Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer zai-bsc-2011-17

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Using Cloud Computing Resources in Grid Systems: An Integration of Amazon Web Services into UNICORE 6

Maik Doleys

am Institut für

Informatik
Gruppe Softwaretechnik für Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

October 14, 2011

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel.	+49 (5 51) 39-17 42010
Fax	+49 (5 51) 39-1 44 15
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den October 14, 2011

Bachelor's thesis

Using Cloud Computing Resources in Grid Systems: An Integration of Amazon Web Services into UNICORE 6

Maik Doleys

October 14, 2011

Supervised by Prof. Dr. Jens Grabowski
Software Engineering for Distributed Systems Group
Georg-August-Universität Göttingen

Abstract

The cloud computing paradigm overcomes its infancy. While the demand of resources for scalable computational systems increases, the integration of cloud computing resources into grid systems can become a potential extension for grid systems.
This thesis presents possible designs that integrate cloud computing resources into grid systems. General requirements and aspects for such integration designs are provided and discussed. Furthermore, we implement a design that automatically integrates resources from the Amazon Web Services (AWS) cloud computing environment into the Uniform Interface to Computing Reources (UNICORE) 6 grid environment.
This implementation has been validated with test specifications verifies, which shows the possibility of extending grid resources with resources of a cloud environment.

Acknowledgement

The work carried out in this thesis has been supported by a research grant of Amazon Web Services to use AWS resources.
I would like to thank Thomas Rings for his help, and the honest and productive discussions.
This thesis is dedicated to my parents. You will always be with me.
Thanks mom, Thanks dad!

Contents

1 Introduction

The concept of grid systems evolved in the 1990s, since this time definitions and use cases for grid systems were developed and defined. But never the less, use cases for grid computing still evolves. The cloud computing paradigm is relatively new compared to grid systems. It was developed by global companies as Amazon, Google and Yahoo. These companies needed huge amounts of computing power to serve request peaks at their services. At the beginning of the 21th century the companies had the idea to sell their computing resources that they do not need in idle times to the public. Offering these computer resources to the public was the beginning of the cloud computing paradigm; from then on the term cloud computing was widely discussed. In the year 2009, the National Institute of Standards and Technology (NIST) published a clear definition of the cloud-computing paradigm that gains a wide range of agreement [78]. But during the writing of this thesis, we notice the increase in the number and type of cloud services.
However, with the integration of cloud computing resources into grid systems, a new use case for the cloud computer paradigm can be developed. We think in the combination of grid systems and cloud computing environments lies huge possibilities for both, e.g. grid systems can get more flexible in resource allocation and the cloud computing paradigm could adopt techniques and standards from the grid systems. We want to provide ideas and discussion about abstract scenarios on how to integrate cloud resources into grid systems. Furthermore, we implement our idea to proof that it is possible to integrate cloud computing resources into grid systems. In a nutshell, we create a small piece towards the integration of the cloud computing paradigma into grid systems.

Motivation

Grid systems provide services to use distributed computing resources and cloud computing provides computing resources and services. Because of the coherence between grid and cloud computing the idea to integrate cloud computing, resources into grid systems seems nearby. To increase the amount of available computing resources for grid environments, we identified two possibilities. First, buy and build physical computing hardware resources and add them to the grid. Second add another already existing distributed computing resource to the grid. In the first solution, the grid user needs time to create computing resources, furthermore long-term investments are necessary and altogether it seems to

be an inflexible solution to extend the grid resources dynamically by the users demand. For the second solution, the grid user has to find stakeholders that offer the user the usage of hardware resources, depending on the grid users demand. These already existing distributed computing resources from the second solution can be cloud computing resources. To include cloud computing resources into grid systems dynamically seem to be a flexible solution to extend the resources of existing grid environments.
However, to use cloud computing resources in grid systems promises some advantages. The allocation of cloud computing resources seems to be very flexible. It is possible to rent cloud computing resources on fixed time rates and the allocation of cloud resources can absorb peaks of resource requests in grid systems.
Our grid system resources in the Institute of Computer Science at the Georg-August University are limited to the existing hardware resource. To increase the possible amount of grid resources, we are not able to increase the physical amount of our computing hardware. For this reason, it could be a solution for us to use cloud computing resources to extend our grid resources on demand.

Goal Description

This thesis analyses possible abstract scenarios on how to integrate cloud computing resources in grid environments. We present general requirements and aspects how to classify such integration scenarios. Furthermore, we provide our implementation that integrates automatically cloud computing resources in grid environments. In this implementation we extend the UNICORE High Level (HiLA) shell with an extension that automatically make cloud computing resources available from the AWS cloud. To validate our implemented extension in the UNICORE HiLA shell, we provide test specification to test the grid infrastructure and possible usability of the UNICORE grid in the AWS cloud.

Thesis Structure

The thesis is structured in five chapters. In Chapter 2, we describe the technical base for our thesis that are computer clusters, grid computing and the cloud computing paradigm. In Chapter 3, we provide abstract scenarios on how to extend grids with cloud computing resources. Furthermore, we list and discuss general requirements and aspects on how to analyse scenarios that extend grid environments. In Chapter 4, we introduce and explain our implementation that extends UNICORE with resources from the AWS cloud. In addition, we validate our implementation with example case studies about using AWS resources in the UNICORE grid environment. In the Chapter 5, we summarize and discuss our thesis and provide suggestions for ongoing works on the thesis content.

2 Foundation

In this chapter, we explain the technical concepts and services that build the basis of this thesis. In Section 2.1 we introduce the computer cluster paradigm. In Section 2.2, we provide a introduction to grid computing systems and the UNICORE grid environment and it's components. Furthermore, in Section 2.3 we provide a definition for the cloud computing paradigm and explain the technical details. In addition, we introduce the AWS cloud computing environment with it components.

2.1 Computer Clusters

A computer cluster connects computer resources to achieve a computing machine with more computing power than a single computer could ever reach. Computing power of individual computers is always limited, because of physical laws or financial costs. However, by connecting different computing units, these limits can be overcome and a powerful and inexpensive computing unit can be provided.
Computer cluster use several technologies, the communication between computers is realized by local network connections, for example Local Area Network or InfiniBand. Therefore the participating computers are arranged in a dedicated network. Computing tasks can be distributed automatically between these connected computing instances. Cluster systems are categorized into different categories; High-availability (HA) clusters, Load-balancing clusters, and compute clusters.
Dividing a computing job into individual disjunctive tasks is the basis of cluster systems. To dismantle the computing job, the cluster divide it into individual and disjunctive tasks. The participating computers of the cluster system calculate the tasks. Finally the solution of the computing job is composed out of the calculated tasks.

2.1.1 Local Resource Management

The Local Resource Management (LRM) concatenates the underlying cluster nodes to a single system. In Figure 2.1, the LRM resides in the Cluster Middleware architecture. The LRM creates a Single System Image (SSI) that is deployed on the the cluster resources [69]. This gives the user of a cluster system the possibility to handle the cluster as a single system. A cluster LRM system has four components with the following functionality [84]:

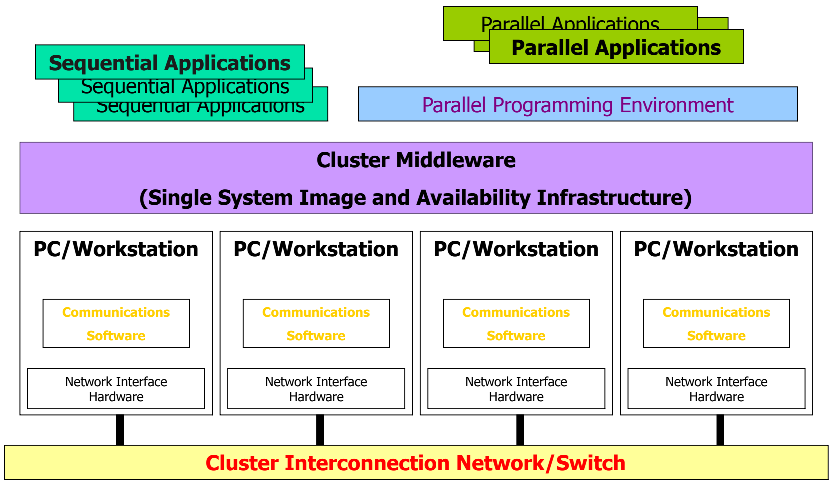

Figure 2.1: Cluster Architecture, [70]

Resource Management The Resource Management component provides status information of cluster resources, for example the processor load or the free capacity of disk storage.

Queuing The queuing components maintains jobs until free cluster resources for execution are available.

Scheduling The scheduling component allocates cluster resources to jobs that depends on scheduling policies.

Execution The execution component submits and executes computing taks on allocated cluster notes and fetches results.

For LRM systems standards do not exist. Therefore information or data exchange between different LRM is hard to achieve. Different cluster LRM systems are for example, Portable Batch System (PBS), LoadLeveler, and Load Sharing Facility (LSF).

2.1.2 Portable Batch System

PBS is a computing job scheduling software package belonging to the LRM systems. It was developed for the NASA in the 1990's. It fits the POSIX Batch Environment Standard. For this standard, it is able to accept batch jobs that calls a shell script and manage job execution from scheduling to fetching the result [45].
Currently, there are three different versions of PBS available:

OpenPBS The OpenPBS version is not maintained anymore, but it is the source for the following [44].

TORQUE The TORQUE version is an improved fork of the original OpenPBS system. The TORQUE software is maintained by Cluster Resources Inc. and it is used in subsection 4.5.3 later in this thesis [57].

PBS Professional The PBS Professional is a commercial version of PBS, developed by Altair Engineering [47].

All three versions have a similar underlying architecture, as shown in Figure 2.2 [76], that sketches a multi host environment. The Host A is the minimal working environment, which can be extended by several execution hosts, such as Hosts B and Host C. A communication procedue between the components for job execution demonstrates the numbers one to eight.

PBS contains four main components. They are categorized into user-level commands and system-daemons. Together, they implement the four components of a cluster LRM system. These components are described in the following [77] [57]:

Commands Negotiation possibilities with the PBS system are POSIX conform line commands or a Graphical User Interface (GUI). Three categories of commands are available: user commands, operator commands, and administrator commands. They achieve the monitoring and management of jobs or configure the system. These commands can be executed on a machine without presence of the PBS daemons.

Job Server The PBS_server daemon is the heart of the PBS system. All other daemons and commands interact over Internet Protocol (IP) protocols with this central instance. The server manages the queues for the scheduler and takes supervision over the jobs.

Job Executer MOM The PBS_mom daemons move jobs into execution and fetch the outcome. At least there must be a daemon on every worker node that executes jobs.

Job Scheduler The PBS_sched daemon manages resources and information for job execution between the moms and the PBS_server. The job scheduler daemon decides about job execution and resource allocation depending on the sites policies.

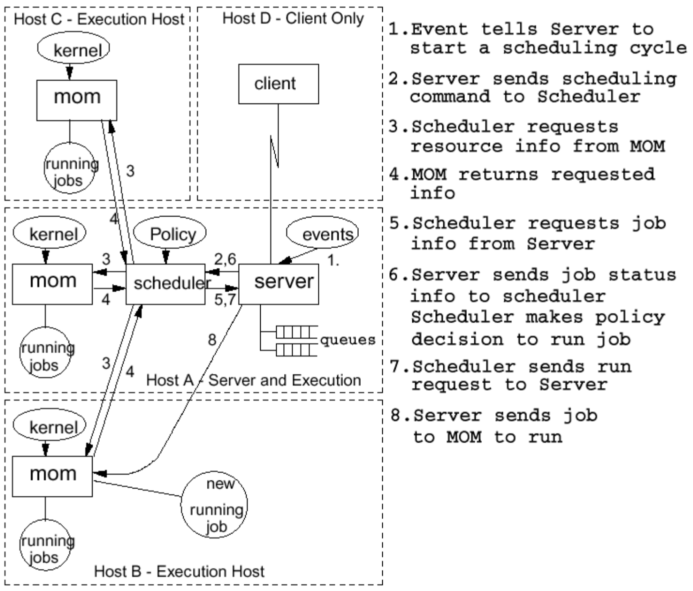
Host C - Execution Host
Host D - Client Only
kernel
mom
running jobs
client
kernel
Policy
events
mom
scheduler
server
queues
Host A - Server and Execution
kernel
mom
new running job
running jobs
Host B - Execution Host
1.Event tells Server to start a scheduling cycle
2.Server sends scheduling command to Scheduler
3.Scheduler requests resource info from MOM
4.MOM returns requested info
5.Scheduler requests job info from Server
6.Server sends job status info to scheduler Scheduler makes policy decision to run job
7.Scheduler sends run request to Server
8.Server sends job to MOM to run

Figure 2.2: PBS Architecture [76]

2.2 Grid Computing

Grid computing systems connect distributed heterogeneous computers and computing resources and provide applications to manage compute cluster and storage to achieve a coherent and powerful system for the user. Grid computing has a long time of groundwork behind it. During the development of the grid computing concept many different definitions were published. In the beginning the term grid computing was widely used and under this circumstance even normal personal computers could be called as grids. But does, for example, accessing computing storage over a local network form a storage grid? Could every personal computer be seen as a grid? To develop the grid computing further a precise definition was necessary.
A widely accepted definition is mentioned in “The Grid: Blueprint for a New Computing Infrastructure ” 1998 by I.Foster and C. Kesselman [75]. In the year 2000, I. Foster and C. Kesselman overworked their definition from 1998 to deal with some upcoming social and policy issues that where related to grid characteristics of coordinated and shared resources from different control domains [**?**]. The definition from the year 2000 evolves two years later as a three points checklist that a system must conform to become a grid [74]. According to I. Foster’s and C. Kesselman’s, the three points checklist a grid system must satisfy the three following points:

- A grid system “coordinates resources that are not subject to centralized control” and integrates resources and users from different control domains under the consideration of security, policy, payment, and membership.

- Furthermore a grid system uses “standard, open, general-purpose protocols and interfaces ”. As in difference to the application-specific systems, grid system protocols and standards have to be standardized and open. And the multi-purpose protocols and interfaces have to fulfil the grid’s requirements.
- Recently a grid system has “to deliver nontrivial qualities of service” and should make it possible to combine various resources in a way that this combination is significant greater than the sum of the parts.

Grid systems are layer-based architectures. On the top, the user- or access layer is the entry point for the user. The user-layer comprises the grid-applications and programming interfaces. In the middle-layer, the grid middleware connects the user-layer with resources from the bottom layer. Examples for typical grid middleware’s are UNICORE [64], Globus Toolkit [30], and g-lite [29]. The bottom-layer is responsible for accessing the hardware specific resources in the grid [72].

2.2.1 Open Grid Service Architecture

The Open Grid Service Architecture (OGSA) that is developed by the Open Grid Forum (OGF) makes different heterogeneous grid middleware systems interoperable with a standardized architecture. [43] The OGSA “does not define any standards or technical recommendations” [41], the standard is a very abstract definition of interfaces and requirements that provide mechanisms for heterogeneous systems in grid environments to communicate and interact. The OGSA architecture defines requirements as “Interoperability and Support for Dynamic and Heterogeneous Environments”, “Job Execution ”, “Availability ”, etc. [41], furthermore needed and optional components that has to implement these requirements are defined. For example such components are “, Infrastructure Services ”, and “ Execution Management ”. All components interact in an service orientated Service Orieted Architecture (SOA) environment, all components are separated from each other to achieve a high flexibility inside the OGSA architecture. The separation between the different components achieves to make changes in the OGSA architecture and instantiated implementations easier [72].

2.2.2 Web Service - Resource Framework

To realize the communication between OGSA components, a stateful communication is convenient, because it is possible that results from the OGSA components depend on each other. Based on stateless web services, the grid community developed the Web Service

- Resource Framework (WS-RF) standard. This service "defines a generic framework for modelling and accessing persistent resources using Web services "[66] that simplifies the development of stateful grid services based on the OGSA architecture. This framework made it possible to achieve a high flexibility handling concrete implementations of the OGSA components. WS-RF relies on the same technical standards as web services For communication between grid services, the Simple Object Access Protocol (SOAP) [54] protocol is used and grid services can be described by Web Service Description Language (WSDL) [72].

