414072	116880657
30	1073741824	4482954500	4664178015	-181223515
Mean values		1234755167	1255058269	-20303101

Table B.1: Averaged execution times from left chart of Figure 6.1

$$Z = \frac{(\bar{x} - \mu_0)}{(s/\sqrt{n})} = \frac{(-20303101)}{(132172886/\sqrt{30})} = -0.841357614375 \tag{B.3}$$

$$|Z| > t_{n-1}; 1 - \frac{\alpha}{2} \tag{B.4}$$

$$0.841357614375 > 1.699 \Rightarrow \mu_0 \; holds. \tag{B.5}$$

As shown in equation (B.5) the test statistic is below the threshold, hence the null hypothesis holds. Java Instrumentation has no significant influence on jCloud-Scale's performance when working with primitive data types.

B.2 Computing a linked list of byte objects

The left chart in Figure 6.2 shows a comparison of averaged execution times, when creating a linked list of random byte objects, sized from 1MB to 100MB. To prove that Java Instrumentation has no influence on a jCloudScale-Server instance's execution performance the averaged execution times are analysed for significant differences, using a paired Student's t-test.

The averaged execution time for the computation of each measured size of the linked list, there differences and mean values are shown in Table B.2. The null hypothesis is specified as $\mu_0 = 0$, meaning that it is assumed that Java Instrumentation does not pose a significant performance impact. A confidence level of $\alpha = 0.1$ was chosen for the test.

$$Spec: \; H_0 : \mu_0 = 0; \; \alpha = 0,1; \; \bar{x} = -494389203 \tag{B.6}$$

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} = \sqrt{\frac{1}{17-1}\sum_{i=1}^{17}(x_i - \bar{x}^2)} = 1508107247 \tag{B.7}$$

$$Z = \frac{(\bar{x} - \mu_0)}{(s/\sqrt{n})} = \frac{(-494389203)}{(1508107247/\sqrt{17})} = 1,35164054691 \tag{B.8}$$

$$|Z| > t_{n-1}; 1 - \frac{\alpha}{2} \tag{B.9}$$

$$1,35164054691 > 1,746 \Rightarrow \mu_0 \; holds. \tag{B.10}$$

No.	**List size**	**with Instr.**	**without Instr.**	**Differences**
#	[byte]	[ns]	[ns]	[ns]
1	0	23211714	67838487	-44626773
2	1048576	905899855	1046989437	-141089582
3	2097152	1639209517	2534414951	-895205434
4	5242880	2825027272	3921885073	-1096857801
5	10485760	5096090714	6993223575	-1897132861
6	15728640	7102077367	7552149123	-450071756
7	20971520	9070855360	7680057593	1390797767
8	26214400	11668608578	11033597973	635010605
9	31457280	11449631517	12648977515	-1199345998
10	36700160	13945216814	14887290025	-942073211
11	41943040	16454009659	16528285990	-74276331
12	47185920	18902270143	20154254018	-1251983875
13	52428800	23896935032	21612260370	2284674662
14	62914560	26364793844	27161549198	-796755354
15	73400320	34523035979	32689971027	1833064952
16	83886080	34462053073	36182858494	-1720805421
17	94371840	42276377015	46314317058	-4037940043
Mean values		15329723732	15824112935	-494389203

Table B.2: Averaged execution times from left chart of Figure 6.2

As shown in equation (B.10) the test statistic is below the threshold, hence the null hypothesis holds. Java Instrumentation has no significant influence on jCloud-Scale's performance when working with object data.

References

[1] Jeff Bezos' Risky Bet. *Businessweek*, 46, 2006. http://www.businessweek.com/magazine/content/06_46/b4009001.htm, Visited: 2011-12-27.

[2] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 2–13, 2006. http://doi.acm.org/10.1145/1168857.1168860, Visited: 2012-01-29.

[3] Janna Anderson and Lee Rainie. The future of cloud computing. *Pew Research Center*, June 11, 2010. http://pewinternet.org/Reports/2010/The-future-of-cloud-computing/Overview.aspx, Visited: 2012-01-14.

[4] Nick Antonopoulos and Lee Gillam. *Cloud Computing: Principles, Systems and Applications.* Springer London Limited, 1. edition, 2010.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. *Electrical Engineering and Computer Sciences, University of California at Berkeley*, 2009. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf, Visited: 2011-12-27.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, 2003. http://doi.acm.org/10.1145/1165389.945462, Visited: 2012-01-29.

