Universidade de Aveiro Departamento de Eletrónica,
Telecomunicações e Informática
2014

Eduardo Miguel Coutinho Gomes de Pinho

Um toolkit web para integração de serviços cloud

A web toolkit for cloud service integration

Universidade de Aveiro
2014

Departamento de Eletrónica,
Telecomunicações e Informática

Eduardo Miguel Coutinho Gomes de Pinho

Um toolkit web para integração de serviços cloud

A web toolkit for cloud service integration

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Carlos Manuel Azevedo Costa, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

agradecimentos /
acknowledgements

Agradecimentos a família, colegas e amigos, pelo constante apoio.

Palavras Chave

computação na *cloud*, interoperabilidade entre *clouds*, serviços web, engenharia de software, aplicações web.

Resumo

A utilização do paradigma de computação na *cloud* está hoje generalizada em diferentes áreas da sociedade. No entanto, a utilização de recursos fornecidos por múltiplos fornecedores de serviços tem um conjunto de problemas associados à normalização e interoperabilidade destes serviços. Os esforços para ultrapassar tal problema têm passado pela criação de especificações abertas e frameworks de integração. Contudo, o desenvolvimento de aplicações web levanta outras questões no que diz respeito ao acesso e gestão de recursos *cloud* por parte da lógica da aplicação executada no lado do cliente. Esta dissertação propõe e desenvolve uma plataforma extensível para a integração de serviços cloud, desenhada para satisfazer os requisitos e padrões comuns das aplicações web. A plataforma inclui políticas de controlo de acesso e mecanismos para a partilha, delegação e replicação de recursos. Finalmente, apresentam-se testes de desempenho da solução implementada, seguindo-se uma análise e discussão dos resultados obtidos.

Keywords cloud computing, cloud interoperability, web services, software engineering, web applications.

Abstract

The latest trends on cloud and multi-cloud computing are well established in our society. However, the lack of interoperability raised a few issues that have been tackled with open standards and integration frameworks. Still, web application development adds a few more issues when accessing and managing cloud resources in the application's logic. This thesis describes an extensible platform architecture for portable cloud service integration, designed to satisfy requirements and usage patterns of web applications. Moreover, it implements access control policies and mechanisms for cloud resource sharing, delegation and replication. Finally, the thesis presents performance tests of the solution, along with an analysis and discussion of the results obtained.

Contents

Contents i

List of Figures v

List of Tables vii

Glossary ix

1 Introduction 1
- 1.1 Overview 1
- 1.2 Proposal 2
- 1.3 Thesis Structure 3

2 State of the Art 5
- 2.1 Web Services 5
- 2.2 Cloud Computing 6
- 2.3 Cloud Services 7
  - 2.3.1 Compute 8
  - 2.3.2 Data Storage 9
  - 2.3.3 Message Queues 10
  - 2.3.4 Notification 10
- 2.4 Cloud Computing Interoperability and Sky Computing 10
  - 2.4.1 Cloud Interoperability Standardization 11
  - 2.4.2 Solutions for Cloud Service Integration 13
- 2.5 Service Delivery Cloud Platform 15
  - 2.5.1 Storage Abstraction 17
  - 2.5.2 Database Abstraction 18
  - 2.5.3 Notification Abstraction 18
- 2.6 mOSAIC Project 18

2.6.1 Architecture 18
2.6.2 Components Description 20
2.6.3 API design 22

3 Platform Design 25
3.1 A cloud service platform oriented to web applications 25
3.2 Platform Requirements 26
3.3 The Architecture 27
3.4 Cloud Controller model 28
3.5 Cloud resource model and manipulation 30
3.6 Agents, Authentication and Access Control Policies 32
3.6.1 Resource Delegation 34
3.7 The API 37
3.8 The Plugin support 37
3.9 The Services 38

4 Platform Development 43
4.1 Pre-Development Considerations 43
4.1.1 mOSAIC Considerations 44
4.2 Data Models and Protocols 46
4.3 Cloud Controller 49
4.3.1 Cloud Controller Web Services 49
4.4 Remote Connector SDK 53
4.4.1 Internal Architecture 54
4.5 Cloud Provider Access Drivers 55
4.6 SDCP Runtime 57
4.7 Resource Type: Simple Database 59
4.7.1 AWS SimpleDB 62
4.7.2 Client Module 62

5 Results and Discussion 63
5.1 Benchmarking 63
5.1.1 Environment Conditions and Sequence of Operations 63
5.1.2 Results 64
5.1.3 Observations 66
5.2 Discussion topics 67
5.2.1 mOSAIC Integration 67
5.2.2 Security 67
5.2.3 Benefits 68
5.2.4 Drawbacks 69

6 Conclusion 71
   6.1 Future Work 72

References 73

List of Figures

2.1 Service Delivery Cloud Platform legacy architecture 16
2.2 Main sub-systems of the mOSAIC architecture 20
2.3 API layers of mOSAIC 24

3.1 SDCP Concept Diagram 28
3.2 SDCP Cloud Controller Model 29
3.3 SDCP Remote Connector model, containing an example of dynamic remote connector access information. 31
3.4 SDCP's cloud resource hierarchy model 32
3.5 SDCP Agent Authentication and Access Control Diagram 33
3.6 Resource delegation in SDCP, client-oriented approach. 36
3.7 Resource delegation in SDCP, server-oriented approach. 36
3.8 Filestore resource model. 39
3.9 Simple database resource model. 40
3.10 Notification service resource model. 41

4.1 Remote connector internal structure 54
4.2 Internal structure of a mOSAIC connector-driver pair implementation 56

5.1 A comparison of performance between the use of SDCP for simple database operations and direct SimpleDB access. 66

List of Tables

4.1 A simple comparison of web server applications and technologies. 44
4.2 JSON document definitions in SDCP. 48
4.3 Main SDCP cloud controller web service. 51
4.4 Administration SDCP cloud controller web service. 51
4.5 Cloud access SDCP cloud controller web service. 51
4.6 Internal SDCP cloud controller web service. 52
4.7 Client Runtime API description. 58
4.8 SDCP Simple Database remote connector web service description. 61

5.1 The benchmarking results containing average and maximum execution times of each operation type, in milliseconds. 65

Glossary

ACL	Access Control List
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AWS	Amazon Web Services
CaaS	Communication as a Service
CAMP	Cloud Application Management for Platforms
CCIF	Cloud Computing Interoperability Forum
CDMI	Cloud Data Management Interface
CFP	Call For Proposals
CORS	Cross Origin Resource Sharing
CSB	Cloud Service Broker
CSV	Comma Separated Values
DaaS	Data-Storage as a Service
DBMS	Database Management System
DICOM	Digital Imaging and Communications in Medicine
FTP	File Transfer Protocol
GAE	Google App Engine
GICTF	Global Inter-Cloud Technology Forum
HTTP	Hyper-Text Transfer Protocol
HTTPS	HTTP Secure
IaaS	Infrastructure as a Service
IDL	Interface Description Language
IP	Internet Protocol
JPA	Java Persistence API
JSON	JavaScript Object Notation
NaaS	Notification as a Service
NIST	National Institute of Standards and Technology
OCCI	Open Cloud Computing Interface
OS	Operating System
PaaS	Platform as a Service
QoS	Quality of Service
REST	Representational State Transfer
RPC	Remote Procedure Call
SaaS	Software as a Service
SDCP	Service Delivery Cloud Platform
SDK	Software Development Kit
SLA	Service-Level Agreement
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TLS	Transport Layer Security
URI	Universal Resource Identifier
URL	Universal Resource Locator
UUID	Universally Unique IDentifier
VM	Virtual Machine
WSDL	Web Service Definition Language
XML	Extended Markup Language
XMPP	Extensible Messaging and Presence Protocol

CHAPTER 1

Introduction

1.1 OVERVIEW

Web applications can deliver a program to the user without a previous installation process on the client machine, while still being able to delegate part of the application's logic (mostly the user interface) to the local machine, regardless of its operating system. The modern HTML5 standard [1], while still under development, has already increased the potential of many web applications with capabilities such as new local storage mechanisms and JavaScript objects. These additional elements significantly reduce the need for third-party dependencies, like for instance, Adobe Flash. Furthermore, from the fact that web applications lift many resource requirements from the local machine, the development of cross-platform applications for mobile devices has become feasible and worthwhile.