2.2.3 UNICORE 6

The UNICORE grid system is a grid middleware. The development started 1997 with the goal to "enable German supercomputer centres to provide their users with a seamless, secure, and intuitive access to their heterogeneous computing resources "[61]. The project Unicore plus that was ongoing until 2002 forms the basis for the UNICORE 6 implementation. Since 2004 UNICORE is published under the open source BSD license and available in a public accessible subversion [62] repository and therefore open for every developer [67].
In the year 2011, as the basic components of the project were established, new requirements became important. According to the UNICORE community, grid services become more and more present in "day-to-day operation of e-infrastructure and in application-driven use cases "[67] .The development focus of the UNICORE community moved from the UNICORE basis, such as concepts, architecture and interfaces, to higher level services such as the "integration of upcoming standards, deployment in e-infrastructures, setup of interoperability use cases and integration of applications ".
The architecture of the UNICORE 6 middleware is displayed in Figure 2.3. The three-layer architecture includes top to the bottom, the client-layer with scientific clients and applications, the service-layer with authentication wrapper and the grid components and at the system-layer with the target systems. Based on the layered structure, we present a description for every layer:

Client layer In the client-layer, grid clients and abstract Application Programming Interface (API) for accessing the grid reside. The UNICORE community distributes a simple command-line client that calls the UNICORE Command-Client (UCC), an client with a GUI that is based on the Eclipse Rich Client Platform (RCP) [26] named UNICORE Rich-Client (URC), and the High Level application programming interface (HiLA API) [82] .
Furthermore, a possible scenario to access grid resources is a grid portal [72]. The UNICORE community does not provide a grid portal, but they try to create interoperable interfaces to access resources in UNICORE grid in a standardized manner,

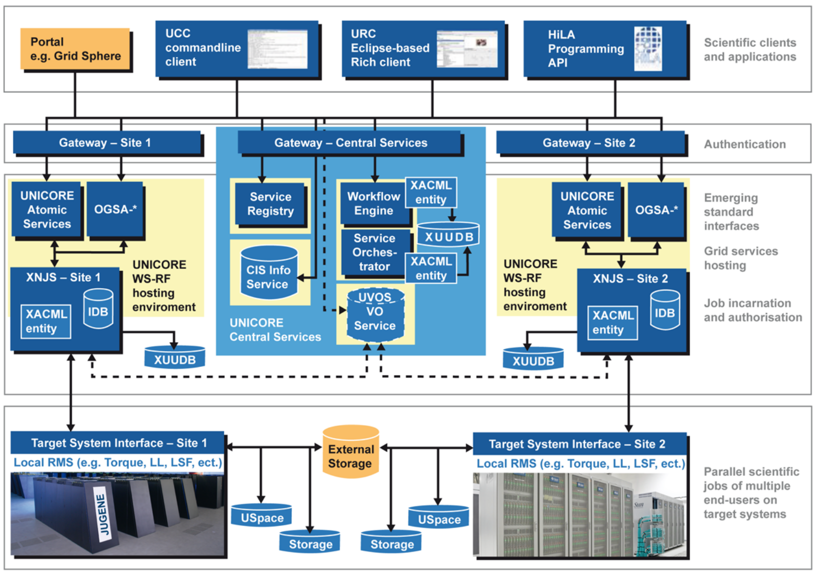
Portal
e.g. Grid Sphere
UCC commandline client
URC Eclipse-based Rich client
HiLA Programming API
Scientific clients and applications
Gateway – Site 1
Gateway – Central Services
Gateway – Site 2
Authentication
UNICORE Atomic Services
OGSA-*
XNJS – Site 1
XACML entity
IDB
UNICORE WS-RF hosting enviroment
XUUDB
Service Registry
CIS Info Service
Workflow Engine
XACML entity
XUUDB
Service Orches-trator
XACML entity
UVOS VO Service
UNICORE Central Services
Emerging standard interfaces
Grid services hosting
Job incarnation and authorisation
Target System Interface – Site 1
Local RMS (e.g. Torque, LL, LSF, ect.)
JUGENE
External Storage
USpace
Storage
Target System Interface – Site 2
Local RMS (e.g. Torque, LL, LSF, ect.)
Parallel scientific jobs of multiple end-users on target systems

Figure 2.3: UNICORE Architecture, [67]

for example the HiLA. Compatible portals for the UNICORE middleware are, for example, the Vine Toolkit [65] and GridSphere [32].

Service layer The service layer is the heart of the grid middleware, “all services and components of the UNICORE Service-Oriented Architecture (SOA) based on WS-RF 1.2, SOAP, and WS-I standards “are arranged in the service layer. Main components in the UNICORE6 grid middleware are:

- Gateway
  The gateway is the main entry point to the grid. The gateway needs only one port in a firewall configuration to communicate with the outside of a secured local area network.
- Registry
  The registry is the central registration point of the UNICORE grid. After the registration of a service, this service is viewable to all other grid services in the grid. For distributed UNICORE grid infrastructures a registry is obsolete, because all clients access the registry for accesible resources.
- Workflow Engine and Service Orchestrator
  The Service Orchestrator executes jobs in a workflow. Both components are

easily replaceable or configurable.

- UNICORE Virtual Organization Service (UVOS)
  Provides user management upon the SAML standard. Can replace the xuudb.
- XNJS with UNICORE Atomic Service (UAS) and OGSA
  The XNJS is the heart of every UNICORE site and is responsible for job management and execution.
- XUUDB
  The XUUDB is the user database and is responsible for the user authorization with X.509 certificates based on eXtensible Access Control Markup Language (XACML) policies.
- Incarnation Database (IDB)
  The IDB translates from abstract Job Submission Description Language (JSDL) job definitions to resource specific commands.
- Target System Interface (TSI)
  The TSI component translates grid commands into system-specific commands.

System layer In the system layer resides the communication with the operating system and resource specific services. The TSI component translates grid jobs to local system-specific commands. For every job exists an environment such as a working directory that can be used for input files and output files or error files and system variables. TSI provides user abstraction, while the TSI runs with root privileges that make it possible for the TSI to change between user accounts and user account environment e.g., home directory and execute all services with the needed privileges.
While a job starts on the base operating system, normally a LRM system executes. In UNICORE, many LRM systems are supported, e.g. LRMS, Torque, LoadLeveler, LSF, SLURM, and OpenCCS.

In Figure 2.3, we show a multi site installation of UNICORE6. Behind the Gateway - Central Services resides the central service point for user access, information management, and workflow services. While the site one on the left and site two on the right are single UNICORE sites, they register their resources at the middle central service point. The user interacts directly with the central service point.
In the year 2011, the UNICORE 6 architecture is based on the requirements of the OGSA, while the implementation is assisted by the WS-RF. According to the UNICORE community "UNICORE 6 is currently the most up-to-date implementation of the core specifications such as WS-RF.

2.3 Cloud Computing

Today, everything is connected with cloud computing. Microsoft and Toyota will build a cloud car: "vehicles will be able to connect via the cloud to control and monitor their car from anywhere - safely and conveniently "[58]. IT-companies offer many different cloud services, for example, Apple connects their different hardware products as mobile phones, desktop computers, and multimedia devices over the cloud [14]. This leads to the possibility to synchronize data between all hardware products and access them over the internet. Google offers a free office suite with, email, computing infrastructure and multimedia services in the cloud. Even simple subversion-like storage services as drop box that synchronizes files across different operating systems and hardware devices with internet access [25] are available. By this variable use of the term "cloud computing ", a clear definition is convenient, otherwise cloud computing arouses suspicion of a good marketing.

2.3.1 Characteristics of Cloud Computing

The NIST [78] provides a widely accepted definition for cloud-computing that defines cloud computing as; "a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction ". This definition clears that cloud computing is more a new computing paradigm, than a new technology. Mainly the user needs were guidance to the cloud computing architecture. To realize cloud computing implementations, common technologies are combined. These include as web services, virtualization, parallelized and distributed computing. Together, these techniques form the service oriented cloud computing paradigm.
Depending on the type of delivered service, cloud services are categorized into three different layers. With every layer, the technical abstraction gets more complex. These layers comprise these service models and are from top to bottom the Software as a Service (SaaS) layer that aims to end-uses, the Platform as a Service (PaaS) layer that aims to developers and the Infrastructure as a Service (IaaS) layer at the bottom that aims to network architects. In the following list we provide a detailed explanation of every level:

SaaS The SaaS architecture is the most abstract service model of all three cloud services models. Users access ready installed software in the cloud. The user does not have to manage or administrate the software or the underlying systems, the focus lies on usability. Furthermore, the software takes advantages from the dynamic resource allocation in the cloud that makes it possible to scale the software to the users' requests. Examples for SaaS are Google Docs and eyeOS [27] that provides a complete web desktop in the cloud.

PaaS The PaaS architecture offers abstract access to system resources. This model offers a ready configured computational environment without direct system access. This enables the user to deploy and run applications on pre configured computing resources. The user can focus on the development of applications and is not engaged to run the underlying computational infrastructure. The resource requested by the user are encapsulated into provider specific APIs that are event orientated and not architecture orientated. For example, an API database request hides the underlying database implementation. Through this abstraction, it is possible to gain a high flexibility in scalability and technical implementation in this service layer. Examples for PaaS are Microsoft Azure and Google App Engine.

IaaS The IaaS model provides computer resources to users with the focus on customizability. In this service model, it is possible to allocate customized resources like storage, processing power and software. All involved resources depend on virtualization. Therefore, it is possible to configure all computing instances on the users preferences and achieve a high scalability in the possible hardware configurations. The IaaS model makes it possible to be independent from long-term hardware investments and it is possible to rent customized needed hardware environment and pay only for used resources.Example for IaaS are AWS [15] and Rackspace [50].

The three cloud service models, SaaS, PaaS, and IaaS can deploy their resources in different cloud deployment models. The **public cloud** deployment model provides access to public usage and can be free of charge or for payment. Often, a free access with limited resources is available. For example Amazon, Microsoft or Google offer their cloud services with limited resources for free. **Private clouds** are enclosed to public usage. Their service is only used by the owning corporation, for example, eBay has it's own private cloud or the private cloud service is limited to dedicated user, for example Ubuntu One [60]. Combinations of the public and private clouds are called **hybrid clouds**. Hybrid clouds separate different parts of the cloud to public and private access and achieve the needed control over this parts. The **community cloud** is under collaborative use by different interest groups.
Cloud computing is based on utility computing in a SOA environment. Contracts between service providers and users are based on Service Level Agreement (SLA)s. The enabling technique for cloud computing is virtualization. Abstract requested resources from the underlying hardware enable a wide range of products for cloud users and vendors [79].

2.3.2 Amazon Web Services

AWS is an arrangement of web services that form a cloud platform. It was launched in July 2002 by Amazon.com Inc.[7] and has grown to more than 20 different cloud services in the year 2011. These services reach from simple storage and monitoring services over database services to more complex computing services that make it possible to build complex applications in the AWS cloud environment. The AWS cloud infrastructure mainly focuses on the IaaS and PaaS service models with the main focus on software developers as target group. Thus, AWS achieves these cloud infrastructure or platforms in a flexible way. These cloud computing services offer many advantages to developers. The developers only pay resources they use. Therefore, no big start-up investments with long time financial obligations are necessary to use a huge amount of computing resources. Using services from the cloud can release resources for concentrating on the core business, e.g. on developing the web application and not managing the servers. Thus, this use of the AWS cloud can seen as outsourcing [68]. Furthermore, the AWS cloud services are accessible for a wide range of different developers, providing many different interfaces to platforms and programming languages. These includes mobile platforms like iOS and Android, as well as Java, .NET , and PHP Software Development Kit (SDK). All cloud services are accessible through a web service, the AWS Management Console, and simple console commands for Unix based clients. The most common services groups are [15] :

Compute The compute services provide services in the IaaS model. These include, for example the Amazon Elastic Compute Cloud (EC2) that offers flexible and scalable compute resources in the cloud. These compute resources include an automatic load balancing service, the Amazon Auto Scaling service and services for handling and processing huge amount of data, the Elastic MapReduce service. Fees for all compute services apply only for used resources on hourly rates.

Database The Amazon database services contain the SimpleDB and Relational Database Service (RDS) that offer database services in the PaaS model.

Networking Amazon offers Domain Name Service (DNS) network services with the Amazon Route 53 service, a mechanism to separate isolated parts of the Amazon cloud for private use with the Virtual Private Cloud (VPC) service, and a Elastic Load Balancing service that automatically distribute incoming traffic across multiple AWS cloud computing resources. All networking services cooperate with the compute services and are located in the IaaS model.

Storage The storage services provide redundant data storage that is accessible over the web by Amazon Simple Storage Service (S3). The persistent storage for EC2 resources over their incarnation time offers the Amazon Elastic Block Store (EBS) service.

Deployment and Management The Deployment and Management services deploy and manage applications in the cloud. The Elastic Beanstalk service automatically handles the deployment details of capacity provisioning, load balancing, auto scaling, and application health monitoring. The Deployment and Management service are examples for the PaaS model.

Monitoring The Amazon CloudWatch service is a web service that monitors Amazon EC2, Amazon RDS, and Amazon Data Base resources. This service is an example for a SaaS model.

3 Designs for Extending Grid Systems with Cloud Resources

Grid systems connect local separated resources from distributed computer systems [74] to provide access to these resources through a single access point. This access point is the grid system. However, in cloud computing environments it is possible to allocate user-defined computing resources and computer systems that comprehend, for example, compute storage, networks services, databases and even special High Performance Computing (HPC) infrastructures. Thus, extending grid systems with user-defined allocated resources from cloud computing environments seems a practicable idea.
In this chapter, we discuss and present general requirements and aspects for such grid and cloud computing combination designs. Furthermore, we discuss the presented designs of grid and cloud systems based on general requirements and aspects and choose one design that we implement in Chapter 4.
However, for all designs in Section 3.3, we made a common decision in the architecture that is equal for all designs. This decision helps to present clear descriptions of our different designs and helps to avoid repeating descriptions of the same facts in all designs. The decision for all designs depends on the design of the grid client software that we use to access our grid with the cloud computing resources. Two different designs for the grid client are possible, which are described in Section 3.1.

3.1 Design of the Grid Client

The basic design decision we made for the grid client to access cloud computing resources, was to extend it with the cloud computing interface and name it Cloud Extension. Shown in Figure 3.1 with the possibility B. The other possibility to design the grid client is to use a standalone application that manages the cloud computing resources for the grid, shown in Figure 3.1 with the possibility A. The most cloud computing vendors offer cloud computing management applications and interfaces, but these do not have the functionality to automatically configure and include cloud computing resources with running grid software into the local grid environment. Therefore, we have to extend existing cloud computing management applications.
In addition, users do not want to use two separated applications to use one system. Grid systems and cloud computing environments are separated systems, but in this thesis we

want to integrate cloud computing resources seamlessly into grid systems. Therefore, grid and cloud computing system should act as one system. And the cloud computing resources should act as a part of the grid and become a grid resource. In contrast to see the cloud computing environment as an independent system.
In addition, to include the cloud computing resources into the grids system-layer, we integrate the management of cloud computing resources into the client-layer of the grid environment. With this we can test whether it is possible to include cloud computing management interfaces into the designs in Section 3.3 and in the implementation in Chapter 4, too.
In summary, it is possible for all designs to separate the cloud extension into a separate application as shown in Figure 3.1 possibility B. As well as include the cloud extension into the grid client as shown in Figure 3.1 possibility A. Because of the discussed reasons we include the cloud extension into the grid client and provide in our designs only the description for the grid client with included cloud extension. In addition, we want to avoid iterative passages in the description of our possible designs.

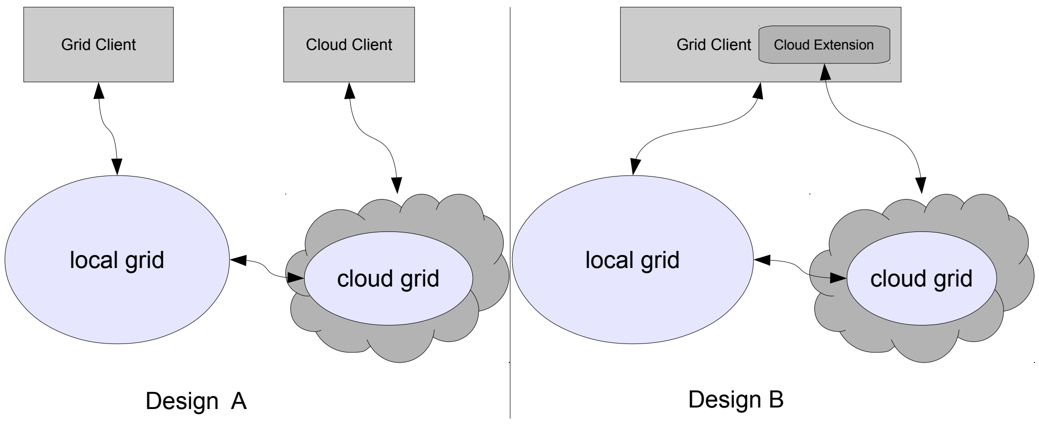

Figure 3.1: Possible designs of the Grid Client

3.2 General Requirements and Aspects

In this section, we give an overview of general requirements and aspects that can be relevant within the combination of grid and cloud computing environments. We can not ensure that we discuss all requirements and aspects for all possible users and use cases. However, we discuss a list of common issues that mainly focus on cloud computing aspects because we want to use cloud computing resources in grid systems. This approach should give a guidance for all users and use cases to see what kind of requirements and aspects are important for individual users and their according use cases. For us, this overview of requirements and aspects provides a basis for a discussion on which we evaluate our designs in Section 3.3 and our implementations in Chapter 4.

Management of cloud resources Cloud computing resource management is arranged over different levels of abstraction. Possible levels are, for example, web interfaces and web applications, desktop applications, direct console access within a common OS command shell, or the embedding in programming languages with special APIs. Thus, these levels require different knowledge from the user and can require specific technologies. Web applications are accessible across different operating systems, standalone desktop applications can only be developed for a specific operating system, and APIs depend on a specific programming language. Therefore, the different pre-conditions and particular objectives from users in the management of cloud-computing resources should lead to the right management solution.

Interoperability between clouds The interoperability between cloud computing environments is in development. In the year 2011, many different Standards Development Organizations (SDO) that are working on cloud-computing standards exist. The Institute of Electrical and Electronics Engineers (IEEE) started two working groups on cloud computing interoperability, the Intercloud Working Group (ICWG) [37] and the Cloud Portability and Interoperability Profiles (CPIP) [36] working group. Furthermore, the Object Management Group (OMG) [42] founded the Cloud Standards Customer Council (CSCC) [22], which is a consortium of different interest groups that have the goal to standardize the cloud computing. The OMG provides an overview of standardizing activities in a wiki, some of the CSCC members work on the cloud computing interoperability aspect. For example, the Interoperable Clouds White Paper from the Distributed Management Task Force (DMTF) or the Open Cloud Computing Interface Core (OCCI) specification from the OGF contain information and solutions for the cloud computing interoperability, for example several implementations exist for the OCCI [40]. Even for different cloud computing APIs exist the libcloud [13] and deltacloud [24] that abstract differences among different cloud computing APIs and make it possible to access simultaneously different cloud computing APIs through one library. The Apache libcloud library is released in the version 0.5 in July 2011 and became an Apache top level Project in May 2011, we think the libcloud library and similar projects can be potentially great. However, the development in cloud interoperability in the year 2011 is still in its infancy.