[8] Colin Barker. HP dismisses cloud 'hype'. *ZDnet News*, December 11, 2008. http://www.zdnet.com/news/hp-dismisses-cloud-hype/255222, Visited: 2012-01-13.

[9] Jeff Barr. AWS Links. *Amazon Web Services Blog*, February 2009. http://aws.typepad.com/aws/2009/02/aws-links-wednesday-february-25-2009.html, Visited: 2012-03-11.

[10] Anitesh Barua, Jon Pinnell, Jay Shutter, and Andrew B. Whinston. Measuring the Internet Economy: An Exploratory Study. *Center for Research in Electronic Commerce, University of Texas at Austin*, 1999.

[11] Christian Baun, Marcel Kunze, Jens Nimis, and Stefan Tai. *Cloud Computing: Web-Based Dynamic IT Services*. Springer Berlin Heidelberg, 1. edition, 2011. DOI: 10.1007/978-3-642-20917-8_6.

[12] Greg Bayer. Scaling with the Kindle Fire. *Google App Engine Blog*, November 2011. http://googleappengine.blogspot.com/2011/11/scaling-with-kindle-fire.html, Visited: 2012-02-11.

[13] Tim Berners-Lee. Information Management: A Proposal, 1989. *http://www.w3.org/History/1989/proposal.html*. Visited: 2011-12-17.

[14] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[15] Arnd Böken. Patriot Act und Cloud Computing: Zugriff auf Zuruf? [in German]. *IX, Heise Zeitschriften Verlag GmbH & Co. KG*, January 2012. http://heise.de/-1394430, Visited: 2012-02-19.

[16] Rajkumar Buyya and Srikumar Venugopal. The Gridbus toolkit for service oriented grid and utility computing: an overview and status report. *1st IEEE International Workshop on Grid Economics and Business Models (GECON)*, pages 19–66, 2004. DOI: 10.1109/GECON.2004.1317583.

[17] Mariana Carroll, Alta van der Merwe, and Paula Kotzé. Secure Cloud Computing: Benefits, Risks and Controls. *Information Security South Africa (ISSA)*, pages 1–9, August 2011. DOI: 10.1109/ISSA.2011.6027519.

[18] Danielle Catteddu and Giles Hogben. Cloud Computing: Benefits, Risks and Recommendations for Information Security. *European Network and Information Security Agency (ENISA)*, November 2009. http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-risk-assessment, Visited: 2012-02-11.

[19] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association. http://dl.acm.org/citation.cfm?id=1251203.1251223, Visited: 2012-01-29.

[20] Inc. CloudBees. CloudBees. *http://www.cloudbees.com/*. Visited: 2012-07-20.

[21] IDC International Data Corporation. IDC Cloud Research. http://www.idc.com/prodserv/idc_cloud.jsp, Visited: 2012-01-14.

[22] Microsoft Corporation. .NET Framework Developer Center. *Microsoft Developer Network (MSDN)*. http://msdn.microsoft.com/en-us/netframework/default, Visited: 2012-05-08.

[23] Microsoft Corporation. Windows Azure. *http://www.windowsazure.com*. Visited: 2011-12-29.

[24] Oracle Corporation. Oracle Java Web Service. *https://cloud.oracle.com/*. Visited: 2012-07-20.

[25] Xamarin Corporation. The Mono Project. http://www.mono-project.com, Visited: 2012-05-08.

[26] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling java just in time. *Micro, IEEE*, 17(3):36–43, May 1997.

[27] Dan Farber. Oracle's Ellison nails cloud computing. *CNET News*, September 26, 2008. http://news.cnet.com/8301-13953_3-10052188-80.html?part=rss&subj=news&tag=2547-1_3-0-5, Visited: 2012-01-14.

[28] Jackie Fenn. When to Leap on the Hype Cycle. *Gartner Group*, January 11, 1995. http://www.gartner.com/DisplayDocument?doc_cd=22229, Visited: 2012-01-14.

[29] Dietmar Fey. *Grid-Computing: Eine Basistechnologie für Computational Science [in German]*. Springer Berlin Heidelberg, 1. edition, 2010.

[30] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, CA, 2000.

[31] Roy T. Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Peter Leach, and Tim Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. *RFC 2616*, 1999. http://tools.ietf.org/html/rfc2616, Visited: 2011-12-16.