While the evolution of web applications has been a reality, it has also brought new challenges. There are many critical applications on the web that need to assure reliability and scalability in order to support their business models correctly, with a minimal chance of failure. The cloud computing concept emerged to become a rather acceptable solution to the deployment of web services, bringing several advantages like, for instance, elasticity, scalability, robustness, flexibility and fault tolerance. The consequence is that many enterprise services are handled by cloud platforms.

Amazon Web Services (AWS) [2], Microsoft Windows Azure [3], Rackspace Online [4], Heroku [5] and PubNub [6] are only a few of the cloud providers that are publicly available for service provisioning to their customers. They supply a distinct range of service types, including but not limited to, storage, computing, notification and database services. The outsourcing of applications, including web applications, became prevalent for more demanding services. One of the main drawbacks of these cloud providers resides at their interfaces, which are not interoperable. Thus, it leads to additional effort when a service needs to be migrated or augmented to another provider. This is a critical aspect of deploying a cloud service, considering the separate billing costs for each provider.

One solution developed in the past, entitled Service Delivery Cloud Platform (SDCP) [7], has aimed to solve these issues with a middleware infrastructure that provides a rich set of services from several cloud providers, while surfacing a unique abstraction out of the several cloud provider specific Application Programming Interfaces (APIs). The middleware contains a component with the responsibility of managing cloud services and relaying them to end users. A Software Development Kit was included for using and making plugins for SDCP in a Java environment, as well as implementing the components of the platform. However, the SDCP system was not designed for being consumed by web client applications. The *cloud gateway*, a daemon desktop application that served as a bridge to the cloud services covered by SDCP, is no longer worthwhile to use in a web environment. Furthermore, the service aggregation component only supports a basic and technology-specific interface, showing no interoperability or portability concerns.

1.2 PROPOSAL

This thesis aims to propose, design and implement a platform to allow and facilitate the integration of heterogeneous and multi-cloud provider services in web applications. Its goal is to let application developers and administrators manage the available services over an extensible range of cloud providers, combine similar services for hosting cloud resources, and control the access to resources by the applications' clients. The platform may also include additional features that can serve useful in some services. The complete solution comprises an Software Development Kit (SDK) for the development of cloud service integrated web applications, and a set of middleware components with a layer of abstraction over cloud resources for use in web applications, from the application server or the client-side program. The platform is designed for the extended support of additional cloud resource types and cloud providers.

1.3 THESIS STRUCTURE

The thesis is organized in 6 chapters. Chapter 2 explains the current state of the art on cloud computing and interoperability in the clouds, and describes the previous SDCP platform solution and the mOSAIC multi-cloud project. Chapter 3 describes the feasibility of a solution to the new problem, explores its requirements and establishes the complete architectural design of the proposed solution identified. Chapter 4 describes the technological approaches into conceiving the solution, including more detailed aspects of the implemented platform. Chapter 5 presents additional discussion topics of the solution and some benchmarking results. Chapter 6 concludes the thesis with the key points of the developed work, highlighting the main contributions and stating future work.

CHAPTER 2

State of the Art

This chapter succinctly explains relevant research topics to the work. First, an introduction to web services and web interfaces is given. The following sections explain the concepts and challenges of cloud computing, cloud interoperability and sky computing. The final two sections make a brief description of the previous version of SDCP and the mOSAIC project.

2.1 WEB SERVICES

Service Oriented Architecture (SOA) is an architectural style whose goal is to achieve loose coupling among interacting software agents [8]. A service is a unit of work done by a service provider to achieve a specific result that will be consumed by a service client. The SOA style in software development has been in use for many years, establishing a manageable separation of concerns in a system.

Web services are providers of resources created and accessible with the use of web technologies and Internet protocols (Hyper-Text Transfer Protocol (HTTP) , File Transfer Protocol (FTP) , . . .), and they usually follow the SOA architecture. One of the most important aspects of a web service lies at their interfaces, also called APIs, which describe how consumers must communicate with a service, including what can be achieved with it and how messages are structured and exchanged between the consumer and the provider.

Designing an API is a crucial task in distributed system development, since an interface misuse will make the system fail to function properly. If an API has to be extended to support new features of a service, the transition to a new interface definition may be complicated if possible extensions were not predicted beforehand.

In addition, an interface needs to prescribe system behavior, and this is very difficult to implement correctly across different platforms and programming languages. Since creating a new interface for each specific service would be exhausting and error-prone, it is often preferred to take a generic interface and express application-specific semantics to them. This is often a trade-off between performance, extensibility and stability of the API. To collaborate with specifying new semantics and the development of systems complying to such interfaces, Interface Description Languages (IDLs) emerged as formal definition languages for describing software interfaces, often coupled with facilities for documenting the API and generating consumer and provider code stubs for multiple platforms or programming languages.

Two of the most used generic interface styles are next described:

- The Simple Object Access Protocol (SOAP) is an Internet protocol for messaging and remote procedure calls, using Extended Markup Language (XML) as the base message format and usually (although not necessarily), HTTP as the transport protocol. Web Service Definition Language (WSDL) is a commonly used IDL for describing a web service using SOAP [9]. This protocol was very popular in its conception but is nowadays becoming surpassed by other solutions such as REST.
- Representational State Transfer (REST) [10] is an architectural style that defines an interface as a means of accessing and manipulating well-identified resources, using HTTP as the transport protocol and a set of methods for reading and writing resource state. REST is praised by its simplicity, performance, scalability and reliability. In the scope of web applications, client modules for consuming RESTful services can be easily implemented without requiring complex external libraries.

2.2 CLOUD COMPUTING

Cloud computing is a recent model for the provision and release of computer resources and services in a convenient, ubiquitous and on-demand manner. It also makes management nearly effortless while keeping provider interaction to a minimum [11]. Its main characteristics are on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service. Computer and network infrastructures maintained for cloud computing are well adapted for the aggregation of distributed resources into a single virtual system, aiming for virtualization, i.e., decoupling the business service from the infrastructure, and for the scalability of new services. The systems' resource growth capability can be used by its services as it becomes needed. These infrastructures follow one of the four distinct deployment models according to the National Institute of Standards and Technology (NIST) definition of cloud computing [11]:

- **Public cloud** infrastructures are provisioned for open use by the general public. They may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. They exist on the premises of the cloud provider.
- **Private cloud** infrastructures are provisioned to a single organization of multiple consumers, and may be owned and managed by the organization, a third party, or a combination of both. The infrastructure may exist on or off premises.
- A **Community cloud** infrastructure shares the properties of a private cloud, with the exception that it is provisioned for exclusive use by a community of organizations with the same concerns and objectives. It is managed and owned by one or more organizations, a third party, or a combination of both.
- When a cloud infrastructure is composed of more than one distinct cloud infrastructure (private, public or community clouds) by means of standardized or proprietary technology, it is thus categorized as a **hybrid cloud**.

To this day, the cloud computing model is serving a very large fraction of users, spanning several backgrounds from social networks to medical purposes. In addition, the costs of hosting a cloud service are often minimized, not only from the pay-as-you-go paradigm of cloud computing, where the user is billed proportionally to the quantity of resources used, but also because only the cloud provider is held responsible for the cloud infrastructure's maintenance.