Reliability of cloud resources Cloud computing resources are not under the physical control of the user. They reside in computer centres that are under the control of cloud computing vendors. These computer centres can be distributed around the world. Often these computer centres act as black boxes that offer defined services without

the user knowing how this service is realized inside the computer centres. Because of the absence of information about the internal operation principles of cloud computing services, scepticism towards the reliability of cloud computing services and resources arise. In addition, the distribution of cloud computing centres around the world in different countries can cause reliability problems. Data in cloud computing centres that reside physically in the United States of America (USA) is accessible by the government of the USA [38] and it is possible that this data in the cloud can be confiscated without provided reason. The government of the USA can forbid the cloud computing vendor to contact the data owner about the reason or any other details of the confiscation. Furthermore, data in cloud computing centres that reside in the European Union can be confiscated if the cloud computing vendor reside in the USA [38]. For example, Microsoft informed cloud computing users that they disclose data to satisfy legal requirements, or protect the rights or property of Microsoft or other [39].
Furthermore, error-free hardware and software does not exist. In contrast, hardware in server centres crashes on a regular basis [83]. Because of this failure rate, cloud computing vendors do not offer a hundred percent durability. Only mechanisms to enhance the durability and trust in cloud computing is offered to the user. These mechanisms can be transparency from the vendor, SLAs, technical solution as for example backup systems, and a personal contact person that can cultivate a personal trust.
In summary, the reliability of cloud computing resources is a very complex issue. Problems can occur in legal circumstances, hardware issues or maybe human failures. Furthermore, issues exist that cannot be influenced by cloud computing vendor and user.

Cost of cloud resources The price model of cloud computing services has a wide range across vendors. The cloud computing service levels in Section 2.3, do not implement common standards and therefore, a wide range of different cloud computing systems exist and make comparison complicated. The pricing models of the IaaS and PaaS service levels are more compareable among each other than the SaaS service level, because in the IaaS and PaaS service level the user rents mostly computer infrastructure that is more related than software in the SaaS level.
Often, the cloud computing vendors offer monitoring and reporting services that afford a detailed analysis of the occurred costs. These cloud computing usage analysing mechanisms offer the possibility to increase the cloud computing fees, because the analysis can detect unused running resources. But to plan the fees in advance is difficult and a detailed knowledge of the future use must exist. To plan the approximated cost, cloud computing vendors offer price comparison tools, for example AWS simple Monthly Calculator [20], Rackspace [48], and GoGrid [31].

In general, the costs of cloud computing must be compared to the investment in own hardware, because after a period of time the turning point of investment in cloud computing resources can occur and the investment in own hardware can be more profitable than paying constant fees.

Security aspects "Security is a process, not a product "this comment from Bruce Schneier describes our process to develop a secure design to use cloud computing resources in grid systems. In the process we begin with a less secure solution in Subsection 3.3.2 and at the end we come to a solution in Subsection 3.3.4 that has a architecture that separates the design completely from all public access and therefore promises the most secure solution. The security of grid systems we do not discuss further in this thesis, because we think plenty of secure architectures and concepts for grid systems exists. The problems of security in grid systems lies more in the secure configuration. And for more information about the security of grids we refer to [71] and for the configuration to the manuals of the used grid software.
However, security in cloud computing environments becomes a very fundamental topic for all cloud computing providers and users. To develop security concepts, cloud computing vendors founded the Cloud Security Alliance (CSA) [23] that provides a "Security Guidance for Critical Areas of Cloud Computing ". This security guidance is a framework that helps to evaluate initial cloud risks and informs about security decisions. Basis for this guidance is a detailed analysis of the users' safety requirements, because the risk tolerance for data and services that are used in cloud computing environments can allow or forbid the usage of cloud computing. But in general, the editors of this guidance think that "Cloud Computing is not necessarily more or less secure than your current environment. As with any new technology, it creates new risks and new opportunities. ".
Therefore, we provide secure architectures in this thesis to use cloud computing resources in grid systems, but in the end the decision about the usage of cloud computing resources is at the user and the users' acceptable risk level.

Performance aspects Performance limitations in possible architectures that include cloud computing resources in grid systems can reside in three different domains. First in the cloud computing environment, second in the connection between the cloud computing environment and the local grid, and at last inside the local grid. Performance issues of the local grid and grid systems in general we do not address in this thesis, because it is out of scope.
However, the two other possible performance limitations we discuss. First, the data transfer over the Internet between the local grid and the cloud computing environment can be a bottleneck. This bottleneck over a network that is mostly not under

the physical control of the grid user can vary on the amount of data that have to be transferred to the cloud environment. In addition, transfer rates rhough the public Internet are not constant. The user must decide how the limitation of the data transfer over the Internet constrains the usability of the cloud computing resources. For example, the quotient of the data to send and the time this data is used in job execution in the cloud can differ and therefore makes the data transfer rates more or less important.
The second domain of performance limitations is the cloud computing environment that is also not under the user's control. The performance in the cloud computing environment correlates with the performance of single cloud resources, their connection and the maximal amount of cloud resources. All performance issues differ between cloud computing vendors. In general, many different single cloud computing resources are possible. These can range from dedicated machines to slow virtualized ones. But the connection between single cloud computing resources can not be controlled by the user. At last, the amount of a cloud computing environment can be a performance limitation, but we think a use case that exhausts the amount of cloud computing resources from a special cloud computing vendor is unusual.
Therefore, both domains can not be completely controlled by the user. It should be possible to create a solution, where these issues do not limit the usability.

Requirements of the local network A fundamental requirement of the local network is a connection to the Internet. This is an essential requirement. If the local network resides in an isolated environment with the leak of a possible connection to the Internet, it is not possible to include cloud computing resources to the local grid, because all cloud computing resources are only accessible over the Internet. Otherwise, the connection to the Internet can be subject to a strict security policy which does not allow connections to the cloud computing environment.
However, the connection between the local network and the cloud computing environment must be physically possible and no policy or other issues should restrict the connection to the Internet and the cloud computing environment. We do not discuss the configuration of the local network, because this is appointed by the technical specification and requirements of the grid software and the cloud computing environment.

Amount of cloud resources Infinity cloud computing resources are good marketing slogans but are rather not the reality. No cloud computing vendor publishes the amount of resources that exist in the vendor's cloud environment. In addition, the cloud computing vendors avoid to specify the maximal usable resources for users. To guarantee the user the boot-up and usage of a specific amount of resources specific SLAs

between the cloud computing user and the cloud computing vendor are possible. The SLA about resource requests differs with the demand of the user. Furthermore, SLAs that guarantee specific time intervals between resource requests and resource allocations can increase the cost of cloud computing resources.
Our solution to avoid bottlenecks in the local grid environment is to add on demand cloud computing resources to the local grid. The user can decide this. Furthermore, the user has to find a cloud computing provider that is able to allocate the needed amount of cloud resources.

The discussed general requirements and aspects demonstrate that many different problems can occur in designs that include cloud computing resources in grid systems. However, the user can handle many of the issues with an analysis of the estimated usage design of cloud computing resources in grid systems. Furthermore, an abstract design for cloud computing usage in grid systems is needed to avoid problems in the first place because of its architecture. In Section 3.3, we provide abstract designs that include cloud computing resources in grid systems. In the Section 3.4 we discuss these abstract designs based on the general requirements and aspects from this section and analyze the issues that can occur in our abstract designs.

3.3 Extend the Grid by Cloud Computing

In this section, we discuss and develop abstract and possible designs that extend grid systems with cloud computing resources. In Section 3.2, we provide some general arguments and requirements that give us a frame to classify these designs in Subsection 3.4.1.

3.3.1 Terminologies for the Grid and Cloud Combination

In the descriptions of all different designs, shown in Subsection 3.3.2, 3.3.3, and 3.3.4, for the grid and cloud combination reside components that are always the same. Therefore, we describe them in the following:

Gateway The gateway component is the entry point to a grid environment. The gateway component can reside in the local grid as well as in the cloud computing environment. Behind a grid component all other grid components can hide and be invisible from the outside and only the gateway component must be reachable to access the resources behind the gateway. Furthermore, different gateway components can communicate with each other over a secure connection.

Cloud Interface Cloud computing vendors mostly offer propriety APIs or other common interfaces that rely on basic web technologies. These interfaces reside in a SOA environment and allow to manage the cloud computing resources from out different programming languages and operating systems. An example for a cloud interface is the AWS API, the libcloud library, or the deltacloud library. We provide more discussion about interoperability between clouds in Section 3.2.

Grid (local Grid) The local grid is a grid environment with a fixed size and has all components that a grid system of a special type needs to work probably. Furthermore, the local grid is able to include compatible grid environments and grid resources from locally separated grid systems. The configuration of the local grid never changes in all designs and an example for a local grid is UNICORE 6.

Cloud Grid The cloud grid is a grid environment that resides in a cloud computing environment. The grid in the cloud can consist of single or multiple grid resources and a gateway can hide the grid resources in the cloud. In general, we describe in our designs how it is possible to integrate these cloud grid into the local grid.

Cloud resources Cloud resources can be all different types of computing resources that reside in the cloud computing environment. For example, cloud resources can be networks specific management interfaces and computing resources as instances. The term instance describes computing cloud resources at the IaaS model that are comparable with server systems. It is possible to run grid software on these resources.

Grid resources in the cloud All types of common grid services and resources can reside in the cloud computing environment. Therefore, grid resources in the cloud are, for example, the gateway, storage, or computing services.

VPN-Tunnel The VPN - Tunnel is an encrypted connection that connects different computer networks. This secure connection can be realized with a wide range of different software and hardware components.

In the following, we describe abstract designs to integrate cloud computing resources into grid systems. In all designs the grid cloud differs in the connection to the local grid, the configuration and arrangement in the cloud.

3.3.2 Local Grid with Public Instances from the Public Cloud

The first design is shown in Figure 3.2. This design contains one gateway that resides in the local grid. This gateway acts as the central entry point for the grid client. Therefore, all grid related communication as grid jobs, job information, grid storage usage, and etc. flows through this gateway. Furthermore, this design contains a cloud interface, to enable the cloud extension to configure, start or stop, and manage the cloud computing environment and the grid resources in the cloud.

In this design, the cloud grid resides in the public domain of the cloud computing environment. Without security arrangements or technically separation from the other public cloud computing resources, the allocated cloud grid resources are visible for all cloud users. If the public domain in the cloud computing environment is accessible from the Internet, then all user from the Internet can access the cloud grid resources.
To protect cloud resources, cloud computing environments offer security concepts, for example, firewalls on instance level or firewalls on network level in the cloud. In this design every resource needs a special security configuration, because it resides in the public domain of the cloud. In addition, to configure every resource separately causes configuration overhead and therefore security failures can creep in. Furthermore, the software on the cloud resources can contain security software to ensure the security of the single cloud resources. But to configure these security software on every single resources, needs a very detailed security analysis. Therefore, failures in the security configuration can appear, too. The complex security configuration requirements in this design cause security issues. Furthermore, the cloud computing environment, the software on the single instances, and the grid software must support all needed security techniques to configure a secure design. To overcome this configuration overhead and security requirements, we developed the next design 3.3.3 that improves this design.

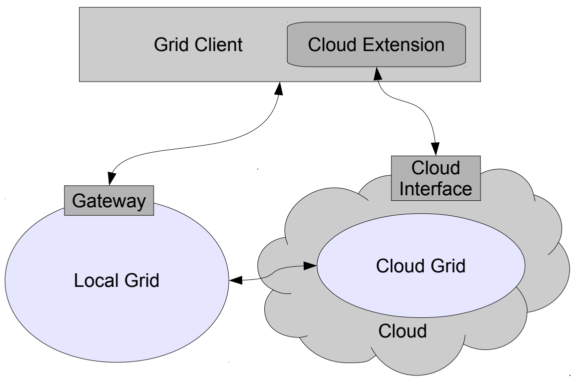

Figure 3.2: Grid in the open Cloud

3.3.3 Local Grid with Instances from the Private Cloud.

The second design is shown in Figure 3.4. This design contains two gateways that connect the grid resources in the cloud to the local grid. The gateway at the local grid is the central entry point for the grid client similar to the first design in Subsection 3.3.2. The gateway in the cloud computing environment connects the grid resources in the cloud with the local grid. The traffic from the grid resources in the cloud flows through the cloud gateway to

the gateway in the local grid. Grid resource in the cloud do not send data besides the cloud gateway to the local grid. In addition, the cloud interface in this design that has the same functionality as the cloud interface in the design in Subsection 3.3.2.
The difference to the first design in Subsection 3.3.2 is the separation from the cloud grid resources to the public cloud. We created a private cloud inside the public cloud computing environment that contains all grid resources. This private cloud avoids public access to the grid resources in the cloud. Therefore, this design approach promises a more secure infrastructure. The private cloud is a technically separated environment inside the cloud and only the gateway is visible.
In this design we overcome the security configuration overhead by creating a secure infrastructure. Only the communication between the both gateways uses the Internet and is not under our control. Therefore this can be a security issue. To secure this communication, the grid middleware has to support encrypted gateway-to-gateway communication. In the next design we provide an infrastructure that avoids public communication over the Internet.

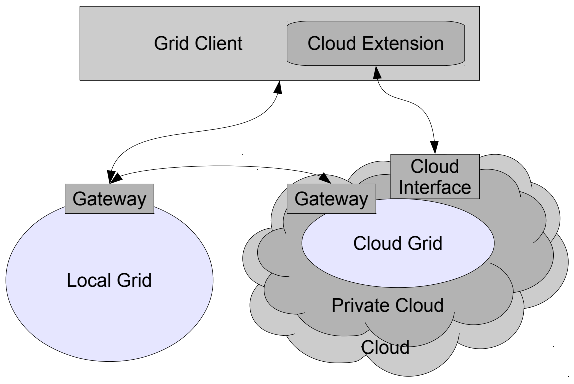

Figure 3.3: Grid with Private Cloud

3.3.4 Local Grid with Instances from the Private Cloud over a VPN-Tunnel

The third design is shown in the Figure 3.4. This design contains one gateway and one cloud interface similar to both designs in Subsections 3.3.2 and 3.3.3. In addition, all grid resources in the cloud computing environment reside in a private cloud similar to the second design in Subsection 3.3.3. The connection between the local grid and the cloud grid is different to the previous designs. Because in this design the cloud grid is only accessible over an encrypted connection between the local grid and the private cloud. Therefore, the private cloud network extends the local network. All traffic in and out of the private cloud flows through the encrypted VPN - tunnel between the local grid and the private cloud.

This design avoids both security issues from the previous designs. Compared to the first design, the cloud grid in the private cloud are not visible to all other cloud user as in the first design in Subsection 3.3.2. In addition, the communication between the local grid and the private cloud uses the Internet as in the second design in Subsection 3.3.3. But this traffic is encapsulated from the public network by a VPN - tunnel. Therefore, this design provides a complete detachment from public networks. Grid resources from the local grid and the private cloud are arranged together in an encapsulated network. The cloud resources appear as local resources inside the local grid network. All cloud resources can be managed and configured similar to local grid network resources. Therefore, this design provides the most secure architecture.

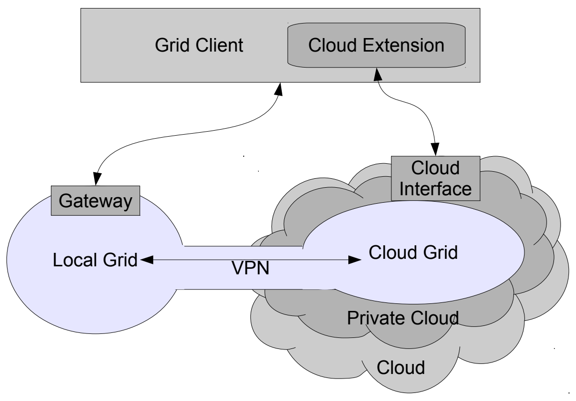

Figure 3.4: Grid with Private Cloud

3.4 Discussion

In this section, we discuss our developed designs from Section 3.3 that integrate cloud computing resources in grid systems. The frame and basis for our discussion and analyse are the general requirements and aspects from Section 3.2. In Subsection 3.4.2 we chose the design that we implement in Chapter 4.

3.4.1 Analysis of the Designs for Extending Grid Systems with Cloud Resources

In this subsection we do not want to assign the best design, because the requirements differ between users and their corresponding use cases. But to provide an assessment can help users to classify the designs and find the best design for their usage environment. The following key aspects are based on the discussion in Section 3.2 where we discussed them in general.

Management of cloud resources In our designs, we manage the cloud computing resources with a cloud extension that resides in our grid client. To realize the cloud extension, we have to include a cloud management tool into it. This cloud management tool can be, for example, a web interface, a desktop application, or an cloud management API that depends on the abstraction level of the grid client. In summary, the cloud management service has to be compatible with our grid client.

Interoperability between clouds In our designs, we did not define an exact specification for our cloud interface, we assume that it has all functionality to manage, configure, start and stop all cloud and grid resources. But without a clear definition of the cloud interface it is barely possible to identify all cloud computing environments that comprehend the functionality to implement our cloud interface. If we found different cloud computing environments for our cloud interface, these could be exchanged theoretically. However, we think the interoperability approaches between cloud computing environments seem to provide the possibility to use arbitrary cloud computing environments, but without a clear specification this will be difficult.

Reliability of cloud resources The cloud can crash under two different circumstances. First the cloud grid runs and contains user data. Second the cloud grid is in idle mode or not started and does not contain user data. If the grid cloud crashes during the running process, this could lead to undefined conditions and in the worst case the complete uptime of the cloud grid is useless. Furthermore, in our designs we did not develop rescue or safety mechanism that can restore data after a cloud crash. If none of the resources are running, even then a cloud crash can cause problems. For example, pre-configured images, or other data that is only stored in the cloud get lost. But to avoid problems of a not running grid in the cloud seems to be easier, because, for example, all data can be backed up in the local network.