[32] Huber Flores, Satish Narayana Srirama, and Carlos Paniagua. A Generic Middleware Framework for Handling Process Intensive Hybrid Cloud Services from Mobiles. *9th International Conference on Advances in Mobile Computing & Multimedia (MoMM)*, December 2011.

[33] Apache Software Foundation. Apache License, Version 2.0. *http://www.apache.org/licenses/LICENSE-2.0*. Visited: 2013-11-03.

[34] The OpenStack Foundation. OpenStack Cloud Software. *http://www.openstack.org/*. Visited: 2014-01-10.

[35] Marc S. Friedman. The USA Patriot Act - Implications for Cloud Computing. *SNR Denton*, November 2011. http://www.snrdenton.com/PDF/USA_Patriot_Act_Cloud_Computing.pdf, Visited: 2012-02-19.

[36] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[37] Erich Gamma, Richard Helm Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1. edition, 1994. ISBN: 978-0201633610.

[38] Frank Gens. IDC's Public IT Cloud Services Forecast: New Numbers, Same Disruptive Story. July 2010. http://blogs.idc.com/ie/?p=922, Visited: 2012-02-10.

[39] Antonio Goncalves. *Beginning Java EE 6 Platform with GlassFish 3*. Apress, 1. edition, 2009. ISBN: 978-1-4302-1954-5.

[40] Google Inc. Google Maps API. http://code.google.com/intl/en/apis/maps/, Visited: 2012-01-30.

[41] Gartner Group. Gartner's 2013 Hype Cycle Special Report Evaluates the Maturity of 1,900 Technologies. August 10, 2013. http://www.gartner.com/newsroom/id/2575515, Visited: 2014-03-29.

[42] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP Version

1.2 Part 1: Messaging Framework. *World Wide Web Consortium (W3C)*, 2007. http://www.w3.org/TR/soap12-part1/, Visited: 2012-04-10.

[43] Sheikh Mahbub Habib, Sebastian Ries, and Max Mühlhäuser. Cloud Computing Landscape and Research Challenges regarding Trust and Reputation. *International Conference on Ubiquitous Intelligence & Computing and 7th International Conference on Autonomic & Trusted Computing (UIC/ATC)*, pages 410–415, October 2010. DOI: 10.1109/UIC-ATC.2010.48.

[44] Rolf Harms and Michael Yamartino. The Economics of the Cloud. *Microsoft Corporation*, 2010. http://www.microsoft.com/presspass/presskits/cloud/docs/The-Economics-of-the-Cloud.pdf, Visited: 2012-01-30.

[45] David Hilley. Cloud Computing: A Taxonomy of Platform and Infrastructure-level Offerings. *CERCS Technical Report, Georgia Institute of Technology*, 2009.

[46] Jason Hsiao. Amazon.com CEO Jeff Bezos on Animoto. *The Animoto Blog*, April 2008. http://animoto.com/blog/company/amazon-com-ceo-jeff-bezos-on-animoto/, Visited: 2012-02-11.

[47] Amazon.com Inc. Amazon Elastic Compute Cloud (EC2). *Amazon Web Services*. http://aws.amazon.com/ec2/, Visited: 2012-03-15.

[48] Amazon.com Inc. Amazon Simple Storage Service (S3). *Amazon Web Services*. http://aws.amazon.com/s3/, Visited: 2012-03-15.

[49] Amazon.com Inc. Amazon Web Services. *http://aws.amazon.com/*. Visited: 2011-12-29.

[50] Amazon.com Inc. AWS Global Infrastructure. *http://aws.amazon.com/about-aws/globalinfrastructure/*. Visited: 2011-12-27.

[51] Amazon.com Inc. Overview of Amazon Web Services. December 2010. http://media.amazonwebservices.com/AWS_Overview.pdf, Visited: 2012-03-15.

[52] Amazon.com Inc. Amazon Web Services: Overview of Security Processes. May 2011. http://media.amazonwebservices.com/pdf/AWS_Security_Whitepaper.pdf, Visited: 2012-03-15.

[53] Facebook Inc. Open compute: data center technology. *http://opencompute.org/project_category/data-center-technology/*. Visited: 2011-12-27.

[54] Google Inc. Google App Engine. *http://code.google.com/intl/en/appengine/*. Visited: 2011-12-29.