2.3 CLOUD SERVICES

NIST has also categorized the cloud computing model into three service layers [11], although a more detailed ontology assumes a total of five layers, including the underlying infrastructure [12]. The definition is described below.

- Infrastructure as a Service (IaaS) provides low-level resources, such as compute, network and storage. Another ontology (L. Youseff et al, 2008) describes storage as a side component called Data-Storage as a Service (DaaS) and low level communication features as part of Communication as a Service (CaaS) , also residing in the cloud infrastructure layer. The user does not maintain the underlying infrastructure, but may control firewalls, network interfaces, operating system and other software, according to the user's purposes.
- Platform as a Service (PaaS) provides a set of frameworks, tools and services for developing and deploying applications to their cloud infrastructure. In this model, the cloud provider maintains the infrastructure, network, operating system and core software, offering a solution to cloud services and utilities.
- Software as a Service (SaaS) provides applications with a particular business logic to end users, consumed either by a web browser or a proper interface application. What the user can control in the application is entirely dependent on the software's business logic.

Under the roof of these layers of cloud computing, many kinds of cloud resources emerged, usually as part of a greater range of services, fulfilling a task that may by itself be categorized into a recognizable service type. Some may provide helpful mechanisms more often fit for a cloud computing environment, which is the case of load balancers [13], [14]. On the other hand, there are many services that can complement existing projects without relying on any other cloud computing capabilities. Next a few of the cloud service types currently used are described.

2.3.1 COMPUTE

Cloud infrastructures provide computational resources to IaaS consumers by permitting access to a Virtual Machine (VM) (seen as a compute instance), the installation of a particular Operating System (OS) and the deployment of applications to run on a cloud. Rather than being used by applications as an external resource, compute services support the deployment and management of a computational node that the application itself is to be hosted in.

2.3.2 DATA STORAGE

Data-Storage as a Service is an extremely prevalent type of cloud service nowadays, used among enterprises and end users for storing their data in a durable, resilient and accessible location. Other requirements are considered by the cloud providers, such as replication, data consistency, and load balancing [15]. However, they may choose to favor one feature over the other, stating their choices as part of the Service-Level Agreement (SLA) .

Not all data storage services are similar, and may fulfill several different kinds possessing the single concept of data persistence in common.

FILE STORAGE

Distributed file storage systems, often called Filestores or Blobstores, are widely available as cloud services, supporting the hierarchical storage of binary data. Amazon S3 [16] and Google AppEngine Blobstore [17] are examples of such cloud services. Dropbox [18] is also considered a storage cloud service, although being a SaaS, mostly aimed to end users and enterprises, rather than developers wishing to have cloud storage in their applications.

DATABASE

Cloud providers may also offer a complete database service, thus lifting the burden of maintaining it from end applications. Likewise, the database is kept at remote disks and can be accessed anytime from any place.

Currently, some cloud providers have relational database services, such as Amazon RDS [19] and Microsoft SQL Azure [20], but non-relational databases, made more popular due to their flexibility and scalability [21], are also available as cloud services (e.g. Amazon SimpleDB [22] and Google AppEngine Datastore [17]). Other existing non-relational Database Management System (DBMS) technologies may be deployed in the cloud. They are often called NoSQL databases and can be categorized into three popular types:

- Key-value databases, like Riak [23] and Redis [24], are focused on storing sets of key-value pairs, with a similar behavior from a hash map, providing fast value access by key indexing.
- Columnar (or column-oriented) databases, like Cassandra [25], contain one exten-

sible column of closely related data, rather than sets of information in a heavily structured table of columns and rows with uniform-sized fields for each record, as is the case with relational databases.

- Document databases, like MongoDB [26], store data as collections of schema-less documents that may often vary in size and complexity.

2.3.3 MESSAGE QUEUES

Message queue services make simple FIFO queues of blocks of data, used for communication between system components. The Advanced Message Queuing Protocol (AMQP) is the protocol most often used for interfacing with message queue systems [27]. Amazon SQS [28] is an example of a message queue cloud service. Furthermore, many existing messaging systems can be deployed in the cloud, such as RabbitMQ, StormMQ and IronMQ.

2.3.4 NOTIFICATION

Notification as a Service (NaaS) came from the need to send and obtain real-time information to a broad range of devices. Applications may wish to receive information from another entity as soon as possible. The push notification concept offers a way of contacting applications when the server has a new message for them. Other solutions for a notification service are also used like, for instance, long polling. This service is often correlated to message queue services, because they need message queues to store this information. PubNub's Data Push service [6] and Amazon SNS [29] are examples of NaaS.

2.4 CLOUD COMPUTING INTEROPERABILITY AND SKY COMPUTING

With the current cloud computing model, developers rely on cloud providers for their end purposes. Implementing a cloud solution, whether it be a new SaaS or any other application or service dependent on cloud resources, often implies that the application (or service) resource will be kept on that provider, and that the developers must use the available API to access them. Conceptually, the previously mentioned service types can aggregate services from multiple and independent cloud providers. However, each service has its own set of terminology and APIs, which make resource migration a difficult task. For instance, without a proper abstraction layer, applications using Amazon S3 for data storage would have to be reimplemented in order to use Google AppEngine Blobstore, without mentioning the required transfer of all data from one cloud provider to another. This makes the solution hard to be decoupled from the particular cloud provider later on (which could be desirable for taking advantage of a better price or Quality of Service (QoS)), resulting in *vendor lock-in*. Its prevention often lies at cloud interoperability: the applications or client services being able to easily change cloud providers for a particular service.

In a similar context, the sky computing concept is defined as the combination of multiple cloud providers in order to create an environment of interoperability, allowing applications to seamlessly use the expanded range of cloud resources [30]. Such resources may either be present in a single provider or originate from an arrangement of similar resources from several cloud providers, thus being often connected to the terms *multi-cloud*, *multi-cloud oriented applications*, or *cloud of clouds*.

Presently, a significant amount of research on designing and implementing an environment for cloud interoperability and sky computing has been made, in a few different ways.

2.4.1 CLOUD INTEROPERABILITY STANDARDIZATION

One of the issues that leads to vendor lock-in and the difficulty of developing multi-cloud services and applications is that cloud providers do not follow the same standards. The creation and wide adoption of a cloud computing standard would neutralize this issue. However, an excessive number of standards, protocols and interoperable APIs were written to this day. The IEEE Intercloud Working Group [31], the Global Inter-Cloud Technology Forum [32], OASIS and the Open Grid Forum [33] are only a few of the organizations focused on bringing uniform standards. The Cloud Computing Interoperability Forum (CCIF) , founded by Reuven Cohen, is a community website for discussion and submission of proposals to cloud computing interoperability, and aims to become the leader and advocacy group in interoperable cloud services. The forum was closed since 2010, and although a revival of CCIF was planned and announced in 2012, no significant progress has been shown since then [34]. CloudAudit, previously known as Automated Audit Assertion Assessment and Assurance (A6), aims to be a common namespace and interface by which cloud consumers can use simple protocols with good authentication to provide access to information from several kinds of cloud resources. It has been part of the Cloud Security Alliance since October 2010 [35]. The Open Cloud Computing Interface (OCCI) is an initiative to create specifications for a common model on Cloud Computing. [36] OCCI has a protocol and API for all tasks of cloud infrastructure management. Some other capabilities related to IaaS and PaaS, such as billing and monitoring, are still under development. The Cloud Application Management for Platforms project defines the artifacts and APIs that need to be offered by a PaaS cloud to manage the building, running, administration, monitoring and patching of applications in the cloud, thus creating interoperability at the application life-cycle management [37]. The Unified Cloud Interface project is also another attempt to create an open and standardized cloud interface [38]. The Cloud Data Management Interface is an international standard for specifying an interface for Cloud Storage [39]. It is currently contemplated in OCCI for the usage and management of this particular resource type.