Cost of cloud resources Cost of cloud resources correspond with the amount of usage. But the cost differ between the designs, because in the second and third design we use the private cloud and the VPN-tunnel feature that might be charged with extra cost from a special cloud computing vendor. In summary, this possible extra cost is quite small compared to the cost of the usage of cloud computing resources.

Security aspects With the three designs, we developed a increasingly secure infrastructure for the grid in the cloud. The last design minimizes the possible security leaks, but has the most configuration and cloud computing feature requirements. Furthermore, the security can differ across cloud computing vendors and grid systems. For example, the cloud interface can be encrypted, or the private cloud can be completely separated from the public cloud. The security in an implementation of our designs relies on the used technologies and their configuration.

Performance aspects Performance differences between our three designs do not exist in general. In Section 3.2, we discuss the performance aspect in general and analyze three different domains towards performance limitations. The implementation of the three different designs will make the difference. For example, a VPN-tunnel can slow down the traffic speed, or the transfer over the Internet. Furthermore, a possible improvement can be to configure multiple gateways.

Requirements of the local Network The requirements of the local network differ between the three designs. In the first and second design, the architecture of our designs has no requirements to the local network. A special case can occur in the first design, if the grid resources in the cloud computing environment are configured to communicate around the gateway directly with the local grid and this communication is not allowed in the local grid network. But this should only be a configuration.
However, the architecture of the third design needs a VPN connection point at the local network that can adopt the VPN - tunnel. This connection point can be a hardware or software component and relies on the used technology.

Amount of cloud resources In general, all three designs should be able to use the same amount of cloud resources and therefore the limits are set by the cloud computing vendor. A difference can exist between the amount of cloud computing resources in the public and private cloud, but that shall differ across the cloud computing vendors.

3.4.2 Choosing a Design

We selected the second design for our implementation. As we analyzed in Subsection 3.4.1 the architectures of the different designs from Section 3.3 have advantages and disadvantages. Design one needs too much security configuration. And based on the security issues from design one, we developed the second design with a more secure architecture in the cloud computing environment.
The infrastructure from the second design provides a separated environment for cloud computing resources and therefore promises a more secure implementation. In addition, the second design promises more interesting aspects in the implementation process, because more functionalities then the grid gateway and the private cloud are used. In contrast to the first model, where only the instances have to be configured and started what is very similar to create local grid resources, in the second design we have to create a complete infrastructure in the cloud with the private cloud that contains the grid resources.

The third design promises the most secure solution, because all resources and traffic is encapsulated from public access. But as we analyzed in the previous subsection, the third design has the most requirements to the local network, extra cost for the VPN-tunnel can occur, and questions in the performance aspects can arise. Furthermore, the third design seems to be very complex to implement, because of the configuration of the VPN-tunnel. The transfer between gateway and gateway as in the second design can promise a secure data transport if the grid system provides encrypted data transport. Normally, a grid system should provide encrypted data transport, because grid systems are developed to connect distributed computing resources.

4 Implementation of the Local Grid with Instances from the Private Cloud

In this chapter, we provide a description for the implementation of the chosen design from Subsection 3.4.2. In Section 4.1, we give an overview of the implementation design and the concrete technologies we use. In Section 4.2, we provide explanations for terms we use in Chapter 4. Afterwards, we introduce detailed designs for the chosen scenario in Section 4.3.
In the Sections 4.4, we introduce the design of the grid client with the Cloud Extension. In the Section 4.5, we describe the implementation of the Cloud Extension. In Subsection 4.5.1, we cover the implementation for the cloud functionality of the Cloud Extension. In Subsection 4.5.2 we provide the description for the grid functionality of the Cloud Extension. In Subsection 4.5.3, we provide the possible usage of cluster resources in the cloud computing environment. In the last Subsection 4.5.4, we describe the security implementation of the Cloud Extension.
At the end of Chapter 4, we provide a infrastructure and job execution validation for our implementation in Section 4.6. In Section 4.7, we complete Chapter 4 with a discussion.

4.1 Overview

The Figure 4.1 shows the abstract architecture of the implementation that extends grid systems with cloud computing resources. With Figure 4.1, we introduce technical components of the implementation to provide an overview and link to the abstract scenario chosen from Subsection 3.4.2. In Section 4.3, we provide a detailed explanation of the different components and the concrete implementation.

To access the local grid depricted in Figure 4.1 that contains a UNICORE6 grid system, we use the UNICORE6 HiLA shell client within the Cloud Extension which is described in Subsection 4.4. But in a nutshell, the main function of the Cloud-Extension that reside in the HiLA shell is to communicate with the cloud computing services inside the cloud and integrate the cloud resources into the local grid system. However, as cloud service we use the AWS described in Subsection 2.3.2 that provides services in the IaaS cloud model. For all UNICORE6 services, the local grid provides a central registry, where every UNICORE6 server, either from the local grid or from the cloud, registers. For user registration, we do not provide a central component. Therefore, the XUUDB user management component

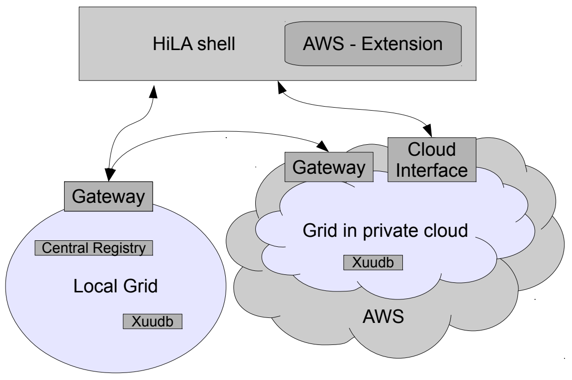

Figure 4.1: Overview HiLA with AWS Extension

exists in both domains, the local grid and the cloud computing environment.
During the implementation process different issues occured that we were not able to solve, because these originate in the technical limitation of the AWS. To overcome these technical limitations in the AWS, we develop different implementation designs for our chosen scenario. In the next section, we provide an explanation of the technical limitations of the AWS and our implementation designs that overcome these limitations.
In this special cloud service, it is only possible to have maximal five fixed IP addresses, that are called Elastic Internet Protocoll (EIP) addresses. EIP addresses make compute resources directly accessible from outside of the virtual private cloud. Resources without an EIP cannot initiate traffic to the Internet, which is needed to interact with the local UNICORE6 grid. With a private subnet and a Network Address Translation (NAT) resource in the virtual private cloud the size of resources that are able to initiate traffic to the internet can be extended as described in the next subsection in detail.

4.2 Terminology for the Grid and Cloud Implementation

Before we explain the implementation in detail, we provide an overview of terms we use in this section. Most of the terms are technical components from the AWS environment:

Instance An instance is an server in the AWS EC2. Different instance types, operating systems, and software packages are available. In addition, we use the term machine simultaneously to instance.

EIP address The EIP addresses are static IP addresses that are provided by AWS. EIP addresses are bound to AWS user accounts and it is possible to associate EIP addresses dynamically to different instances in the AWS cloud.

In the AWS exists two different types of EIP, one associates only to EC2 instances that do not reside in a virtual private cloud and the second can be associated only to EC2 instances that reside in a virtual private cloud. For our scenario we can only use the five EIP that are usable inside a virtual private cloud.

AWS VPC The Amazon VPC isolates instances and cloud computing resources from public access. Furthermore, it is possible to connect the Amazon VPC over a encrypted VPN connection to already existing networks.

AMI An Amazon Machine Image (AMI) is an encrypted machine image that contains information to boot an instance based on it. For example we use an AMI that contains all software to act as a UNICORE6 server. Furthermore, an AMI can be private to a particular user or public to all users.

NAT Instance AWS provides a NAT instance that is a pre-configured instance to perform NAT. The AMI of the instance is ami-d8699bb1 and is always the same.

Availability Zone Availability zones are distinct locations that are engineered to be insulated from failures in other Availability Zones. Furthermore, they provide inexpensive, low latency network connectivity to other Availability Zones in the same Region [5].

Region Regions consist of one or more Availability Zones, are geographically dispersed, and will be in separate geographic areas or countries [5].

Multiple locations Multiple locations are composed out of Regions and Availability Zones. Therefore, AWS provides the ability to place instances in multiple locations [5].

4.3 Design of the Cloud and Grid Configuration

In the AWS technical limitations exist that restrict our implementation possibilities. First, the limit to allocate a maximum of maximal five EIP addresses. Second, the maximum number of allowed subnets in an Amazon VPC. Furthermore, other limitations exist in the AWS that do not influence our implementation. For example, the restriction of 16,500 simultaneously running instances in a VPC.
However, the amount and computing power of AWS instances has a wide range. For example Cluster Computer instances exist with 33,5 EC2 Compute Units and 18GB of memory. One EC2 Computer Unit is equivalent with a 1.0-1.2 GHz Opteron or Xeon Processor from the year 2007. Furthermore, a VPC can contain more than 16,000 instances that promise a huge computing resource and is enough for our implementation.
To overcome the EIP address and VPC limitation, we implemented two different architectures. We name them **Single Subnet Virtual Private Cloud** and **Multi Subnet Virtual**

Private Cloud. In the next paragraphs, we provide an overview of both implementations.

4.3.1 Single Subnet Virtual Private Cloud

The first issue we discuss is the limitation to maximal five EIP addresses. The basis of this issue is that only instances with an associated EIP can initiate traffic out of an Amazon VPC subnet. With the EIP address limitation, we created the first solution for the chosen scenario from Subsection 3.4.2. The Figure 4.2 shows the architecture of the implementation. In detail, we configured one subnet with a 16-bit subnet mask in the Single Subnet

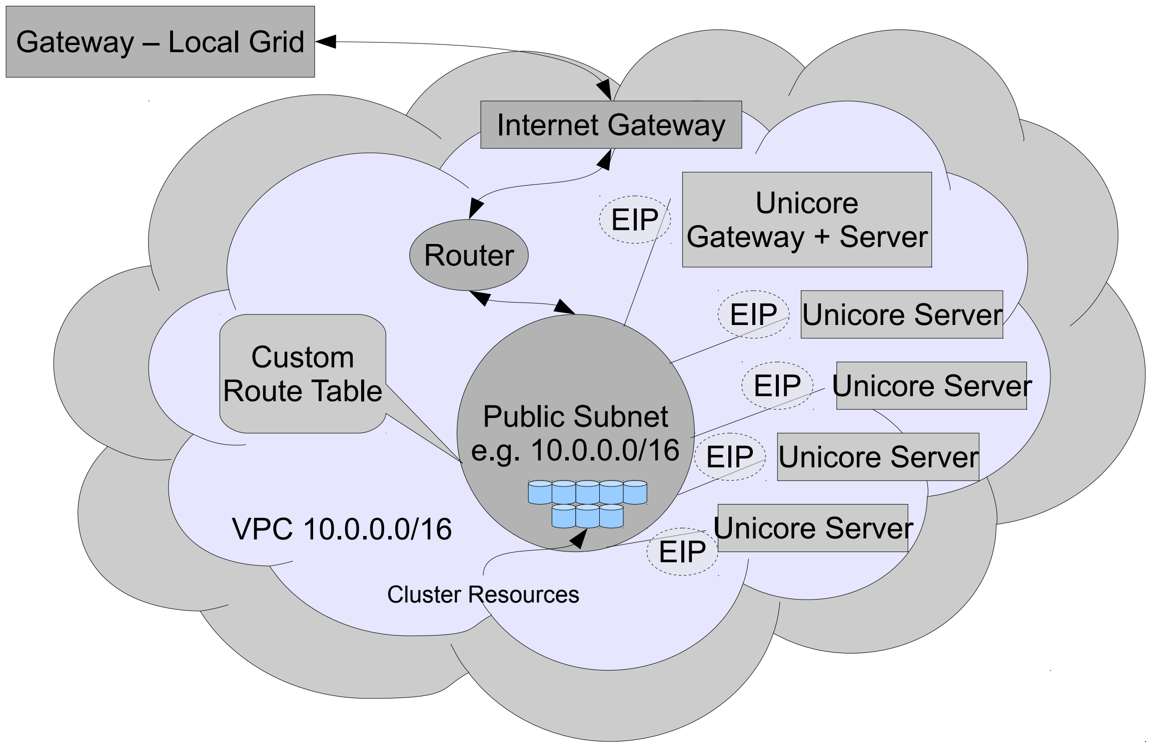

Figure 4.2: Single Subnet VPC

VPC implementation that resides inside the VPC. The subnet can contain at maximum 16,500 instances and 65,536 possible IP addresses. Furthermore, the subnet contains all UNICORE6 resources that reside in the cloud computing environment. In addition, the subnet contains AWS network resources that include a route table and a router. The AWS Internet Gateway is the link to the Internet. Therefore, the communication to the local grid is done over the AWS Internet Gateway. The maximal five UNICORE6 instances are associated with an EIP address. A detailed description of the cloud environment configuration is provided in Subsection 4.5.1.1 and for the grid environment configuration see Subsection 4.5.2.

4.3.2 Multi Subnet Virtual Private Cloud

To overcome the limitation of maximal five grid servers in the VPC subnet, we created the Multi Subnet VPC approach. Figure 4.3 shows a general view of the implementation. The

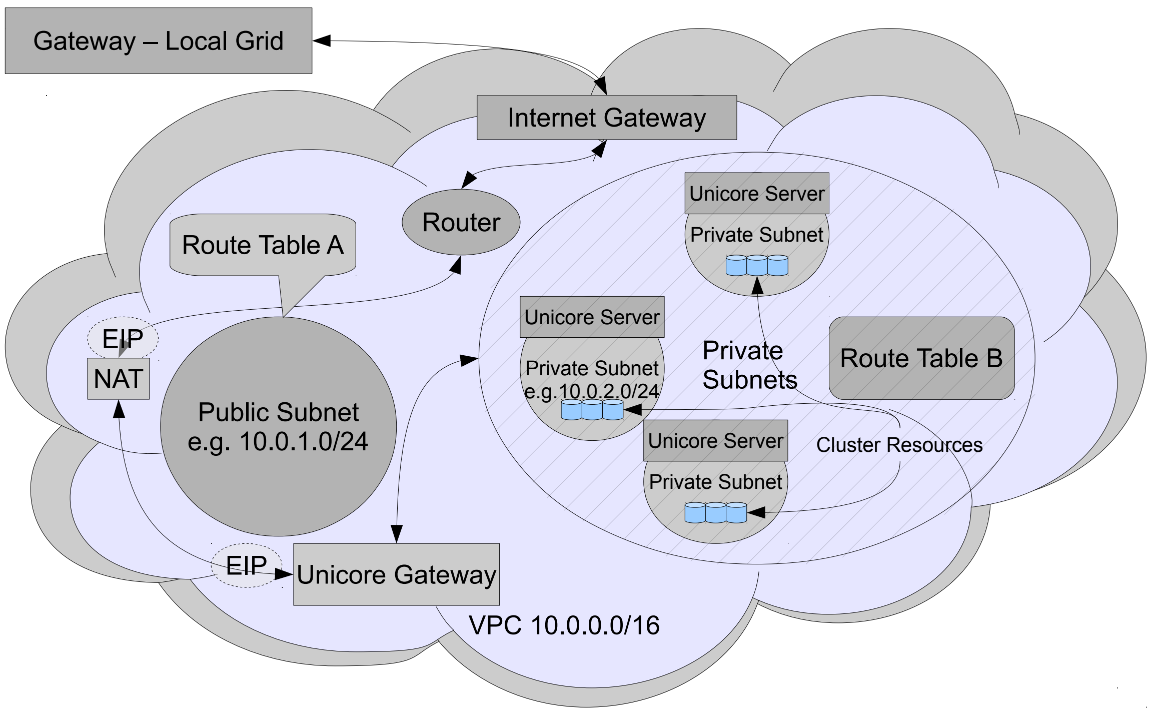

Figure 4.3: Multi Subnet VPC

difference for the user is the possibility to run more than five UNICORE6 servers. In a Multi Subnet implementation, every instance that resides in one of the private subnets can initiate traffic to the Internet without an associated EIP address. To realize the communication between cloud instances and the Internet or the local grid, we created a NAT instance, a Router, and the Route Table A and Route Table B, shown in Figure 4.3. Therefore, it is possible for all instances of public and private subnets to communicate with the Internet. In the Multi Subnet architecture, every instance from the public subnet or the private subnets can act as UNICORE6 server. Therefore, the number of UNICORE6 servers can be enhanced in comparison to the Single Subnet VPC solution. We decided to logically isolate the UNICORE6 servers and configure a separate private subnet for every UNICORE6 server. Because this promises a balanced load, shows the possible usage of the cloud networks, and promises felxibility for the Uniform Interface to Computing Resources 6 (UNICORE6) server arrangement. But this structure is not mandatory, for example, a private subnet can also contain more than one UNICORE6 server. Furthermore, the configuration of the 10.0.0.0/16 VPC network mask offers a complete flexibility to configure and arrange private subnets. We configured 20 private subnets where every subnet has a 24 bit subnet mask and can stud maximum 254 instances.