[55] Google Inc. Google data centers. *http://www.google.com/about/datacenters/*. Visited: 2011-12-27.

[56] Internet Systems Consortium (ISC). Internet host count history. *https://www.isc.org/solutions/survey/history*. Visited: 2011-12-17.

[57] Internet World Stats (IWS). Internet Growth Statistics. *http://www.internetworldstats.com/emarketing.htm*. Visited: 2011-12-17.

[58] Wayne A. Jansen. Cloud Hooks: Security and Privacy Issues in Cloud Computing. *44th Hawaii International Conference on System Sciences (HICSS)*, pages 1–10, January 2011. DOI: 10.1109/HICSS.2011.103.

[59] Inc. Jelastic. Jelastic. *http://jelastic.com/*. Visited: 2012-07-20.

[60] Bobbie Johnson. Cloud Computing is a trap, warns GNU founder Richard Stallman. *The Guardian*, September 29, 2008. http://www.guardian.co.uk/technology/2008/sep/29/cloud.computing.richard.stallman, Visited: 2012-01-13.

[61] Balachandra Reddy Kandukuri, Ramakrishna Paturi, and Dr. Atanu Rakshit. Cloud Security Issues. *IEEE International Conference on Services Computing (SCC'09)*, pages 517–520, September 2009. DOI: 10.1109/SCC.2009.84.

[62] Lori M. Kaufman. Data Security in the World of Cloud Computing. *IEEE Security & Privacy*, pages 61–64, August 2009. DOI: 10.1109/MSP.2009.87.

[63] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[64] Leonard Kleinrock. History of the Internet and its flexible future. *Wireless Communications, IEEE*, 15(1):8–18, 2008.

[65] Leonard Kleinrock. An early history of the internet [History of Communications]. *Communications Magazine, IEEE*, 48(8):26–36, 2010.

[66] Vivek Kundra. Federal Cloud Computing Strategy. *Chief Information Officers Council*, February 8, 2011. http://www.cio.gov/documents/Federal-Cloud-Computing-Strategy.pdf, Visited: 2012-01-20.

[67] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-oriented Programming.* Manning, 2. edition, 2003. ISBN: 978-1930110939.

[68] Philipp Leitner. CloudScale's Project Site. *http://code.google.com/p/cloudscale/*. Visited: 2013-11-03.

[69] Philipp Leitner, Benjamin Satzger, Christian Inzinger, Waldemar Hummer, and Schahram Dustdar. CloudScale - a Novel Middleware for Building Transparently Scaling Cloud Applications. *27th Symposium On Applied Computing (SAC)*, 2012.

[70] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the Cloud? An architectural map of the Cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing (CLOUD'09), Washington, DC, USA*, pages 23–31. IEEE Computer Society, 2009.

[71] Di Liu, Tao Gu, and Jiang-Ping Xue. Rule Engine based on improvement Rete algorithm. *International conference on Apperceiving Computing and Intelligence Analysis (ICACIA)*, pages 346 – 349, 2010. ISBN: 978-1-4244-8025-8.

[72] Zalgham Mahmood and Richard Hill. *Cloud Computing for Enterprise Architectures, Computer Communications and Networks*. Springer London Limited, 1. edition, 2011. DOI 10.1007/978-1-4471-2236-4_3.

[73] Q. Mahmoud. *Middleware for Communications*. Wiley, 2005.

[74] Soumaya Marzouk and Mohamed Jmaiel. A Policy-based Approach for Strong Mobility of Composed Web Services. *Service Oriented Computing and Applications*, Print ISSN: 1863-2386, Online ISSN: 1863-2394, 2013. Springer-Verlag, DOI 10.1007/s11761-013-0131-9.

[75] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. *NIST Special Publication*, (800-145), 2011. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf, Visited: 2011-12-27.

[76] Microsoft Corporation. Office 365. http://www.microsoft.com/en-us/office365/online-software.aspx, Visited: 2012-01-30.

[77] Ralph Mietzner. Cafe Project Homepage. http://www.iaas.uni-stuttgart.de/forschung/projects/cafe/, Visited: 2012-04-30.

[78] Ralph Mietzner, Tobias Unger, and Frank Leymann. Cafe: A Generic Configurable Customizable Composite Cloud Application Framework. *Lecture Notes in Computer Science*, 5870/2009:357–364, 2009.