A list of cloud standards is currently maintained at the Cloud Standards Wiki [40].

CLOUD RESOURCE MIGRATION, REPLICATION AND LOAD BALANCING

Although having cloud services conforming to an open standard facilitates interoperability, the manipulation of resources residing at many different clouds brings more problems than simply dealing with heterogeneous interfaces. Once resources exist in one or more cloud providers, additional engineering challenges arise. This issue is most prevalent in data storage, where load balancing and data replication may be a troublesome task, especially under large amounts of data or database-managed information [41].

A situation where cloud computing providers stop supplying their services will certainly harm the cloud clients, thus why resource replication across multiple clouds may be desirable, even when a cloud provider already provides resource redundancy. Under this situation, the same service can be accessed from another endpoint in case of failure on one other cloud provider.

Computational resources are also target to multi-cloud replication or migration, wishing to achieve higher availability or lower cloud resource costs, and VM placement across multiple clouds has been studied [42].

An additional challenge in cloud and multi-cloud computing concerns the choice of cloud vendor. A Cloud Service Broker (CSB) is an entity that manages the use, performance and delivery of cloud services, and negotiates relationships between cloud providers and cloud consumers [43]. With a CSB, application developers may perform this negotiation process in order to find the most appropriate cloud provider(s) for hosting the application's resources, often by establishing a Service-Level Agreement. CSB solutions have also been implemented and tested [44].

CLOUD PROVIDER ACCESS WRAPPERS

Some of the interoperability solutions create an abstraction layer over cloud providers by specifying and implementing APIs that wrap around the specific services' main features, so that, as an example, using Azure Queues in an application would be developed the same way as with Amazon SQS.

Apache jclouds [45] and libcloud [46] are open-source libraries, the former in Java and the latter in Python, with a portable and extensible set of abstract interfaces to several cloud providers. The Simple Cloud API is a common API to a variety of cloud services, but it is only defined for 3 types of cloud services at this time: file storage, document storage and simple queue [47]. The Apache Deltacloud API is a cloud abstraction API based on REST for accessing to any cloud provider [48]. Although it also assumes the existence of an interfacing server, it is stateless and does not keep any credentials about the client, who needs to send the username/password for the back-end cloud on every request.

The main disadvantages of these solutions are that the cloud resource origins are defined in development time rather than in deployment and/or execution time. Library-based cloud access does not offer brokerage between multiple clouds based on the client's SLAs.

2.4.2 SOLUTIONS FOR CLOUD SERVICE INTEGRATION

Some middleware solutions in the form of frameworks, platforms and system development tools have been made for creating a sky computing environment.

The mOSAIC open-source project [49] combines a full stack of components, tools and APIs to decouple the development of a cloud-based application from its deployment to execution. This project addresses several key aspects to the development, deployment, execution, configuring and monitoring of multi-cloud applications. It also pays a particular attention to the design of the interoperability API aiming to provide programming language interoperability and protocol syntax or semantic enforcements [50], [51].

RESERVOIR has developed a framework for building an even bigger cloud with lower costs, balancing the workloads across distinct geographic locations through a federation of clouds [52].

The OPTIMIS project "is aimed at enabling organizations to automatically externalize services and applications to trustworthy and auditable cloud providers in the hybrid model" [53]. One of its key features is the composition, bursting, and brokerage of multiple services and resources in an interoperable and architecture-independent manner [54].

The SeaClouds project is a work in progress for a seamless development and management of multi-cloud applications in multiple heterogeneous PaaS platforms, taking into account cloud service orchestration, process monitoring, QoS violation tracing and other useful features [55].

Also in a similar context on cloud interoperability, although not attempting to solve the exact same issues, there already are some enterprise solutions to cloud interoperability and multi-cloud computing, consisting of platforms and services that support and facilitate the integration of services from multiple cloud providers.

MuleSoft's CloudHub is a cloud platform for the integration of several SaaS and other cloud services. It is currently available as a commercial solution for enterprises [56].

InterCloud's Cloud Access Provider makes another commercial solution to the integration of cloud services by creating a virtual private network from the enterprise to the cloud providers, thus dealing with security and performance issues [57].

ISSUES

Therefore, there are still some problems to tackle on the subject:

- Too many cloud interoperability standards were created, which leads to none of them being completely reliable at the moment [58]. Cloud service systems have yet a long effort and consensus to make before reaching full interoperability. In the meantime, they should be implemented in a way that separates standards-reliant components from the rest of the system in order to minimize the impact of standards evolution [59].

- Some of the interoperable APIs specify abstractions for only a set of service types, and extensions to this interface are sometimes not supported. As a consequence, the integration of new services may not be a trivial process, if at all possible. In addition, the greatest efforts of interoperability were made over the IaaS kind of resources, with the most critical issue being VM migration. This issue serves no utility to applications not hosted in the cloud, which can still rely on cloud services to function.
- Interoperable APIs for cloud computing on their own may lack the ability to combine, decorate, orchestrate and implement service-oriented control access to cloud resources, thus contemplating the usage of a sky computing environment. For instance, not all storage cloud services support storing ciphered data, but it could be implemented by the abstraction. This example is quite relevant due to the several privacy implications of storing data in the cloud [60]. Another potential use case of these features is to apply resource limits to some cloud services for a set of users, such as setting a maximum storage value for each provider.
- The implemented solutions are mostly aimed at desktop computers and servers. In a web application environment, service orchestration is an important issue [61], and it may be preferable that the client program can consume these services directly, without the assistance of the application server. A web client is not prevented by the network from accessing cloud providers, but sharing the organization's cloud provider access credentials to the client is out of the question. There could be a case where the client possesses an account for that cloud provider, but the cloud resources would have to be shared to that account before being used. In addition, moving to a multiple cloud background highly increases the complexity and difficulty of such process.

2.5 SERVICE DELIVERY CLOUD PLATFORM

The Service Delivery Cloud Platform was created in order to solve interoperability among cloud providers and related incompatibility issues. The main goals to solve by using the platform were: (1) To grant interoperability between different cloud providers, creating an abstract layer for several cloud services; (2) To deliver new services using

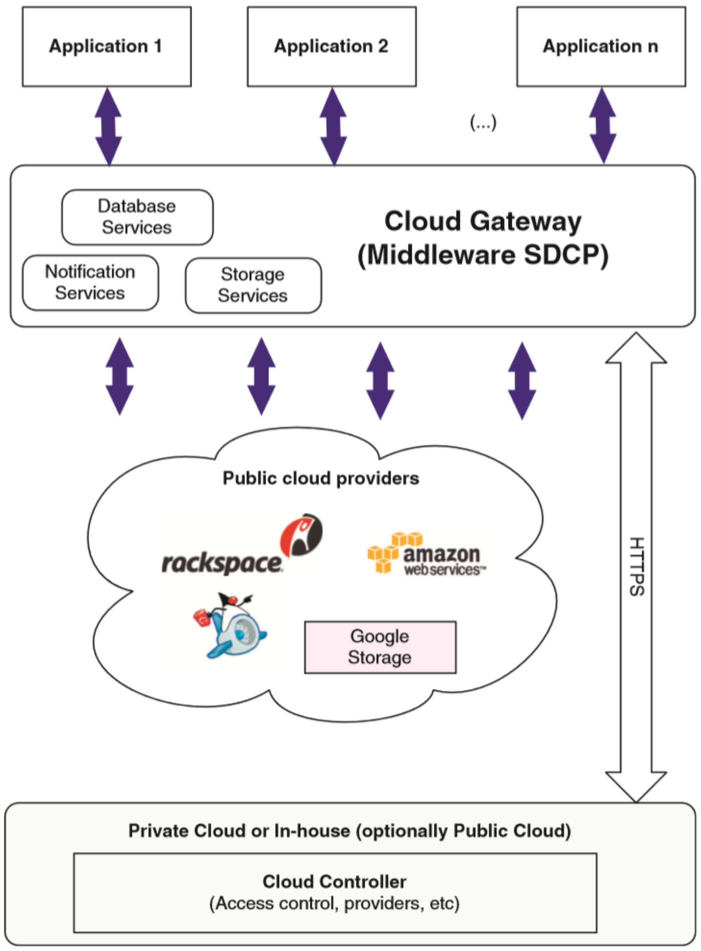

Figure 2.1: Service Delivery Cloud Platform legacy architecture

the available cloud providers, granting interoperability with protocols that already exist; (3) To provide service combination, decoration and orchestration.