4.4 Design of the Grid Client - with the Cloud Extension

Clients to access UNICORE grid resources vary from command line clients, API clients, and GUI clients, see Section 2.2.3. All clients have their respective merits. Therefore, we analyzed the which of them is the best choice for us to get extended with the Cloud Extension. The Cloud Extension has mainly two functions: managing UNICORE resources in the AWS and managing the AWS cloud environment. The preconditions and requirements of such a client need are described in the following. The client must support the embedding of the AWS library. Furthermore, the client should be easy to use but provide common grid operations to utilize all grid functionalities. In addition, the underlying code of the client should be extendable.
For our implementation a grid client on the lowest layer of abstraction promised a fast development. Therefore, we do not have to care about GUI programming. In addition, the client could be extended with a GUI later on.
However, the UNICORE community provides the HiLA shell that is a grid client without GUI. Furthermore the HiLA shell is open source, programmed in Java, and has a object oriented code structure that is based on the HiLA API [82].
Concluding, the HiLA shell programming does not require GUI programming. However, it promises all functionality similar to the UNICORE GUI clients, and its code can be extended. Therefore, we use the HiLA shell for our grid client and to progam its cloud extension.

HiLA - the base of the grid client

The HiLA API promises a programmatically infrastructure to access grid resources with a minimal programming overhead. To access grid resources, the same tedious and repetitive tasks are necessary; the HiLA API hides these tasks behind interfaces. For different grid systems these interfaces are implemented including the ones for UNICORE 6 and UNICORE 5 grid systems. An implementation for the OGSA Basic Execution Service (BES) are

in development [73], [82]. Furthermore, implementations for other grid systems are possible, for example the Globus Toolkit or g-lite.
In detail, the HiLA interface methods to access grid environments follow a resource oriented structure, rather than a task oriented. A grid environment includes different resources, for example, the grid, sites, storages, files, and tasks. These grid resources depend on each other. A grid site consists of storages, and storages consist out of files. This dependency structure creates a navigable hierarchy of grid resources, as shown in Figure 4.4. First the user enters a grid, then one of the storage resources, and for example, copies a file from the storages location to another storage location inside another grid. This architecture approach makes grid environments as browsable as common file system hierarchies. The UNICORE community provides a shell client based on the HiLA API to browse grid environments. This shell client is the HiLA shell that we use in this thesis.

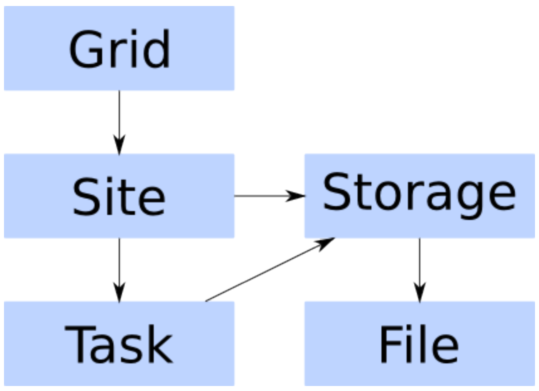

Figure 4.4: Architecture Overview of HiLA [82]

4.5 Implementation of the Cloud Extension

We develop the Cloud Extension as a package of java classes inside the HiLA shell application. Figure 4.5 shows all classes of the Cloud Extension package. The package contains the `CreateSingleSubnetVPC` class and the `CreateMultiSubnetVPC` class that include the application logic. During the runtime the classes `SingleSubnetVPC` and `MultiSubnetVPC` save runtime specific data for the corresponding operation. Furthermore, the class `EC2Methods` manages the AWS resources and the `UnicoreConnection` class manages the UNICORE6 resources. The class `SHHToEC2` provides functions to connect over the ssh protocol to the AWS instances. In addition, the helper classes `ConfigHelper`, `ElasticIPAddress`, and

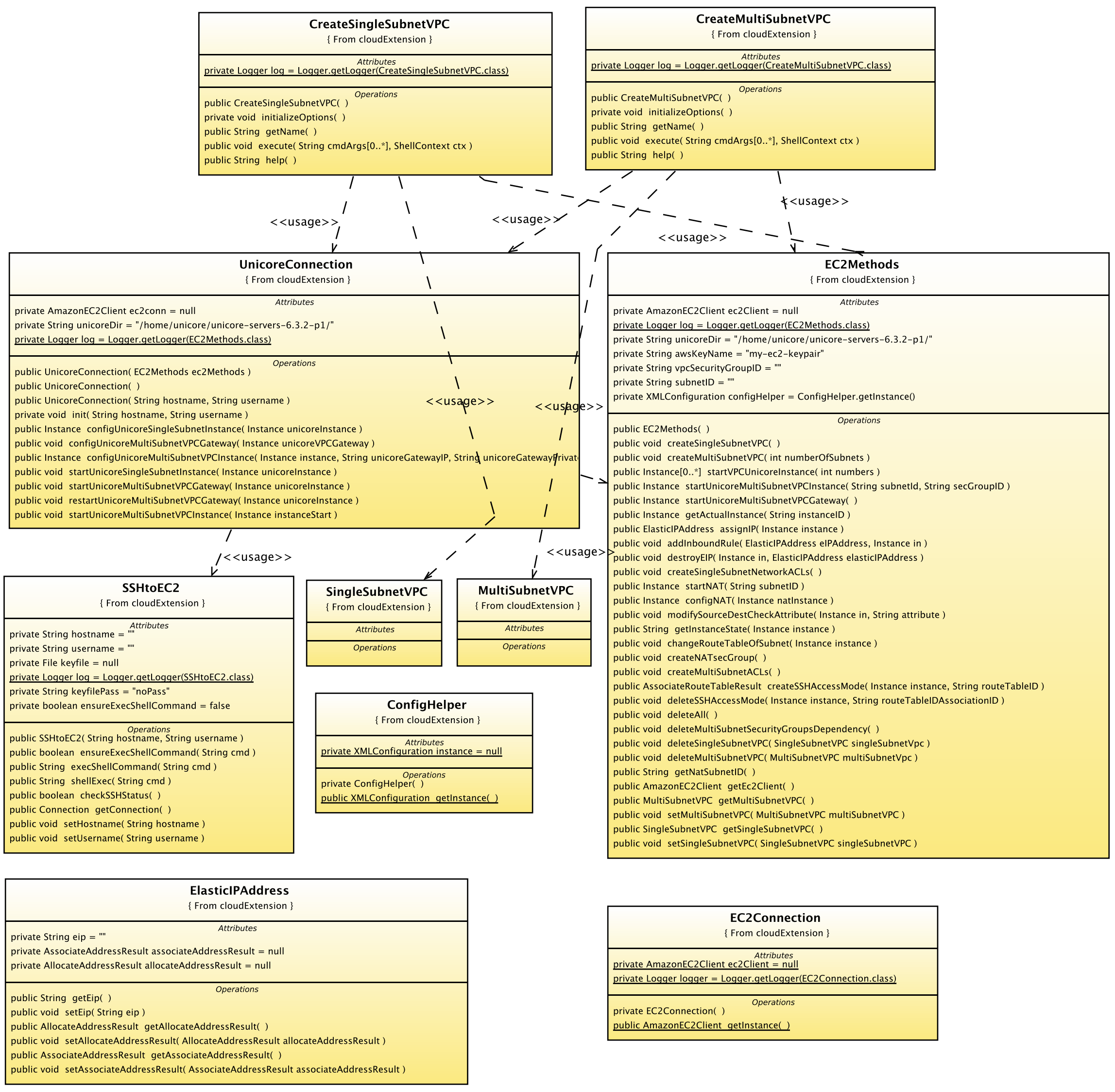
CreateSingleSubnetVPC
{ From cloudExtension }
Attributes
private Logger log = Logger.getLogger(CreateSingleSubnetVPC.class)
Operations
public CreateSingleSubnetVPC( )
private void initializeOptions( )
public String getName( )
public void execute(String cmdArgs[0..*], ShellContext ctx)
public String help( )
CreateMultiSubnetVPC
{ From cloudExtension }
Attributes
private Logger log = Logger.getLogger(CreateMultiSubnetVPC.class)
Operations
public CreateMultiSubnetVPC( )
private void initializeOptions( )
public String getName( )
public void execute(String cmdArgs[0..*], ShellContext ctx)
public String help( )
<<usage>>
<<usage>>
<<usage>>
<<usage>>
UnicoreConnection
{ From cloudExtension }
Attributes
private AmazonEC2Client ec2conn = null
private String unicoreDir = "/home/unicore/unicore-servers-6.3.2-p1/"
private Logger log = Logger.getLogger(EC2Methods.class)
Operations
public UnicoreConnection(EC2Methods ec2Methods)
public UnicoreConnection( )
public UnicoreConnection(String hostname, String username)
private void init(String hostname, String username)
public Instance configUnicoreSingleSubnetInstance(Instance unicoreInstance)
public void configUnicoreMultiSubnetVPCGateway(Instance unicoreVPCGateway)
public Instance configUnicoreMultiSubnetVPCInstance(Instance instance, String unicoreGatewayIP, String unicoreGatewayPrivat
public void startUnicoreSingleSubnetInstance(Instance unicoreInstance)
public void startUnicoreMultiSubnetVPCGateway(Instance unicoreInstance)
public void restartUnicoreMultiSubnetVPCGateway(Instance unicoreInstance)
public void startUnicoreMultiSubnetVPCInstance(Instance instanceStart)
<<usage>>
<<usage>>
EC2Methods
{ From cloudExtension }
Attributes
private AmazonEC2Client ec2Client = null
private Logger log = Logger.getLogger(EC2Methods.class)
private String unicoreDir = "/home/unicore/unicore-servers-6.3.2-p1/"
private String awsKeyName = "my-ec2-keypair"
private String vpcSecurityGroupID = ""
private String subnetID = ""
private XMLConfiguration configHelper = ConfigHelper.getInstance()
Operations
public EC2Methods( )
public void createSingleSubnetVPC( )
public void createMultiSubnetVPC(int numberOfSubnets)
public Instance[0..*] startVPCUnicoreInstance(int numbers)
public Instance startUnicoreMultiSubnetVPCInstance(String subnetId, String secGroupID)
public Instance startUnicoreMultiSubnetVPCGateway( )
public Instance getActualInstance(String instanceID)
public ElasticIPAddress assignIP(Instance instance)
public void addInboundRule(ElasticIPAddress eIPAddress, Instance in)
public void destroyEIP(Instance in, ElasticIPAddress elasticIPAddress)
public void createSingleSubnetNetworkACLs( )
public Instance startNAT(String subnetID)
public Instance configNAT(Instance natInstance)
public void modifySourceDestCheckAttribute(Instance in, String attribute)
public String getInstanceState(Instance instance)
public void changeRouteTableOfSubnet(Instance instance)
public void createNATsecGroup( )
public void createMultiSubnetACLs( )
public AssociateRouteTableResult createSSHAccessMode(Instance instance, String routeTableID)
public void deleteSSHAccessMode(Instance instance, String routeTableIDAssociationID)
public void deleteAll( )
public void deleteMultiSubnetSecurityGroupsDependency( )
public void deleteSingleSubnetVPC(SingleSubnetVPC singleSubnetVpc)
public void deleteMultiSubnetVPC(MultiSubnetVPC multiSubnetVpc)
public String getNatSubnetID( )
public AmazonEC2Client getEc2Client( )
public MultiSubnetVPC getMultiSubnetVPC( )
public void setMultiSubnetVPC(MultiSubnetVPC multiSubnetVPC)
public SingleSubnetVPC getSingleSubnetVPC( )
public void setSingleSubnetVPC(SingleSubnetVPC singleSubnetVPC)
<<usage>>
<<usage>>
SSHtoEC2
{ From cloudExtension }
Attributes
private String hostname = ""
private String username = ""
private File keyfile = null
private Logger log = Logger.getLogger(SSHtoEC2.class)
private String keyfilePass = "noPass"
private boolean ensureExecShellCommand = false
Operations
public SSHtoEC2(String hostname, String username)
public boolean ensureExecShellCommand(String cmd)
public String execShellCommand(String cmd)
public String shellExec(String cmd)
public boolean checkSSHStatus( )
public Connection getConnection( )
public void setHostname(String hostname)
public void setUsername(String username)
SingleSubnetVPC
{ From cloudExtension }
Attributes
Operations
MultiSubnetVPC
{ From cloudExtension }
Attributes
Operations
ConfigHelper
{ From cloudExtension }
Attributes
private XMLConfiguration instance = null
Operations
private ConfigHelper( )
public XMLConfiguration getInstance( )
ElasticIPAddress
{ From cloudExtension }
Attributes
private String eip = ""
private AssociateAddressResult associateAddressResult = null
private AllocateAddressResult allocateAddressResult = null
Operations
public String getEip( )
public void setEip(String eip)
public AllocateAddressResult getAllocateAddressResult( )
public void setAllocateAddressResult(AllocateAddressResult allocateAddressResult)
public AssociateAddressResult getAssociateAddressResult( )
public void setAssociateAddressResult(AssociateAddressResult associateAddressResult)
EC2Connection
{ From cloudExtension }
Attributes
private AmazonEC2Client ec2Client = null
private Logger logger = Logger.getLogger(EC2Connection.class)
Operations
private EC2Connection( )
public AmazonEC2Client getInstance( )

Figure 4.5: Cloud Extension Class Diagram

`EC2Connection` load configuration files, save EIP address information for instances, and provide a connection to the AWS EC2 service.
The base for our implemented HiLA shell commands is the general implementation of HiLA shell commands. Every command, for example cp for copy, mkdir, monitor, or submit inherits from the class `ShellCommandBase` and implement the interface `ShellCommand`. This approach standardizes all commands and ensures that all commands contain the same methods. The central inherited method for all commands is the execute method. The execute method is the central point for every command that parses the input options and runs the specific program logic.
However, we have developed two new commands, the CreateSingleSubnetVPC command and the CreateMultiSubnetVPC command, both according classes are shown in Figure 4.5. These commands automatically configure and start the AWS and UNICORE6 resources in the cloud and include them into the local grid. Therefore, these commands start the Single Subnet VPC scenario and the Multi Subnet VPC scenario from Section 4.3.
The **CreateSingleSubnetVPC** command starts the Single Subnet VPC environment. Hereby, the option `-number [1-5]` is mandatory to start the environment, which defines the number of instances that are started in the Single Subnet VPC environment. Furthermore, the option `-delete` is possible. The `-delete` option terminates all running resources in the cloud and excludes them from the local grid. Therefore, the Single Subnet VPC environment is deleted.
In addition, the command **CreateMultiSubnetVPC** starts a Multi Subnet VPC environment. This command includes two options. The `-subnets` option specifies the number of subnets that are started and is mandatory. The -delete option behaves similar to the Single Subnet `-delete` option.

4.5.1 Automatic AWS Cloud Management

The class `EC2Methods` contains all methods to configure, start and stop the cloud environments. To access the AWS web interface these methods use the AWS SDK for java in version 1.2.6 [16]. Furthermore, the class `EC2Methods` needs helper classes and access credentials to make requests to the AWS SOAP API. Helper classes are the `EC2Connection` class that provides the connection to the AWS services. And the `ConfigHelper` class that provides the path to the AWS credentials and further information for the cloud environment configuration.
However, the central methods of the class `EC2Methods` are the method `createSingleSubnetVPC()` and the method `createMultiSubnetVPC()` that both start the corresponding cloud resources for the Single Subnet VPC scenario and the Multi Subnet VPC scenario. In addition, the relations of these classes are depricted in a class diagram in Figure 4.5.

4.5.1.1 Single Subnet VPC

The method `createSingleSubnetVPC()` in the class `EC2Methods` starts and configures the cloud environment for the Single Subnet VPC scenario. Figure 4.2 depicts the cloud environment for this design.
The Amazon VPC is the base for all other components that reside in the cloud computing environment. After the VPC start-up, the network configuration of the VPC is unchangeable. To reconfigure the network properties of the VPC, the VPC has to be deleted, changed and restarted. In the method `createSingleSubnetVPC()` we create a VPC with a 10.0.0.0/16 Classless Inter-Domain Routing (CIDR) block [21]. This CIDR configuration creates 65,536 possible IP addresses in the VPC, it is the minimal allowed network mask with about 28 bit. In our Single Subnet VPC scenario, we partition the 65,536 possible IP addresses to one single subnet with 65,536 possible IP addresses. Therefore, the single subnet in the VPC contains 65,536 possible IP addresses with a 10.0.x.x prefix. But not all IP addresses are available for simultaneous usage as running instances. AWS provides only 16,500 simultaneously running EC2 instances in this subnet. Furthermore, AWS reserves the first four and the last IP address of a VPC subnet. But all other 10.0.x.x IP addresses are reachable inside the VPC and separated from access outside the VPC.
However, to enable communication to the outside of the VPC subnet the method `createSingleSubnetVPC()` starts and configures the Internet gateway. Amazon provides the Internet gateway as a pre-configured component. The Internet gateway enables communication to the Internet, to EC2 instances outside of the VPC, and especially important for us the communication to the local grid. But only instances with associated EIP addresses are able to communicate over the Internet gateway. Therefore, a maximum of five instances can send and receive traffic out of the VPC in this scenario. In addition, outgoing traffic over EIP addresses always apply charges. The minimum cost are Regional Data Transfer Rates [3].
To make the Internet gateway reachable for the VPC EC2 instances, Amazon provides a implied router. The route table is shown in Table 4.1 and terms Route Table A. The first entry routes all packages inside the VPC subnet. This route enables the instances to communicate with each other. The second route routes all packages to the Internet gateway and therefore it enables the instances with associated EIP address to communication outside the VPC.

Destination	Target
10.0.0.0/16	local
0.0.0.0/0	Internet Gateway

Table 4.1: Route table A

The maximal five instances with associated EIP addresses include the UNICORE6 server, but at least one of the instances contain a gateway component in addition. The class `UnicoreConnection` that we describe in Subsection 4.5.2 configures and starts these UNICORE6 components. The base for these instances is the Linux Ubuntu 10.04 LTS release that the Ubuntu community provides as AMI [59]. Based on this AMI, we created the AMI with the id ami-0c22d965 that contains the needed UNICORE6 components.
However, in the next paragraph, we describe the Multi Subnet VPC scenario that overcomes the limitations of maximal five available UNICORE6 servers.