[79] Russell Miles. *AspectJ Cookbook.* O'Reilly Media, 1. edition, 2004. ISBN: 0-596-00654-3.

[80] A. Nilsson and S.G. Robertz. On real-time performance of ahead-of-time compiled java. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 372–381, May 2005.

[81] Oracle Corporation, VirtualBox. http://www.virtualbox.org/. Visited: 2012-01-29.

[82] Andy Oram and Greg Wilson. *Beautiful Code: Leading Programmers Explain How They Think.* O'Reilly Media, 1. edition, 2007. Chapter 17, pages 279–291.

[83] Christy Pettey and Ben Tudor. Gartner Says Worldwide Cloud Services Market to Surpass $68 Billion in 2010. August 2010. http://www.gartner.com/it/page.jsp?id=1389313, Visited: 2012-02-10.

[84] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17:412–421, July 1974. http://doi.acm.org/10.1145/361011.361073, Visited: 2012-01-29.

[85] Rackspace US, Incorporation. http://www.rackspace.com. Visited: 2012-01-30.

[86] Mendel Rosenblum. The Reincarnation of Virtual Machines. *Queue*, 2:34–40, July 2004. http://doi.acm.org/10.1145/1016998.1017000, Visited: 2012-01-29.

[87] Uwe Rozanski. *Enterprise JavaBeans 3.1: Einstieg, Umstieg, Praxis und Referenz [in German].* mitp Verlag, 1. edition, 2011. ISBN: 978-3-8266-9066-2.

[88] Salesforce.com. Salesforce CRM. http://www.salesforce.com/crm/, Visited: 2012-01-30.

[89] Americo Sampaio and Nabor Mendonca. Uni4Cloud: An Approach based on Open Standards for Deployment and Management of Multi-cloud Applications. *International Workshop on Software Engineering for Cloud Computing (SECLOUD) 2011*, May 2011.

[90] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with Soap.* O'Reilly Media, 1. edition, 2002. ISBN: 978-0596000950.

[91] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2. edition, 2006.

[92] The Eclipse Foundation. Aspectj project. http://eclipse.org/aspectj/, Visited: 2011-12-16.

[93] The Eclipse Foundation. The AspectJ 5 Development Kit Developer's Notebook. http://eclipse.org/aspectj/doc/released/adk15notebook/, Visited: 2012-04-10.

[94] Huaglory Tianfield. Cloud Computing Architectures. *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1394–1399, October 2011. http://fatech.org.uk/dpsi/publication/2011smc11.pdf, Visited: 2012-01-29.

[95] Dave Valliere and Rein Peterson. Inflating the bubble: examining dot-com investor behaviour. *Venture Capital*, 6(1):1–22, 2004.

[96] Jinesh Varia. Cloud Architectures, 2008. *http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf*. Visited: 2011-12-27.

[97] Jinesh Varia. Architecting for the Cloud: Best Practices. *Amazon Web Services*, January 2010. http://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf, Visited: 2012-03-18.

[98] Christian Vecchiola, Xingchen Chu, and Rajkumar Buyya. Aneka: A Software Platform for .NET-based Cloud Computing. *CoRR - Computing Research Repository*, abs/0907.4622, 2009. http://arxiv.org/abs/0907.4622, Visited: 2011-12-29.

[99] Markus Voelter, Michael Kirchner, and Uwe Zdun. *Remoting Patterns - Foundations of Enterprise, Internet and Realtime Distributed Object Middleware.* John Wiley & Sons, 1. edition, 2004. ISBN: 978-0470856628.

[100] Thorsten von Eicken. Animoto's Facebook scale-up. *RightScale Blog*, April 2008. http://blog.rightscale.com/2008/04/23/animoto-facebook-scale-up/, Visited: 2012-02-11.

[101] World Wide Web Consortium (W3C). A Little History of the World Wide Web. *http://www.w3.org/History.html*. Visited: 2011-12-17.

[102] Jon Watson. VirtualBox: Bits and Bytes masquerading as Machines. *Linux Journal*, 2008. http://dl.acm.org/citation.cfm?id=1344209.1344210, Visited: 2012-01-29.

[103] Peter R. Wheale and Laura H. Amin. Bursting the dot.com Bubble: A Case Study in Investor Behaviour. *Technology Analysis & Strategic Management*, 15(1):117–136, 2003.

[104] XenSource Incorporation, XEN. http://www.xen.org. Visited: 2012-01-29.