The first goal (1) allows the development of applications that operate with distinct cloud providers' services using a normalized interface, proposing a common API that minimizes the actual deficit of cloud API standardization and provides secure and redundant service allocation. One of the key aspects is that applications using SDCP can work alongside as many vendors as desired, taking advantage of the existent cloud providers. This also means that the end application can create a federate view of all available resources, regardless of which cloud the resource resides in. The platform isn't restricted to public cloud providers, as it supports interoperability with other protocols inside more restricted networks (such as private networks) with the development of specific plugins. SDCP allows therefore, the creation of off-premise applications that work inside the organizations, but rely on storage/database resources from the cloud(s).

The initial version of SDCP (Figure 2.1) has two main components:

- The *Cloud Controller* is one of the main components of SDCP. Its task is to aggregate user credentials, handle authentication procedures with cloud providers, implement access control to cloud resources and manage new services. Once installed (preferably in a private cloud, although it would also function in a private server), the user can access cloud services available in the platform via a web service. The contention of user credentials means that the Cloud Controller must be deployed in a trustworthy provider.

- The *Cloud Gateway* is the component that makes the connection between the local applications and the cloud applications. Dynamic plugin loading was featured in order to implement custom services that took advantage of the cloud resources infrastructure without provider dependence. This component was implemented as a daemon process on the end-user machine, which breaks the initial ideal that web applications should not depend on such software from the client.

Aside from the main components, a Software Development Kit for SDCP-based applications was also developed, which was not only used to implement the previously mentioned components, but also aimed at simplifying the development of new applications. The SDK was implemented in Java.

SDCP also contemplated three cloud service abstractions in order to generate the desired interoperability among similar services.

2.5.1 STORAGE ABSTRACTION

DaaS in SDCP provides transparent remote data storage based on the **blobstore** concept. Blobs are blocks of unstructured data that are indexed by a key string and kept in containers. Each blobstore has a list of containers with additional access policies in the form of Access Control Lists (ACLs). Such an abstraction is capable of fulfilling the usual operations of reading and writing blobs and containers, and cloud providers usually support the storage of huge blobs. In the SDCP SDK, the API relied on the design pattern of making connections to a blob as a Java socket, where data would be read or written using synchronous streams.

2.5.2 DATABASE ABSTRACTION

The initial implementation of SDCP supported the **columnar database** type, where entries are contiguously kept in columns, rather than rows. In the SDK, the user would access to these databases using the Java Persistence API (JPA) abstraction, offering a commonly used interface for Java developers while still being able to perform queries on the database.

2.5.3 NOTIFICATION ABSTRACTION

Another common paradigm came from the need to obtain information as soon as possible. With the **publish/subscribe** model, considered in SDCP, entities subscribed to a particular resource will be automatically notified when information is published to that resource. The conceived Java abstraction was based on the Observer pattern.

2.6 mOSAIC PROJECT

The main goal of mOSAIC is "to create, promote and exploit an open-source Cloud API and platform targeted for designing and developing (multi-) Cloud applications" [62]. Once an application is developed for mOSAIC, it may run on any appropriate cloud provider(s). It addresses how applications can be ported from one cloud to another, which cloud is the most appropriate for an application, and how applications can use an expanded range of cloud services spanning multiple cloud providers.

The objectives of SDCP has a great range of similarities with mOSAIC's, thus why this project is so relevant to the thesis. A proper identification of the platform's capabilities will help determine whether an extension of mOSAIC to support the new intended features is more worthwhile than building a new platform from scratch. This topic aims to highlight and analyze the current specifications of the mOSAIC project, in order to identify key features that are similar to SDCP and those to the thesis' problem that are currently not contemplated in mOSAIC.

2.6.1 ARCHITECTURE

The overall architecture of a mOSAIC system [62] is represented in Figure 2.2. The underlined components make the bridge between the cloud application and the cloud resources, which makes them essential to the goals of SDCP. The cloud agency component may also serve useful to the purpose of SDCP when properly integrated (see Section 2.6.2).

The architecture also contemplates the following concepts:

- **Cloud resources** : the resources acquired from Cloud providers during a provisioning phase. They are state-full resources hosted by a cloud provider and are accessible through a dedicated API.
- **Cloud component** : a building block, controlled by user, configurable, exhibiting a well defined behavior, implementing functionalities and exposing them to other application components, and whose instances run in a cloud environment consuming cloud resources.
- **mOSAIC application** : a Cloud application (i.e. a distributed application which consumes cloud resources) developed using mOSAIC solutions.
- **Application descriptor** : a file describing the application composition in terms of components, cloud resources, their relationship and eventual constraints on their behavior.
- **Component list** : the part of the application descriptor listing the components involved.
- **Call For Proposals (CFP)** : a document that defines information relevant for brokering, including the resource listing and brokering strategy policies.

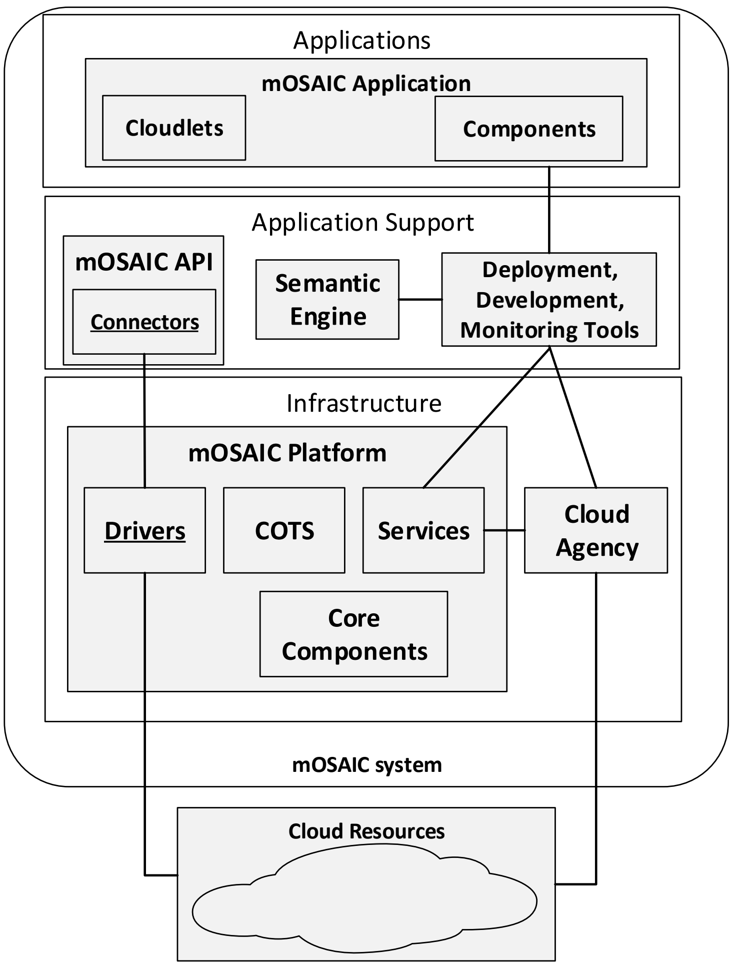

Figure 2.2: Main sub-systems of the mOSAIC architecture

2.6.2 COMPONENTS DESCRIPTION

Four of mOSAIC's components are described in greater detail.