4.5.1.2 Multi Subnet VPC

The method `createMultiSubnetVPC()` from the class `EC2Methods` configures and starts the cloud environment for the Multi Subnet VPC implementation. The Figure 4.3 displays the cloud environment for this implementation. Similar to the single subnet implementation there exists a VPC in the cloud environment. The configuration of the VPC is similar to the single subnet VPC configuration. The difference is the configuration of the subnets in the VPC, because the VPC is segmented in more than one subnet. The VPC subnet CIDR blocks are limited by the configuration of the VPC netmask configuration which means that the subnets can not contain more IP addresses than the VPC. Furthermore, the subnet CIDR address blocks can not overlap.
However, AWS restricts the number of subnets inside a VPC up to 20. We exhaust this limit of subnets while creating subnets with a 21 bit netmask. That means each subnet contains 2046 IP addresses, with the restriction that AWS reserves the first four and the last IP address from every subnet.
Similar to the previous implementation is the router in the VPC. In addition, the route table in the public subnet is the same route table as in the Single Subnet VPC implementation, see Route Table A 4.1.
Different is the route table B shown in Table 4.2 in the private subnets that differs from

Destination	Target
10.0.0.0/16	local
0.0.0.0/0	NAT Instance

Table 4.2: Route table B

route table A. In the Multi Subnet scenario the route table B is the main route table. Every subnet that is not associated with the route table A uses the route table B. The first route in the route table B enables all VPC instances with subnet specific IP addresses, for example 10.0.8.23, to communicate with each other. The second route in route table B sends all traffic to a NAT instance in the public subnet. The NAT instance enables all instances in the VPC to communicate with the Internet and therefore the NAT enables the communication

with the local grid.
The configuration of the NAT instance is different to the other instances. Therefore an EIP address has to be associated to the NAT instance and the source/destination checking attribute has to be disabled. The disabled source/destination checking attribute allows the instance to send and receive date from other instances. If the source/destination checking is enabled, instances can only send and receive data that is initiated by themselves.
To start the Multi Subnet VPC implementation, we created the command `createMultiSubnetVPC` in the HiLA shell. This command has two possible options. The option `-subnets` specifies the number of subnets and UNICORE6 servers that will start in the cloud computing environment. The option `-delete` stops and deletes all resources in the cloud computing environment.
The difference between the Single Subnet VPC implementation and the Multi Subnet VPC implementation is the higher amount of UNICORE6 servers. Besides, our arrangement of subnets and UNICORE6 servers is not a definite arrangement. We start a single UNICORE6 server in every subnet, but it is although possible to start more than one UNICORE6 server in a VPC subnet. As a maximum, all instances in the VPC can be UNICORE6 servers. In contrast, the minimal number of subnets in this implementation should be two, because one subnet with route table A and one subnet with route table B must be provided. In the next subsection we describe the arrangement of UNICORE resources in detail.

4.5.2 Automatic Deployment and Configuration of UNICORE Grid Resources on AWS Instances

After the start-up of the cloud computing resources that we explained in Subsection 4.5.1, the Cloud Extension automatically starts the UNICORE resources on the cloud instances. The class `UnicoreConnection` in the Cloud Extension, shown in Figure 4.5, contains all methods to configure and start the UNICORE components in the cloud environment.
To configure the UNICORE6 components in the cloud, we connect it over the Secure Shell 2 (SSH-2) protocol to the instances [55]. We included the open source library ganymed-SSH2 for Java [28] in the Cloud Extension. This library implements the SSH-2 protocol in pure Java. Therefore, the ganymed-SSH2 library enables encrypted connections to the cloud instances SSH-2 servers. We configured the user account `unicore` on every instance and use this user account to receive these SSH-2 requests.
The library supports SSH-2 sessions with remote command execution and shell access. This enables the methods from the class `UnicoreConnection` to edit the UNICORE6 configuration files directly on the instances. Furthermore, the methods can start and stop the UNICORE6 components by running the component according start-up shell script. These editing and start-up happens under the user account `unicore`.

Similar to the implementation in the cloud environment that we described in Subsection 4.5.1, different methods for the Single Subnet und Multi Subnet implementation exist in the class `UnicoreConnection`. Because of both implementations, we provide a detailed description for the both implementations in the Subsections 4.5.2.1 and 4.5.2.2.
But both implementations have some similar characteristics. All UNICORE resources register at the registry in the local grid. Therefore, both implementations do not need the registry component. Furthermore, the UNICORE TSI component that enables the usage of cluster resources is optional. We described clusters in Section 2.1, and the usage of clusters in the Multi Subnet VPC and Single Subnet VPC implementations in the Subsection 4.5.3.

4.5.2.1 Single Subnet VPC

The methods `configUnicoreSingleSubnetInstance(Instance unicoreInstance)` and `startUnicoreSingleSubnetInstance(Instance unicoreInstance)` in the class `UnicoreConnection` configure and start the UNICORE components on the passed unicoreInstance variable, shown in Figure 4.5. Both methods connect with the user account `unicore` to the EIP address of the passed instance. The UNICORE6 components that get configured and started are the gateway, unicorex, and the xuudb component, described in Subsection 2.2.3.
However, the concrete configuration of UNICORE6 components is as follows. The name of the UNICORE6 server site and the Group component Identification (GcID) is set to AWS-Grid-"InstanceID ". The "InstanceId" is the ID of the AMI and is a unique identifier. The address of the external registry is set to the local grid gateway, behind which the external registry resides that all UNICORE servers use for registration. Furthermore, the publication IP address of the unicorex server component is changed from the private IP address to the EIP address. Because the unicorex server publishes this address to the external registry at the local grid, only the EIP address is available from the Internet. Afterwards the publication IP and local DNS name of the VPC subnet have to be set in the `/etc/hosts` file. Because UNICORE6 uses the hosts file for DNS lookup, it crashes if the correct entry does not exists. The last configuration step we do, is to add the site name AWS-Grid-"InstanceID" to the xuudb.
After the configuration of the maximum five UNICORE6 servers, these are ready to start. Therefore, the program logic in the command `createSingleSubnetVPC()` calls the method `startUnicoreSingleSubnetInstance(Instance unicoreInstance)` that start-up the configured UNICORE6 servers.

4.5.2.2 Multi Subnet VPC

For the MultiSubnet VPC implementation we added methods in the `UnicoreConnection` class, shown in Figure 4.5. Two configuration methods reside in this class: one for the

UNICORE gateway in the public subnet and the other for UNICORE6 servers in the private subnets. In addition to the configuration methods, we have two start-up methods to start the UNICORE gateway and the UNICORE servers. These methods establish a SSH-2 connection to configure and start the UNICORE processes on the UNICORE instances.
However, the UNICORE gateway in the public subnet consists of the UNICORE gateway component, the UNICORE/X component, and the xuudb user database. In the private subnets, we start UNICORE6 servers that contain only the UNICORE/X component.
The configuration of the UNICORE gateway component is similar to the configuration of the UNICORE6 servers in the Single Subnet VPC implementation described in Subsection 4.5.2.1. Only the name of the gateway site is different, we call the gateway site DEMO-SITE-VPC-GW.
Furthermore, the first configuration steps of the private subnet UNICORE6 servers are similar to the configuration of the Single Subnet UNICORE6 servers. But the Multi Subnet VPC private subnets UNICORE6 servers require one more step. For these servers, we set the private address of the UNICORE6 gateway that reside in the public subnet as gateway host address. Furthermore, the UNICORE6 servers in the private subnets use the xuudb user database from the UNICORE6 gateway instance in the public subnet. Because of that we set the private IP address of the UNICORE6 gateway as xuudb user database server address for the private subnets UNICORE6 servers. Finally, we add the combination of site name and private IP address of the private subnets UNICORE6 servers to the `connections.properties` list at the UNICORE gateway. This list matches site names with their corresponding private subnet IP addresses. Therefore, this list enables the availability of the UNICORE6 servers in the private subnets. The last step, is to add the site names to the xuudb user database at the UNICORE gateway in the public subnet.
The problem during the configuration and start-up process is the accessibility of the private subnets instances. All instances in the private subnets are not accessible from outside of the VPC. Therefore, we have two possibilities to connect to the private subnet machines. First, we login over the SSH-2 protocol to a public subnet instance and from there we can connect to all private subnet machines. Second, during the configuration process we change a private subnet to a public subnet configure to all resources. And after the complete configuration, we change the subnet back to a private subnet.
We implemented the second solution. Therefore, we associate an EIP address to the instance in the temporary public subnet, before we configure and start-up the UNICORE components on that instance. The change of the subnet and association of the EIP address is implemented in the program logic from the command `createMultiSubnetVPC` and uses methods from the class `EC2Methods`.

4.5.3 Deployment and Configuration of PBS Cluster Resources on AWS Instances

In the Cloud Extension we did not implement methods that automatically start-up cluster resources in the cloud computing environment, because to implement a class as the `UnicoreConnection` that starts automatically cluster resources inside the VPC is out of scope. In both implementations we can use the leftover from the 16,500 possible instance inside the VPC as cluster resources. For test purpose we started the Single Subnet VPC environment and the Multi Subnet VPC environment and configured manually an cluster inside these environments.
As cluster software we used the PBS that we described in the Subsection 2.1.2. Two general steps to include the cluster resources into the cloud grids are necessary. First, the TSI component on the UNICORE servers must be configured and installed. Second, the PBS system must run and be connected to the UNICORE TSI components. For the configuration and start-up of the TSI component, we refer to the UNICORE documentation [63]. For the configuration of the PBS we refer to several PBS user guides and documentations, [46] and [76].
The configuration of the cluster resources in the cloud subnets is similar to subnets that do not reside in a cloud environment. Therefore, the user should be able to configure the cluster resources with the standard PBS user guides. Furthermore, the configuration process in both implementations is almost the same. Only the possible arrangement of the cluster resources differs.

4.5.3.1 Single Subnet VPC

In the Single Subnet VPC implementation, cluster resources reside in the public subnet together with the UNICORE servers. We configured pbs_mom daemons to IP blocks and mapped these groups to one of the UNICORE servers. For example, we configured pbs_mom daemons in the subnet block 10.0.1.x and mapped these to the UNICORE server one. The pbs_mom daemons in the subnet block 10.0.2.x we associated to the UNICORE server two. This fragmentation promises a structure that makes the association of cluster resources transparent.

4.5.3.2 Multi Subnet VPC

In the Multi Subnet VPC implementation we configured the cluster resources reside in the private subnets. Therefore, the arrangement of the subnets already provides a structure that fragments the cluster resources. The subnets contain 2046 possible IP addresses, where 2040 IP addresses are available for PBS daemons.

4.5.4 Security Management

In the development, we located three interesting issues for security concerns. First, the security of the local grid, second, the security of the connection between the local grid and the AWS, and third, the security of the AWS.
The first point that regards a secure configuration of the local grid is not a great security concern. The local grid resides in a productive and running local network and we did not notice any security issues. Besides that, a firewall protects this local network against threads from the Internet. To reach the local grid behind the firewall, the UNICORE gateway needs only one open port in the firewall. We opened port 22223 and configured the local grid gateway for this port.
The second point is the security of the communication between the UNICORE6 gateway components. This security concern is only a problem of secure configuration. Because the UNICORE installation across multiple physical locations that are behind firewalls is well established and documented from the UNICORE community and these reasons promise the possibility to configure a secure communication between UNICORE6 gateways. Furthermore, from an abstract point of view, the AWS cloud is just another physical location. We do not discuss the security feature of the UNICORE components and mechanism any further and refer to the UNICORE6 documentation [81].
We analyse the third point with the security concerns of the cloud environment in detail. For example, possibilities to create policies in the VPC or in the subnets that only the UNICORE6 gateways are accessible, similar to the local grid security configuration. Furthermore, we analyse the secure connection of the AWS API to the cloud environment and what kind of security concepts exist in the AWS. To analyse these security issues, we describe the security concepts and techniques of the AWS VPC and introduce secure possibilities to connect to the VPC instances. Furthermore, we explain the security mechanism of the AWS API.

The AWS VPC environment provides two possibilities to ensure security for VPC instances. Security Groups act on instance level and provide a firewall for single instances. In addition, network Access Control List (ACL) act on subnet level and provide security policies for subnets. It is possible to secure all instances only with security groups, but the usage of ACL can provide some advantages, as described in Table 4.3. To provide an overview of the different features from ACL and security groups, we provide the Table 4.3 where we compare security groups and network ACLs.
Furthermore, the security of the AWS SOAP API request is ensured over X.509 certificates in a public key infrastructure. Therefore, the private key that accords to the public X.509 certificate encrypts the AWS API request. With the public X.509 that is included in every API request, AWS confirms that the request sender is correct.

Security Groups	Network ACLs
Security groups act on the instance level	ACLs act at the subnet level
Only allow rules for ingress and egress traffic	Allow rules and deny rules for ingress and egress traffic
Stateful: They allow return traffic always	Stateless: For return traffic must exist network ACLs
To allow traffic, all rules must allow the traffic	The first match in a definite order determines the traffic.
In the instance startup request, the security group must be added manually, this is error-prone.	ACLs can be automatically assigned to instances.

Table 4.3: Compare security groups and network ACLs

However, to connect with the ganymed-ssh library for Java or another Secure Shell (SSH) client to the AWS EC2 instances or launch an EC2 instance, Amazon provides an extra key pair. This key pair contains of a key pair name, a private key file, and a public key file. In every launch request, the key pair name defines the key pair that is used for this particular instance. The private key pair should only reside on the system that handles the connection or launches requests. Amazon stores the public key and copies it during the start-up to the instances that are started with the particular key pair name. Therefore, only the person with the private key can access the instances [18].
In the next two subsections we describe the usage of the AWS security features in our implementations in detail. In addition, we mention that all usage of the security features from the AWS environment is implemented in the class `EC2Methods` in the Cloud Extension. Therefore the Cloud Extension starts and stops the security configurations automatically.

4.5.4.1 Single Subnet VPC

In the Single Subnet VPC implementation, the Cloud Extension creates security groups and network ACL to secure the instances. Every UNICORE6 server becomes a member of the security group "bigVPC". This security group allows inbound traffic for SSH access on port 22 and grid access on port 8080. Furthermore, a loop inbound permission that allows inbound connections from the particular instance to itself is created. This loop permission enables the UNICORE6 gateway component, the UNICORE/X component and the xuudb user database on the instance to be reachable for particular instances. The permissions for the security Group "bigVPC" are displayed in the Table 4.4. In the security group "bigVPC" only exists one outbound rule, displayed in the Table 4.5. Because security groups are state-full, no other outbound rules are needed. Only the 8080 outbound

rule is obsolete, because the UNICORE/X components on the UNICORE servers need to initiate traffic to the local grid without inbound traffic. For example, for the registration at the local grid registry.
However, the configuration of these in- and outbound rules is configured for our particular IP address of the local grid gateway. We configured the source for inbound SSH and gateway access to the fixed IP address 134.76.81.106, shown in Table 4.4. These fixed IP addresses are changeable to any other possible configuration. Setting fixed inbound IP address promises the security advance that only the allowed IP is permitted for connection. For the outbound rule of the gateway in Table 4.5 we configured a fixed IP address, too.

port	usage	source
22	ssh	134.76.81.106/32
8080	unicore gateway	134.76.81.106/32
7777	Unicore/X	private IP
9128	Unicore/X jmxPort	private IP
34463	xuudb	private IP

Table 4.4: Security Groups Inbound rules

port	usage	destination
8080	gateway	134.76.81.106/32

Table 4.5: Security Groups Outbound rules

Additional to the security groups, the Cloud Extension creates network ACL for the subnets in the VPC. This network ACL configuration secures the complete subnet in the VPC against access from outside. In the complete VPC subnet, we only allow inbound traffic over the port 8080 for the UNICORE gateway server and inbound traffic over the port 22 that is opened for SSH connections. Table 4.6 shows the inbound rules for the network ACL. In contrast to the inbound traffic, we do not restrict the outbound traffic. Furthermore, for inbound rules we do not configure a fixed source IP address or range as in the security groups. But similar to security groups, the concrete definition of source IP addresses is possible. We decided not to configure them, because we demonstrated and tested the possibility in the security group configuration.
In addition, the configuration of the network ACLs is optional in contrast to the obsolete security groups. The network ACLs have to open two ports to ensure that the UNICORE components can work probably. Through these both open ports, it is possible to access the resources in the VPC subnet. Because of this possible access the security groups are necessary, and secure these open ports on instance level. Based on this discussion, the network ACLs become really interesting if the VPC subnet contain more resources, for

example a computer cluster.

port	source	type
22	0.0.0.0/0	ALLOW
8080	0.0.0.0/0	ALLOW
ALL	0.0.0.0/0	DENY

Table 4.6: Network ACLs Inbound rules

4.5.4.2 Multi Subnet VPC

In the Multi Subnet VPC environment the Cloud Extension creates three security groups and two network ACLs to ensure the security of the resources. First we provide a list of security groups:

multiSubnetVPC security group This security group is for the UNICORE6 gateway in the public subnet.

privateSubnetUnicoreServers security group This security group is for the UNICORE6 servers in the private subnets.