CLOUDLET

A Cloudlet, which term is derived from Servlet, is an independent, stateless element residing in the cloud, which is implemented by application developers in order to fulfill a particular business logic. These components can be implemented using the mOSAIC Cloudlet API.

CONNECTOR

Connectors are the means of cloud resource access from mOSAIC applications, exposing a well defined API for a particular cloud resource type. No more than a single interface is made for any multiple means of obtaining the same resource type. In other words, the same interface could used for file storage in either Amazon S3 or Google Blobstore. Although the two services do not function entirely the same way, a common set of operations and conditions are met to make a common abstraction. In addition, when new cloud providers appear, the interface remains the same. They have an in-process API that is implemented for each resource type and programming language. This means that to make mOSAIC cloud applications in Python that support this resource type, a Python implementation of the connector would be needed. The specifications of the API establish functions meant to be invoked under the scope of a cloud resource accessor.

DRIVERS

Drivers are active components of the platform that form an access gateway from mOSAIC to cloud resources. They can be programmed in any programming language, and rely on native APIs to access the resources. With the Interoperability API, drivers are used by connectors to provide a link between cloud resource invocations of the mOSAIC application and effective cloud operations performed in that cloud service.

A mOSAIC Driver does not necessarily link to external cloud services: a service may be deployed in the same cloud as the application's, which is later used by the connector. This is the case for the available key-value storage connector and corresponding Drivers.

CLOUD AGENCY

The Cloud Agency (CA) is one of the components of mOSAIC, which aims to address the most common business problems when developing cloud applications:

- Specify unambiguously, what cloud resources the application needs and how to request them;
- Discover and select providers that offer the necessary resources;
- Negotiate the best solution for the user, either in deployment time or during the execution of the application, by monitoring resource usage and performance while interacting with the cloud providers or with a broker;

The Cloud Agency maintains the cloud infrastructure in general, without a mandatory intervention of the user (though they can still access a graphical interface to invoke CA services). It can reserve new resources, or release the active ones, and reconfigure itself in the meantime. Cloud Agency can be used to book Cloud Resources and eventually to monitor and reconfigure them also without programming with the mOSAIC SDK. The CA will be able to scale up and down resources or change the providers in an autonomic way. Cloud Agency offers a set of services to the Cloud user by an OCCI extended interface.

2.6.3 API DESIGN

The mOSAIC Deliverable 1.3 [63] makes a clear distinction of an **in-process API** from a **remote API**: an in-process API is the one often used by the developer, implemented for a particular programming language and following a common paradigm. The remote API is the one used to communicate with opaque agents, based on fully transparent immutable data structures, either taking the form of web services, remote procedure calls, message passing or application-dependent protocols. The claim is that most cloud interoperability API standards are focused on defining a remote API, rather than standardizing an in-process API. However, such a focus is natural due to the fact that remote APIs can be independent of the client application's programming language or development platform. Once a remote API is established, any developer can implement a new solution targeting one other technology. The process may also be facilitated with the use of an IDL. Not specifying a remote API means that communication specifications to a web service are left unknown, which could make the resource access from other applications a troublesome task.

The characteristics of the mOSAIC API model are further described:

- The mOSAIC project offers a programming model (cloudlets), facilities for specifying the technical needs of the applications deployed in the cloud (hardware resources, response times, etc.) as well as an environment that will allow the applications to meet the requirements of high availability, fault tolerance and scalability that every distributed system has to offer.
- The API takes an asynchronous, non-blocking approach, in which operations are given a callback for when the operation completes.

- The architecture of the API follows a subdivision of layers (Figure 2.3). From top to bottom:
    1. The Coudlet API layer focuses on the adaptation and integration into the targeted language ecosystem and which are directly used by the developers in their Cloud applications. D 1.3 specifies the operations that deal with the life cycle and overall monitoring of the cloudlet.
    2. The Connector API layer depends on the programming language and provides abstractions for the cloud resources. Each resource type (Key-Value store, columnar database, file system store, resource monitor, ...) may have one connector API. D 1.3 specifies the API for *Key-Value Store*, *File System Store*, *Columnar Database* and *Resource Monitor* connectors, each with a well defined set of operations. Not all of them were implemented by the mOSAIC consortium, however. The Columnar Database and the Resource Monitor connectors currently do not have an implementation.
    3. The Interoperability layer addresses the low-level issues related with resource interaction (not for the final developer, but for the platform developers) which wrap and enrich it by offering high-level functionalities as part of the mOSAIC framework (i.e. object marshaling, schema validation). The final aim of the interoperability API is to offer a solution which enables Connectors to invoke Driver operations, having no knowledge of the underlying technology and (if needed) residing on different machines. Although D 1.3 specifies the requirements of the interoperability API, no specific protocol or data structures are defined in the document. In practice, the available implementations used ZeroMQ [64] and Google Protocol Buffers [65] to implement them, disregarding existing, more interoperable API implementations and focusing on speed.
    4. The Driver API layer wraps the existent native APIs, offering a first level of uniformity by exporting all the resources of the same kind through a unique interface. The specification introduces the concepts of *Cloud Datastore* (service model in which data is stored, maintained, managed, backed-up and accessed remotely), *Datastore Object* (the smallest entity that could be stored in a Cloud Datastore) and *Repository* (a logical storage location that holds Datastore Objects). Although this API was described in Deliverable 1.3, it was actually not used for implementing the existing driver modules for the mOSAIC Java platform.

5. The Native API layer lies at each cloud provider, outside of the scope of mOSAIC, making the final endpoint for the cloud resources. Each provider uses its own programming language and communication protocols, therefore it is necessary to abstract the functionality offered by each one to reach a certain level of interoperability between different vendors.

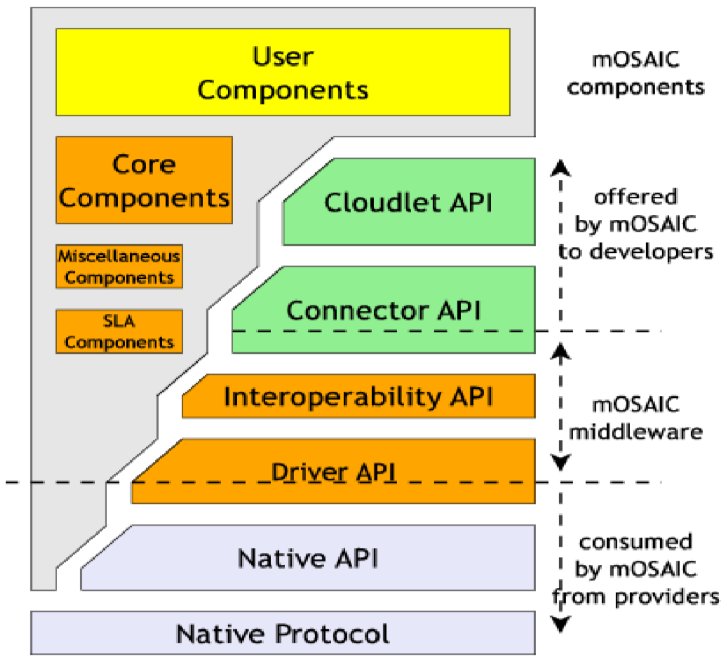

Figure 2.3: API layers of mOSAIC

CHAPTER 3

Platform Design

This chapter discusses the use, feasibility and implications of the proposed platform, as well as defining its architecture and related concepts.

3.1 A cloud service platform oriented to web applications

Over time web technologies have evolved to give web developers the ability to create new generations of useful and immersive web experiences [66]. The existence of a common client-side scripting language allowed web developers to execute a part or all of the application's logic directly on the client. The beneficial outcomes of this approach are the reduction of work-load on the web server and the potential room for improved user interaction, since some results may be rendered without waiting for the server. The former advantage is particularly important, considering that they may provide the service to a huge number of clients. Therefore, the establishment of services in a cloud resource integration platform in a web environment is deemed to bring advantages to web application development.

The concept of a platform granting the use of multi-vendor cloud services to client-side web applications may raise some questions. Namely, why the application developer would prefer to access a middleware system to consume the cloud services needed, and second, why the access to such resources could not be performed by the server application.

Outsourcing web applications to the cloud is still worthwhile in more than a single aspect: the application may rely on particular cloud services such as storage and database, with their inherent advantages; and the web application as a whole may be deployed to a cloud provider as a SaaS. Usually, these applications are deployed over a third party SaaS, which frees the developer from handling the underlying cloud infrastructure. In such cases, it is only natural to rely on services from the same cloud provider [67]. However, cases of vendor lock-in make a clear disadvantage of this approach. Therefore, once an application is developed for an interoperable cloud platform, it can transparently make use of several cloud providers through a portable API, decreasing the vendor lock-in effect for eventual future migrations of the application. Furthermore, since the client-side program runs on the client rather than the cloud infrastructure, it is free from PaaS-specific implementation details.

The second question is already answered by the statement that the web application, although provided by the web server, can be heavily decoupled from the server, and the collection of data (or other results from the use of a service) from cloud resources directly to the client can also be part of the decoupling process.

3.2 PLATFORM REQUIREMENTS

The outline and specification of platform's requirements is important for a proper design of a new SDCP version oriented to Web client applications.

Access to multiple cloud services. The key concept of SDCP is to deliver cloud services from multiple cloud providers. The platform must expose web services that serve as a gateway to the effective resources being handled. In addition, the platform must support extensions that increase the available range of cloud providers, including access to private cloud infrastructures.

Cloud resource type extensive. There are many useful cloud resource types, and the creation of new types may be as simple as providing a new kind of service residing in a cloud infrastructure. The platform must be extensible in order to support additional resource types.

Resource access abstraction. One of the most important goals of SDCP is to grant interoperability between different cloud providers, creating an abstract layer for several cloud services. Entities may access and manipulate resources with the same API, as long as they are of equal (or similar) kind.

Federated, multi-tenant view of resources. Rather than considering separate scopes when accessing resources from distinct credentials, SDCP should allow a broad view of resources of each type, creating a uniform cloud resource environment. The implications of a complete resource origin abstraction emerge when, for example, a list of files in a file storage cloud service may contain files from distinct cloud providers, or even simultaneously contained in more than one cloud.

Resource access control. When dealing with a uniform cloud resource access point, where several entities can register and manipulate resources, it is also required that they can share their own resources to other users of the platform and carefully define to what extent, regardless of the resources' real origins or required credentials for the access to such resources.

Web application support. As mentioned in Section 3.1, the client running the web application contains web service consumption capabilities. As part of SDCP's requirements, the delivered services and respective cloud resources must be easily accessible by the web browser, without installing additional software in the client. In addition, a JavaScript library should be created for interfacing with the platform.

3.3 THE ARCHITECTURE

The proposed architecture for SDCP (Figure 3.1) follows a similar approach to its previous version, although more adapted to the scope of web applications.

- The *Cloud Controller* component is kept as an essential middleware entity, with the same goals described in Section 2.5. This component hosts web services that allow clients to be granted access to cloud resources, registering agents, cloud access credentials, cloud provider details and resource type information.
- The *Remote Connectors*, or resource gateways, expose a service for access and manipulation of cloud resources, each for a specific resource type.
- The *Cloud Provider Access Drivers* implement the effective access to a particular cloud provider, whether it be private or public.

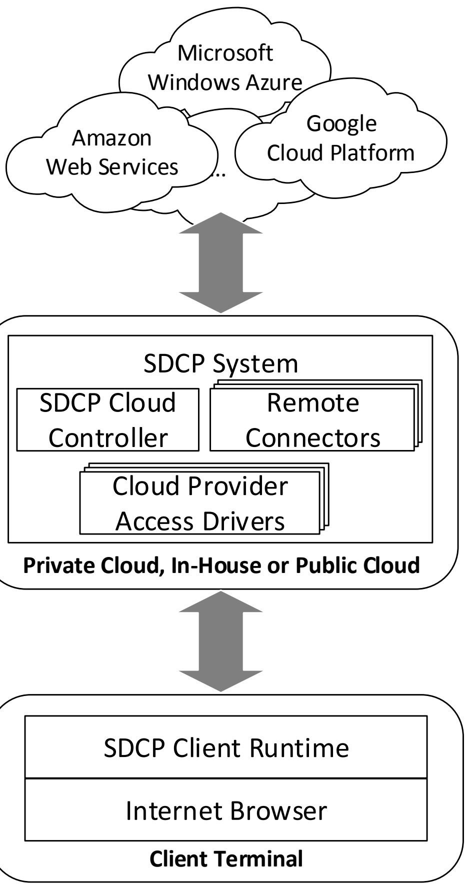

Figure 3.1: SDCP Concept Diagram

- The *SDCP Client Runtime* (or just *SDCP Runtime*) is a JavaScript module for interfacing with the Cloud Controller and the registered remote connectors. It is transferred alongside the web application's client-side code and contains the skeleton of cloud service API aggregation. Cloud service interface modules, either contained or referenced in a resource type descriptor, contain functions and other data structures for using a particular type of cloud resource, and can be loaded when required by the underlying application.

3.4 CLOUD CONTROLLER MODEL

The Cloud Controller specifies and implements the complete abstraction with its own data model (Figure 3.2). Thus, a unique identity is given to cloud service users, cloud providers and resource types. With this data model, the Cloud Controller can:

- Authenticate a known agent (see Section 3.6);

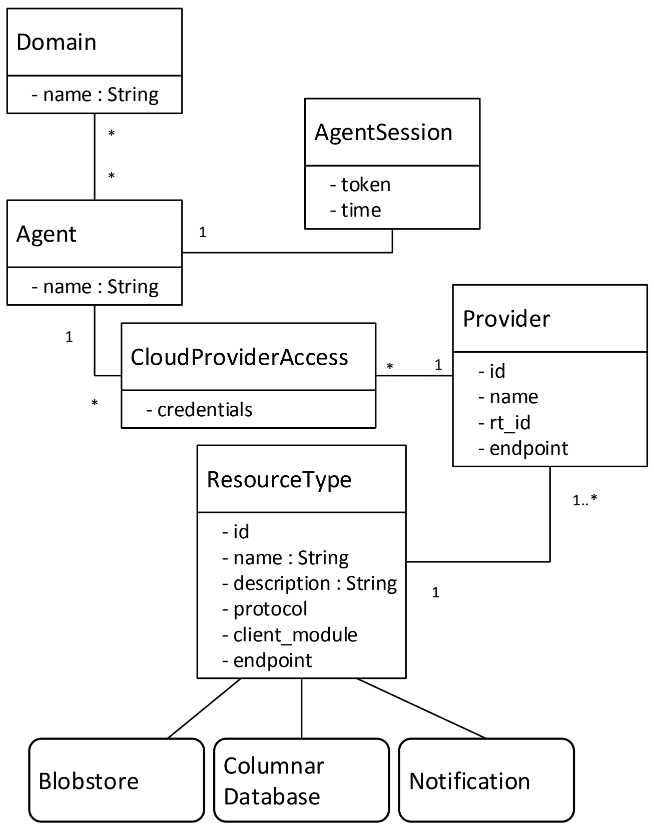

Figure 3.2: SDCP Cloud Controller Model

- Register and provide information about the available resource types and cloud providers;
- Keep cloud provider access credentials for each known agent.