NATsecGroup This security group is for the NAT instance in the public subnet.

port	usage	source
22	ssh	134.76.81.106/32
8080	unicore gateway	134.76.81.106/32
ALL	subnet loop	privateSubnetUnicoreServers sec. group

Table 4.7: multiSubnetSecurity group Inbound rules

The Table 4.7 shows the inbound rules for the multiSubnetSecurity group. The UNICORE gateway instance resides in this security group. Similar to the Single Subnet VPC implementation, this security group has inbound rules for the UNICORE gateway and SSH access. In addition, an inbound rule exists for all instances that reside in the privateSubnetUnicoreServers security group. The inbound rules for the privateSubnetUnicoreServers are depicted in Table 4.8. The loop rule allows all instances to connect to themselves which is similar to the Single Subnet VPC implementation. The public subnet rule allows the UNICORE gateway to connect to the UNICORE servers in the private subnets. Because the UNICORE servers in the private subnets reside in the privateSubnetUnicoreServers security group. Outbound restrictions do not exist in the privateSubnetUnicoreServers security group. The NATsecGroup security group has only one inbound rule that allows the

port	usage	source
ALL	loop	privateSubnetUnicoreServers security group
ALL	public subnet	multiSubnetSecurity security group

Table 4.8: privateSubnetUnicoreServers security group Inbound rules

connections from the instances in the privateSubnetUnicoreServers security group. Therefore, the NATsecGroup gives the instances in the private subnet the permission to access the NAT instance in the public subnet. Similar to the other two security groups, the NATsecGroup does not have any outbound restrictions.
In the Multi Subnet implementation, the Cloud Extension creates network ACLs, too. The first network ACL is for the private subnets. This network ACL is the default network ACL that is created automatically by the AWS during the VPC start-up. In this default network ACL we have no restrictions, all traffic is allowed. This network ACL can be configured based on the possible cluster systems that can reside in the private subnets. Furthermore, the private subnets are only accessible over the NAT instance which provides a simple network protection. For the public subnet the Cloud Extension creates a more restrictive network ACL. In this network ACL inbound traffic is only possible for SSH access and UNICORE access, similar to the network ACL in the Single Subnet implementation. In addition, in this implementation the Cloud Extension creates access for the private subnet instances. For outbound traffic we allow all traffic that comes out of the VPC.

4.6 Validation

In this section, we provide a validation of our implementations. To validate our implementation, we define test purposes that describe the estimated behaviour and functionality of our implementations. Furthermore, we monitor the behaviour of our implementation in two isolated parts. The first part starts, configures, and includes the cloud computing resources in the local grid. Therefore we name it infrastructure validation for the Single or Multi Subnet VPC implementation which we described in the Subsection 4.6.1. The second part executes jobs at the grid resources in the cloud. Therefore, we use the Single or Multi Subnet VPC as desired and we name this validation job execution validation which is described in Subsection 4.6.2.
We execute all validations manually, for example, by running the commands in the HiLA shell, check the started cloud resources over the AWS web interface or verify the output files manually. The basis for this validation approach is further described in [80].

4.6.1 Infrastructure Validation

In this subsection, we validate for both implementations that they boot-up the expected cloud infrastructure and register the cloud resources into the local grid. In a nutshell, the test purpose for both architectures is: start and configure a UNICORE 6 infrastructure in the AWS cloud computing environment and register the grid resources in the cloud in the local grid.
The infrastructure validation description for the Single Subnet VPC is shown in the Table 4.9. The validation description for the Multi Subnet VPC is nearly similar to this, therefore we did not show a table for the Multi Subnet VPC validation. For both implementations we use the HiLA shell with the Cloud Extension as test application. With the AWS Service Health Dashboard [19] and the AWS Management Console we check the pre-test condition that all cloud resources are available. Furthermore, we start the HiLA shell and verify that no error messages occur in the log files during the boot-up process. At last, we check if the local grid is available by verifying the log files and accessing the local grid with a UNICORE6 client. If these pre-conditions are checked, we can start the test sequence.
The test sequence runs over 10 steps. In step one we start the boot-up of the Single Subnet VPC or the Multi Subnet VPC infrastructure through the according commands. The `creatSingleSubnetVPC -number a`, with a from one to five starts the Single Subnet VPC. And the command `createMultiSubnetVPC -subnets b`, with b from 1-20 starts the Multi Subnet VPC. In step two to eight we verify that the cloud resources has started as estimated. To verify the started resources, we use the AWS Management Console and look in the logging files of the HiLA shell for error messages. In nine and ten we login over SSH to the single cloud instances and check that all estimated UNICORE6 processes are running. In addition, we search the UNICORE6 start-up log files for error messages that might accure during the start-up. Furthermore, we check in the local grid if the grid resources from the cloud have registered probably. After this infrastructure validation, the grid infrastructure in the cloud is ready for job execution.

4.6.2 Job Execution Validation

After we validated that our grid infrastructure in the AWS cloud runs and is registered in the local grid, we validate the job execution which shows that it is possible to execute grid jobs on grid resources in the cloud. Therefore, we define test-purpose that checks if a grid job can be submitted and run on AWS cloud computing resources. To validate the job execution, we provide three use cases. In the first use case, we send a grid job over the HiLA shell to the grid. In the second use case, we programmed a Java standalone application that sends jobs to the grid. The third use case uses the UNICORE6 Rich Client to send a job to the grid.

Single Subnet VPC infrastructure Test Description		
Summary	HiLA shell with Cloud Extension starts and configures the UNICORE6 infrastructure in the cloud correctly	
Pre-test conditions	The cloud resources are available HiLA shell with the Cloud Extension runs The local grid is available for registration	
Test Sequence	Steps	Action
	1	User enters the command to start the Single Subnet VPC
	2	Verify that the VPC is correctly created
	3	Verify that the Public Subnet is correctly created
	4	Verify that the Router is correctly created
	5	Verify that the Internet Gateay is correctly created
	6	Verify that the Custom Route Table is correctly created
	7	Verify that the UNICORE 6 instances are started
	8	Verify the EIP addresses
	9	Verify that the UNICORE processes run correctly
	10	Verify that the UNICORE servers are registered in the local registry

Table 4.9: Single Subnet VPC Infrastructure Test description

4.6.2.1 Jobs in the HiLA shell

For the validation with the HiLA shell we provide a test description which is depicted in Table 4.10. The test application is the HiLA shell with the Cloud Extension. To fulfil our pre-test purposes, we use the HiLA shell with the Cloud Extension to automatically start the Single Subnet VPC or Multi Subnet VPC infrastructure. We need these pre-test conditions, because they establish the grid infrastructure in the cloud on which we execute the grid jobs. As additional pre-test condition, we establish the input file `bash.jsdl` for the test command in the home directory of the current test user at the local grid server.

To start the test sequence with step one, we execute the HiLA shell command “submit” that executes jobs in grid environments. The submit command has four possible arguments, -r to run a job right away, -j specify the JSDL document to be submitted, -s select the site to which the job is submitted, and -h display the help text. We use the complete command to run a job on a UNICORE6 server in the AWS cloud as depicted in the Listing 4.1.

Listing 4.1: submit a job in the HiLA shell

```
sites :> submit -r -j bash.jsdl -s unicore6:/sites/AWS-Grid-i-b1b899d0
unicore6:/sites/AWS-Grid-i-b1b899d0/tasks/43465d95-804a-46df-8ef3-f2540a6c4198
sites :> export stdout stdout_43465d95-804a-46df-8ef3-f2540a6c4198
```

Execute jobs on cloud resources		
Summary	The HiLA shell with cloud Extension submits and runs a grid job in the cloud	
Pre-test conditions	The Single/Multi Subnet VPC infrastructure is available HiLA shell with the cloud extension runs The input file from the local grid is available	
Test Sequence	Steps	Action
	1	User enters the command to start submit and run the job
	2	Verify that the job is successfully finished
	3	Verify that the stdout data is copied to the local host
	4	Verify that the output file is correct

Table 4.10: Job Execution validation with the HiLA shell

To ensure that our job executes on a cloud resource, we set the -s option to an UNICORE6 server that resides in the cloud. In our example, we run the job on the server with the site name AWS-Grid-i-b1b899d0. After we have executed the submit command, the HiLA shell prints the task directory for the submitted job on the console (line two Listing 4.1). With the information about the unique id of the task directory, we can validate step two by changing to the tasks directory and check if the command is successfully accomplished. The task directory is the directory of the current job. In step three, we export the output from the job to our local home directory (line 3 Listing 4.1). In step four, we use the output from the job to validate that the job was executed on the estimated cloud resource. In the output file, the IP address of the site AWS-Grid-i-b1b899d0, the actual date, and time of the execution from the bash.jsdl JSDL file should be printed.
The input file defines the content of the output file. The content of the input file is the print date and print hostname command. We defined this input file in the `bash.jsdl` as command input file (line 23 Listing 4.2). Furthermore, the bash.jsdl file defines the information which applications the job runs on the cloud server. In the `bash.jsdl` we define the `/bin/bash` shell-script that runs and executes the commands from its input file. After the `/bin/bash` shell-script has been executed the commands from the input file, the output is passed to the stdout file in the task directory. To view the stdout file, we can access it directly in the task directory on the server. Alternatively, we can export it to our home directory in the local grid on which the HiLA shell is executed. Line three in Listing 4.1 demonstrates the command to export the output file to our home directory.

Listing 4.2: bash.jsdl for job execution validation

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <jsdl:JobDefinition xmlns="http://www.example.org/"
3 xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl"
4 xmlns:jsdl-posix="http://schemas.ggf.org/jsdl/2005/11/jsdl-posix"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <jsdl:JobDescription>
  <jsdl:JobIdentification>
   <jsdl:JobName>HiLA shell validation</jsdl:JobName>
  </jsdl:JobIdentification>
<jsdl:Application>
 <jsdl:ApplicationName>Bash shell</jsdl:ApplicationName>
 <jsdl-posix:POSIXApplication>
  <jsdl-posix:Executable>/bin/bash</jsdl-posix:Executable>
  <jsdl-posix:Environment name="SOURCE">input</jsdl-posix:Environment>
  <jsdl-posix:Environment name="STANDARD_ERROR">stderr</jsdl-posix:Environment>
  <jsdl-posix:Environment name="STANDARD_OUT">stdout</jsdl-posix:Environment>
 </jsdl-posix:POSIXApplication>
</jsdl:Application>
 <jsdl:DataStaging>
  <jsdl:FileName>input</jsdl:FileName>
   <jsdl:Source>
    <jsdl:URI>
unicore6:/sites/DEMO-SITE-GRID2_201108191352/storages/Home/files/input
    </jsdl:URI>
   </jsdl:Source>
 </jsdl:DataStaging>
 </jsdl:JobDescription>
</jsdl:JobDefinition>
```

4.6.2.2 Jobs with the HiLA API

For job execution validation with our implemented standalone application that is based on the HiLA API, we provide a test description in the Table 4.11. In our test application “Grid Application” two Java classes exist. The class “RunJob” contains the logic and the class “Grid” contains helper methods. During the application procedure, the Grid Application first prints all grid sites to the standard output. Then, the application prints the content of a JSDL file that is created in the class Grid with the JSDLBuilder Factory from the UNICORE6 HiLA implementation. The JSDL file is similar to the `bash.jsdl` in Listing 4.2. Afterwards, the Grid Application submits and executes the JSDL job to a random cloud site. Alternatively, it is possible to set the CandidateHosts element in the `bash.jsdl`. In the last step, the application exports the output from the job to the temporary directory on the host on which the application is executed.
However, the pre-test-conditions are the same as in the previous validation with the HiLA shell which means that the Single Subnet VPC or the Multi Subnet VPC infrastructure must be started and contain at least one grid site in the cloud before test execution.
To start the test sequence, we execute the standalone application. Then the application prints the status of the test sequence, steps two and three to the standard output, and a log file. If the output prints `SUCCESSFUL` for both test sequence steps, the output from the grid

job is written to a file that resides in the local temp directory. To verify the output of the grid job, we verify the content of this file which should be similar to the output file in the job execution validation with the HiLA shell.

Execute jobs on cloud resources		
Summary	The Standalone Application with HiLA API submits and runs a grid job in the cloud	
Pre-test conditions	The Single/Multi Subnet VPC infrastructure is available The standalone application is executeable	
Test Sequence	Steps	Action
	1	User runs the standalone application
	2	Verify that the job is successfully finished
	3	Verify that the stdout data is copied to the local host
	4	Verify that the output file is correct

Table 4.11: Job Execution Validation with the HiLA API

4.6.2.3 Jobs with the UNICORE6 Rich Client

At last, we want to mention the possible validation with the UNICORE6 Rich Client. The test description for the rich client is similar to the previous ones. The rich client has capabilities to display the grid resources, to browse through their storages directory, execute grid jobs, and monitor grid jobs.
An improvement in the job execution compared to both our previous solutions is the capability of the rich client to schedule jobs between not fully loaded grid sites. The user runs the job on the grid and the rich client chooses the execution site for the job.
In this subsection we provided a validation approach to validate our Cloud Extension that automatically creates a UNICORE6 grid in the AWS cloud. Furthermore, we validated the possibility to use these cloud resources in the UNICORE6 grid environment.

4.7 Discussion

In this section, we discuss our implemented scenarios. This discussion is based on the general requirements and aspects we analyzed in Section 3.2. However, our implementation can serve a wide range of use cases. The Cloud Extension implementation is usable in shell context, standalone applications, and in the UNICORE6 GUI client. In the shell context and in the standalone application, we can manage the cloud resources directly from the shell or include the Cloud Extension in a standalone application. To use the Cloud Extension in the UNICORE6 GUI client, we have to use the shell client to manage the cloud resources, or we have to include the Cloud Extension in the GUI. In this discussion we give suggestions to classify the Cloud Extension for the users particular use cases.

Management of cloud resources Two different domains there that need to be managed in the AWS cloud computing environment. First, the AWS cloud resources, for example, instances, NATs, subnets, and security groups servers. Second, the UNICORE6 components need to be managed. To manage the AWS cloud resources, we use the AWS Java SDK which is described in Section 4.3. The SDK is the lowest level of abstraction to manage the cloud resources, but it promises to be the most flexible to integrate in our client, because all UNICORE6 components are programmed in Java. In contrast to the SDK, it is possible to manage the cloud resources over a graphical web interface. But with the graphical web interface, it is not possible to start and stop our implementations with an automatic start-up sequence. For testing purposes and during the implementation process we used this web interface. In addition, we used it to monitor our resources.
Furthermore, a plug-in for the Eclipse Integrated Development Environment (IDE) that promises basic monitoring and managing functionality for the cloud resources is available. We use the IDE plug-in during the implementation process similar to the graphical web interface, even if the IDE plug-in does not provide the same amount of managing possibilities. This IDE plug-in can be a basis for an integration of cloud resources management in the UNICORE6 GUI clients, since both rely on the RCP architecture.
However, for the management of the UNICORE6 components an API or web interface does not exist. Therefore, we developed the Cloud Extension to overcome this limitation and take a step towards a management software for the UNICORE6 components.

Interoperability between clouds The focus during the implementation was not to implement a solution that is portable between different cloud vendors. We used the AWS Java SDK that is a particular cloud computing vendor SDK to manage our cloud resources. Because of the focus during the implementation and the particular SDK, our implementation is hard to port to another cloud vendor, especially with only little changes in the implementation. These changes can be necessary on different points, for example the exchange of the AWS Java SDK, the configuration of the UNICORE6 components, and even in the job execution on the grid client side.
If the implementation changes to another cloud computing vendor, all requests against the cloud might have to be changed. Especially, the class EC2Methods makes extensive use from the AWS Java SDK. Furthermore, another cloud environment should provide the same configuration possibilities of subnets, gateways, possible IP addresses, etc. Only with the same cloud configuration, it is possible that no changes in

the configuration methods of the UNICORE6 components are necessary. Problems on job execution can occur, too. It is possible to integrate the candidateHostName attribute in a JSDL file, but if the host name changes, these JSDL files have to be changed.
A possible solution is the use of a cloud software that is specified by the OCCI. Alternatively, a layer that translate from AWS cloud commands to common cloud commands can be created. This layer can be used to integrate a unified interface to the cloud request. An example for such cloud APIs is the Apache libcloud.

Reliability of cloud resources Amazon promises a 99.95% availability for their EC2 resources across the Amazon EC2 Regions [4]. The 99.95% availability promises less then 5 hours unavailability per year. During the writing and implementation of this thesis, disruptions in the AWS cloud occurred and data was lost [56], [35]. These failures do not bother our implementation process, but within a productive running implementation data would be lost.
However, in august AWS offered a credit voucher for that crash for all customers [6]. Furthermore, Amazon states that the crash does not affect the SLA uptime, because only one of two regions were down. These reliability problems make very clear how important it is to have a backup plan for a possible crash of the cloud. In addition, the SLAs are not easy to understand and cloud users are well advised to have a critical look into the SLAs. Caused of the both AWS crashes, a controversal discussion arisesd in the AWS community, if the crashes affect the SLA uptime.
However, an approach to secure data against the patriot act that we mentioned in the Section 3.2 provides the security provider for clouds Porticor Cloudsecurity [49]. Furthermore, AWS updated their VPC features and added the multiple Availability Zone feature to the VPC [10]. With this feature, the crashes mentioned above are partly absorbed.

Cost of cloud resources AWS offers three different purchasing models to rent Amazon EC2 instances. These three are On-Demand Instances, Reserved Instances, and Spot Instances. For all purchasing models no minimum fee exists and in all purchasing models the cost depends on the region where the instances are running. The price differs on the instance type in all purchasing models. Available are instance types with more or less computing power.
However, On-Demand Instances can be rented on an hourly basis. They offer the opportunity to increase or decrease the amount of instances every time. For On-Demand Instances, only the used uptime is purchased. For Reserved Instances, the user pays in advance for an estimated time and therefore reserves an arranged usage time for this instance. The fees for running Reserved Instances are less than the fess

of an On-Demand Instance. The third purchasing model is the Spot Instance Model. To run Spot Instances, the user sets a maximum fee that is acceptable to pay for an particular instance type. In fixed time intervals Amazon sets a Spot Price for all instance types in an Availability Zone. This Spot Price depends on demand and supply. If this Spot Price drops below the maximum fee the users defined, the instance runs [3].
For our implementation we only used On-Demand Instances, because in the development process we could not estimate how much instance time we would need and therefore we could not use Reserved Instances. Furthermore, to wait for a implementation test until the chosen Spot Price is reached, is not a practicable way, too. In a productive environment, the different purchasing models could maintain to economies. If the estimated usage for a job is clear, the Reserved Instance can lead to lower costs. In the Spot Instances purchasing model, a problem can occur with the saving of the grid state in the cloud when the Spot Price forces the instances to shut down. There must be a clear approach, how the running job in the cloud grid can be paused and restored.
In the future, the number of pricing models of cloud resources will increase. With more interoperability between cloud resources, it can be possible that there will be cloud resource stock exchanges at which cloud computing resources can be handled like fuel. Even AWS searches for new employees that will work on concepts to sell cloud computing resources [33] in standardized ways.

Security aspects The security mechanisms offered by AWS, provide the possibility to configure secure cloud computing environments. In Subsection 4.5.4 we created secure implementations that used many possible capabilities of the AWS.
However, risks reside in the detail of configuration and usage. For example, researches from the Center for Advanced Security Research Darmstadt (CASED) found vulnerabilities in the usage of the AWS [11]. The vulnerability lies in the publishing of AMIs. In a nutshell, AWS users make their AMIs public without deleting all private data from the public AMIs. In detail, the CASED scientists found private keys for authentication to the users AWS services, private SSH keys for root access, valid Secure Socket Layer (SSL) certificates with associated private keys, and other private data in over 30% of the inspected AMIs. To check an AMI for security or private critical data, the AMI aiD (AMID) tool was created. This AMID tool scans a AMI system for critical data [12].
Furthermore, AWS offers a wide range of manuals on how to publish the AMIs without security issues [53], [34],[52]. But the rate of 30% AMIs with private data shows that even good documentation makes it not easy to configure a secure cloud environment. In general, a user should be aware that the publishment of AMIs could

cause vulnerabilities of more than the cloud systems. With a private root SSH key, an attacker can access the private local network of a company or any other facility that has been published the AMI.
To develop a secure cloud computing environment, the user should be aware at this issue in every process phase, from the design to the implementation and the monitoring and management. In Section 3.2, we provide some general aspects on cloud computing security. Furthermore, AWS supplies lots of information about the security in their cloud environment, too [17].

Performance Aspects We did not experience performance concerns in our implementation. Also, we did not stress tested our implementation. The available resources and services in the AWS VPC that affect the performance are increasing continuously [51], for example, maximum of available resources, dedicated instances, type of available resources, etc. In addition, AWS published an VPC update on the August 4th 2011 that allows up to five VPC environments for each user account. The possibility of five VPC environments increases the maximum of available instances dramatically. We discuss this in the issue amount of cloud resources. Together with the wide variety of possible instances [2] and tHigh Performance Computing [9] instances AWS a wide range of computing resources can be used.

Requirements of the local Network The requirements for our implementations to the local network are simple. For the UNICORE6 gateway, the port 22223 in our local network firewall must be open. No further requirements are necessary. Except that the local grid must be reachable, but for that the open port is needed.

Amount of cloud resources In the Section 4.3, we discussed the limits of the AWS cloud and provided implementations to overcome these limitations. For example the limits are maximal one VPC per user account, the maximum of 5 EIP, the maximum of 20 subnets in the VPC. But AWS limits more components as discussed above, a complete overview is provided in the AWS VPC documentation [8]. The limits are set by AWS that include, for example, security groups, network ACLs, route tables, and gateways. The limits reside in resource limitations and in configuration limits. If a user changes the implementation, the user must be aware of the resource limitations. Another limitation can occur at the start-up request for On-Demand instances. AWS does not ensure that it is always possible to start On-Demand instances directly. On-Demand Instances are maybe not able to start for "short periods of time "during high demand periods [1].
But during our implementation and validation process, we did not have any problems with lazy starting On-Demand Instances. We also did not have any problems with lack of cloud resources.

5 Conclusion

In this chapter, we conclude the thesis by providing a summary of our work. Furthermore, we discuss our thesis in general and provide an outlook on how the work could be extended.

Summary

We analyzed general requirements and aspects of the integration of cloud computing resources into grid systems and provided different abstract scenarios to integrate cloud computing resources into grid systems. We provided a discussion about the regarding scenarios regarding general requirements and aspects. Furthermore, we chose one abstract scenario for implementation that promises to be a combination of a secure and reliable solution. The scenario can be implemented without overhead in the configuration and the programming requirements.
We developed two implementation models for our chosen abstract scenario that provide possibilities to overcome limitations of the AWS cloud. Both these implementation models have been implemented and we provided a validation with example uses cases. The validation verified that both implementations integrate AWS cloud computing resources into the UNICORE6 grid environment. Furthermore, we provided a discussion about our two implemented scenarios that focuses on the general requirements and aspects.

Discussion

During the implementation of the design different problems occur. For example, we were not able to implement our chosen abstract scenario from Subsection 3.4.2 directly, because we had to overcome limitations of the AWS cloud. In other cloud environments, limitations and restrictions might be different. This difference can cause an increasing implementation effort and even different implementation solutions.
During the writing of this thesis, the AWS API changed twice. API methods got deprecated and over this two API versions some got even erased. In addition, we needed some time to understand the HiLA API because for this API, UNICORE provides just one paper as documentation. Beside that the code from the HiLA API is not documented.

Outlook

Our implementation works and can be the foundation for further development. In the following paragraphs we provide possible extensions for this work and implementation.
If the cloud computing interoperability evolves further, this evolvement promises a possibility to get more independent from a specific cloud vendor. Our implementation is only compatible to the AWScloud. The possibility exists, to integrate a layer in the implementation that is able to communicate with different cloud environments. Therefore, it is possible to plug-in different cloud computing environments easily.
The possibility to exchange the cloud environment arbitrarily, directs to the configuration of the grid environment in different cloud computing environments. To develop concepts and functionalities for the grid systems, on how to configure the grid systems across different cloud computing environments promises more flexible environments. The grid configuration could be implemented into a layer, similar to the cloud computing connection. This approach could make the exchange of grid systems possible. In a nutshell, our Cloud Extension in the HiLA environment could be extended to an abstract framework that can be integrated into different types of grid and clouds.
Furthermore, it could be an approach to change the TSI component from the UNICORE server suite to be able to run on cloud computing resources without the presence of other parts of the UNICORE server suite. In addition, the TSI component must be changed to communicate with the UNICORE grid component. These changes could enable the configuration of cloud instances with the TSI component and a cluster system. Then this instance could be integrated into an existing UNICORE grid environment. The benefit from this approach is to get grid resources in the cloud with a minimum number of UNICORE components.

List of Abbreviations

ACL	Access Control List
AMI	Amazon Machine Image
AMID	AMI aiD
API	Application Programming Interface
AWS	Amazon Web Services
BES	Basic Execution Service
CASED	Center for Advanced Security Research Darmstadt
CIDR	Classless Inter-Domain Routing
CPIP	Cloud Portability and Interoperability Profiles
CSA	Cloud Security Alliance
CSCC	Cloud Standards Customer Council
DMTF	Distributed Management Task Force
DNS	Domain Name Service
EC2	Elastic Compute Cloud
EBS	Elastic Block Store
EIP	Elastic Internet Protocoll
GcID	Group component Identification
GUI	Graphical User Interface
HiLA API	High Level application programming interface
HiLA	High Level

HPC	High Performance Computing
IDB	Incarnation Database
JSDL	Job Submission Description Language
LRM	Local Resource Management
IaaS	Infrastructure as a Service
IEEE	Institute of Electrical and Electronics Engineers
ICWG	Intercloud Working Group
IDE	Integrated Development Environment
IP	Internet Protocol
NAT	Network Address Translation
NIST	National Institute of Standards and Technology
OCCI	Open Cloud Computing Interface Core
OGF	Open Grid Forum
OGSA	Open Grid Service Architecture
OMG	Object Management Group
PaaS	Platform as a Service
PBS	Portable Batch System
RCP	Rich Client Platform
RDS	Relational Database Service
S3	Simple Storage Service
SaaS	Software as a Service
SAML	Security Assertion Markup Language
SDO	Standards Development Organizations
SDK	Software Development Kit

SLA	Service Level Agreement
SOA	Service Orieted Architecture
SOAP	Simple Object Access Protocol
SSH-2	Secure Shell 2
SSH	Secure Shell
SSI	Single System Image
SSL	Secure Socket Layer
TSI	Target System Interface
UAS	UNICORE Atomic Service
UCC	UNICORE Command-Client
UNICORE	Uniform Interface to Computing Reources
UNICORE6	Uniform Interface to Computing Resources 6
URC	UNICORE Rich-Client
USA	United States of America
UVOS	UNICORE Virtual Organization Service
VPC	Virtual Private Cloud
VPN	Virtual Private Network
WS-RF	Web Service - Resource Framework
WSDL	Web Service Description Language
XACML	eXtensible Access Control Markup Language

Bibliography

[1] Amazon EC2 Instance Purchasing Options. http://aws.amazon.com/ec2/purchasing-options/.

[2] Amazon EC2 Instance Types. http://aws.amazon.com/ec2/instance-types/.

[3] Amazon EC2 Pricing. http://aws.amazon.com/ec2/pricing/.

[4] Amazon EC2 Service Level Agreement. http://aws.amazon.com/ec2-sla/.

[5] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[6] Amazon Post-Mortem open questions. https://forums.aws.amazon.com/thread.jspa?threadid=74018&tstart=0.

[7] Amazon Timeline. http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-corporatetimeline.

[8] Amazon Virtual Private Cloud. User Guide. http://docs.amazonwebservices.com/amazonvpc/latest/userguide/.

[9] Amazon Web Services - High Performance Computing (HPC). http://aws.amazon.com/de/ec2/hpc-applications/.

[10] Amazon Web Services. Release Notes. http://aws.amazon.com/releasenotes/amazon

[11] AmazonIA: When elasticity snaps back. http://trust.cased.de/amid.

[12] AMI aid (AMID) - Scanning a system for security or privacy critical data before publishing or when started as Amazon Machine Image (AMI). https://code.google.com/p/amid/.

[13] Apache Libcloud. http://libcloud.apache.org/.

[14] Apple iCloud. http://www.apple.com/de/icloud/.

[15] AWS Products. http://aws.amazon.com/products/.

[16] AWS SDK for Java. http://aws.amazon.com/sdkforjava/.

[17] AWS Security and Compliance Center. http://aws.amazon.com/security/.

[18] AWS Security Credentials. http://docs.amazonwebservices.com/AWSSecurityCredentials/1.0/AboutAWSCredentials.html.

[19] AWS Service Health Dashboard. http://status.aws.amazon.com/.

[20] AWS Simple Monthly Calculator. http://calculator.s3.amazonaws.com/calc5.html.

[21] Classless Inter-Domain Routing (CIDR). http://tools.ietf.org/html/rfc4632.

[22] Cloud Standards Customer Council. http://www.cloud-council.org/.

[23] CSA. Cloud Security Alliance. https://cloudsecurityalliance.org/.

[24] DeltaCloud. An API that abstracts the differences between clouds. http://incubator.apache.org/deltacloud/.

[25] Dropbox. http://www.dropbox.com/.

[26] Eclipse - Rich Client Platform. http://www.eclipse.org/home/categories/rcp.php.

[27] EyeOS.The Cloud Desktop. http://eyeos.org/.

[28] Ganymed SSH-2 for Java. Open Source Library. http://www.cleondris.ch/opensource/ssh2/.

[29] gLite - Lightweight Middleware for Grid Computing. http://glite.cern.ch/.

[30] Globus Toolkit. Open Source grid software toolkit. http://www.globus.org/.

[31] GoGrid. Cloud Cost Calculator. http://www.gogrid.com/cloud-hosting/cloud-hosting-pricing-calculator.php.

[32] Gridsphere Portal Framework. http://www.gridsphere.org/gridsphere/gridsphere.

[33] HIRING: Build the World's Largest Dynamically Priced Market for Computing Resources. http://aws.amazon.com/spot-jobs/.

[34] How to share and use public AMIs in a secure manner. http://aws.amazon.com/articles/0155828273219400.

[35] http://aws.amazon.com/de/message/2329b7/.

[36] IEEE. Guide for Cloud Portability and Interoperability Profiles. http://standards.ieee.org/develop/project/2301.html.

[37] IEEE. Intercloud Working Group. http://standards.ieee.org/develop/wg/icwg-2302_wg.html.

[38] Microsoft admits Patriot Act can access EU-based cloud data. http://www.zdnet.com/blog/igeneration/microsoft-admits-patriot-act-can-access-eu-based-cloud-data/11225.

[39] Microsoft Online Services. Trust Center. http://www.microsoft.com/online/legal/v2/?docid=23.

[40] OGF. Open Cloud Computing Interface Implementations. http://occi-wg.org/community/implementations/.

[41] OGSA 1.5 Spec http://www.gridforum.org/Public_Comment_Docs/Documents/Apr-2006/draft-ggf-ogsa-spec-1.5-008.pdf.

[42] OMG Object Management Group. http://www.omg.org/.

[43] Open Grid Forum. http://www.gridforum.org/.

[44] OpenPBS. http://www.pbsworks.com/?aspxautodetectcookiesupport=1.

[45] PBS Admin Guide http://doesciencegrid.org/public/pbs/pbs.v2.3_admin.pdf.

[46] PBS Architecture http://hpc.sissa.it/pbs/pbs1.html#ss1.3.

[47] PBS Professional. http://www.pbsworks.com/product.aspx?id=1.

[48] Porticor Cloudsecurity. Cloud Privacy, Regulation, and the Patriot Act. `http://www.rackspace.com/cloud/cloud_hosting_products/servers/pricing/`.

[49] Porticor Cloudsecurity. Cloud Privacy, Regulation, and the Patriot Act. http://www.porticor.com/2011/08/cloud-privacy-regulation-and-the-patriot-act-2/.

[50] Rackspace Cloud Hosting. http://www.rackspace.com/.

[51] Release Notes Amazon VPC. http://aws.amazon.com/releasenotes/amazon

[52] Reminder about Safely Sharing and Using Public AMIs. http://aws.amazon.com/security/security-bulletins/reminder-about-safely-sharing-and-using-public-amis/.

[53] Sharing AMIs Safely.
`http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/index.html?AESDG-chapter-sharingamis.html`.

[54] SOAP Specifications. http://www.w3.org/tr/soap/.

[55] SSH-2 Protocol Architecture. http://tools.ietf.org/html/rfc4251.

[56] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/message/65648/.

[57] TORQUE RESOURCE MANAGER. http://www.clusterresources.com/products/torque-resource-manager.php.

[58] Toyota/Microsoft News Conference - Akio Toyoda, Steve Ballmer. http://pressroom.toyota.com/article_display.cfm?article_id=2962.

[59] Ubuntu EC2 Instances. https://help.ubuntu.com/community/ec2startersguide.

[60] Ubuntu One. https://one.ubuntu.com/.

[61] UNICORE History. http://www.unicore.eu/unicore/history.php.

[62] UNICORE Subversion Repository. http://sourceforge.net/scm/?type=svn&group_id=102081.

[63] UNICORE TSI. Manual. http://www.unicore.eu/documentation/manuals/unicore6/files/tsi/tsi-manual.html.

[64] UNICORE. Uniform Interface to Computing Resources. http://www.unicore.eu/.

[65] The Vine Toolkit. http://vinetoolkit.org/.

[66] Web Service Resource Framework (WSRF). Primer, http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-01.pdf.

[67] Unicore 6 - Recent and Future Advancements UNICORE_6-Recent_and_Future.pdf, 2010.

[68] AWS White Paper. Overview of Amazon Web Services.http://aws.amazon.com/whitepapers/., December 2010.

[69] Mark Baker. Cluster computing white paper. *CoRR*, cs.DC/0004014, 2000. informal publication.

[70] R. Buyya. *High Performance Cluster Computing: Architectures and Systems.* Prentice Hall, 1999.

[71] Anirban Chakrabarti. *Grid Computing Security*. Springer, 2007.

[72] Frederic Magoules Choi-Hong Lai, editor. *Introduction to Grid Computing*. CRC Press, 2009.

[73] I. Foster. Open Grid Services Architecture Basic Execution Services. http://www.ogf.org/documents/gfd.108.pdf.

[74] Ian Foster. What is the Grid? A Three Point Checklist. *Argonne National Laboratory & University of Chicago*, 2002.

[75] Ian Foster and Carl Kesselman, editors. *The Grid 2, Second Edition: Blueprint for a New Computing Infrastructure (The Elsevier Series in Grid Computing)*. 2004.

[76] James Patton Jones. Portable Batch System External Reference Specification. Technical report, Veridian Systems Inc., 2000.

[77] James Patton Jones. Portable Batch System User Guide. Technical report, Veridian Systems, Inc., 2001.

[78] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, 2009.

[79] Thomas Rings. Cloud computing presentation. WS2010/11.

[80] Thomas Rings, Jens Grabowski, and Stephan Schulz. A Testing Framework for Assessing Grid and Cloud Infrastructure Interoperability. *International Journal On Advances in Systems and Measurements (SysMea11v4n12) ISSN: 1942-261x, 2011 vol 3 nr 1&2, IARIA*, July 2011.

[81] UNICORE Team. Setting up multiple UNICORE sites with the graphical installer. Technical report, April 2010.

[82] Unicore Team. UNICORE HiLA 2.1. Documentation, August 2010.

[83] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing Cloud Computing Hardware Reliability. Technical report, Microsoft Research.

[84] Chee Shin Yeo and Rajkumar Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Softw., Pract. Exper.*, 36(13):1381–1419, 2006.

All links have been checked on October 14th 2011.