The controller model also introduces new concepts described below.

- An **Agent** is any entity wishing to access or share cloud resources. A known agent has its own name and password, which can be used for logging in to the service.
- When an agent logs in, a temporary **Agent Session** is created, which is identified by a token. All subsequent operations of the agent require that token for authentication purposes.
- A **Domain** makes a logical aggregation of resources and agents. A list of agents is kept in each domain, and access control policies may be defined in the scope of a domain, so that all agents belonging to it may be granted resource access permissions.

- The controller keeps all **Resource Type** descriptors as simple data documents. The essential information of each descriptor is the resource type identifier and the various means of accessing the cloud service.
- A list of available **Cloud Providers** is kept, so that agents can identify the possible cloud resource origins and register credentials for them if needed.
- A known agent may hold **Cloud Provider Access** credentials, thus defining what cloud providers are available for that agent and consequently, what types of resources can be created. The credentials required for the specific authentication process are often, but not necessarily, a user name and password. These are not to be confused with the Agent credentials described in Section 3.6.

The choice of describing credentials is complicated, because not all cloud providers provide the same means of authentication and cloud access. Some vendors such as AWS and Google support cloud access with an access key and a secret key, but many other identity managements and authentication mechanisms can be used [68]. However, the two main issues that SDCP has to address is the containment of any form of cloud provider access credentials, plus the transfer of these credentials to the cloud access drivers.

3.5 CLOUD RESOURCE MODEL AND MANIPULATION

Ultimately, the platform will have to perform a bridge between agents and the actual resources, with all its inherent implications mentioned further in this chapter. Whereas the previous version of SDCP used the Cloud Gateway to achieve this, the concept was extended into **remote connectors**: components with a web interface of their own that serve as gateways to resources of a particular type.

A remote connector is registered to the cloud controller when deployed, thus making the platform support the new resource type. The registration process is made by sending a document describing the resource type, and an endpoint address to the remote connector's exposed web interface.

The application developer can request the previously mentioned document in order to retrieve the endpoint to the remote connector's service. From then, the application agent can communicate with the remote connector, sending the agent session token for authentication purposes. Figure 3.3 shows a simple example of how a client can

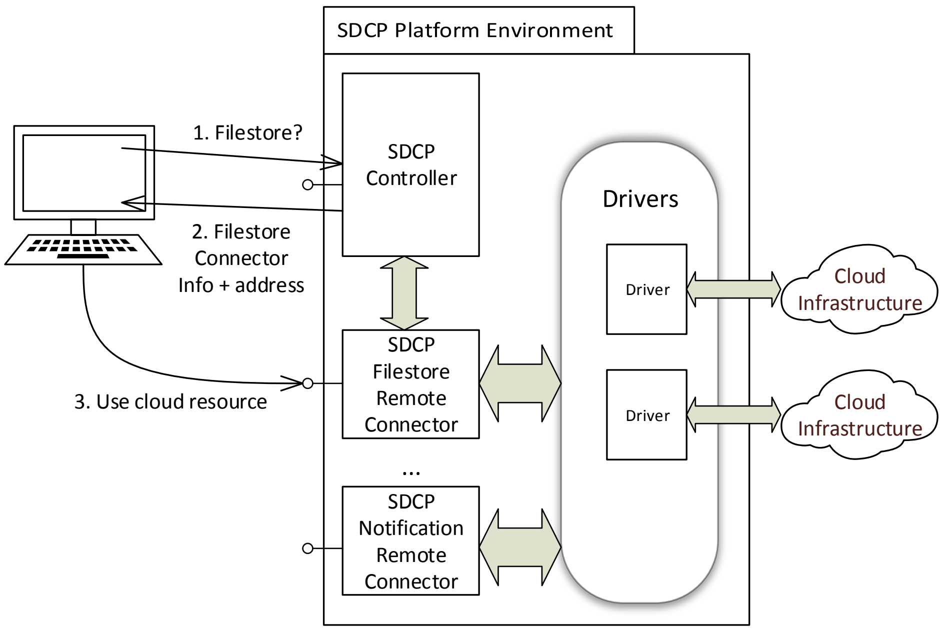
SDCP Platform Environment
1. Filestore?
SDCP Controller
2. Filestore Connector Info + address
Drivers
Driver
Driver
Cloud Infrastructure
Cloud Infrastructure
SDCP Filestore Remote Connector
3. Use cloud resource
...
SDCP Notification Remote Connector

Figure 3.3: SDCP Remote Connector model, containing an example of dynamic remote connector access information.

dynamically obtain resource type information, including the endpoint to the remote connector's web service. The resource type defines the API of its associated remote connector, including the operations that are applicable to the resources and what access control policies can be made. The tasks of controlling resource access, managing registered resource meta-data, and translating requests from the agent, are left to this component. Operations requested may translate to zero or more operations on the external cloud services, depending on the operation itself.

Each remote connector keeps a well defined hierarchy of resources, starting with the enclosing resource *scope*: either an agent name that owns the resource or a domain name that the owner agent belongs to (further distinguished with a wild card). A root resource is the type of resource that must be created before creating smaller parts on that resource's scope. A whole database, a Blobstore or a notification channel set are examples of root resources that would be considered in the assigned remote connector. Hence, the Universal Resource Identifier (URI) is used to identify the full resource path (Figure 3.4). As an academic example, the URI `/App/D/G/xy` would point to a resource called `xy` inside resource `G`, which is in root resource `D`. `App` would be the name of the agent which owns the resource. Such an example would be valid for as long as `D` was registered in the resource scope of the agent itself (rather than a domain).

To sum up, the remote connector can:

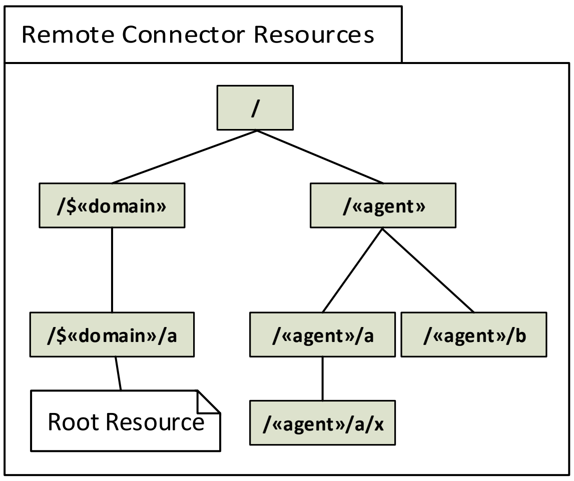

Figure 3.4: SDCP's cloud resource hierarchy model

- Communicate with the cloud controller in order to register itself to the platform and obtain agent session and cloud access information;
- Issue operation invocations over a cloud service of its type, to cloud provider access drivers, when so is requested by an agent;
- Translate a cloud resource ID to a concrete resource from a specific cloud provider, or more than one cloud provider;
- Keep track of cloud resource usage with a form of resource monitoring;
- Evaluate existing cloud control policies on an agent in order to know whether an operation over a resource is allowed.

3.6 AGENTS, AUTHENTICATION AND ACCESS CONTROL POLICIES

The **agent** is the base SDCP user (Figure 3.5). The agent authenticates to the Cloud Controller by sending a user name / password pair in a secure connection, which will be replied with a time-bound token object for use in the following operations. Unknown users (also named *public agents*) perform the same login process without sending any credentials, in order to receive a session token and create a temporary agent that will serve as an ID for the public user.

Before an agent session expires, it can be safely extended with a renewal operation. This will generate a new session token while keeping the agent ID unmodified, even in

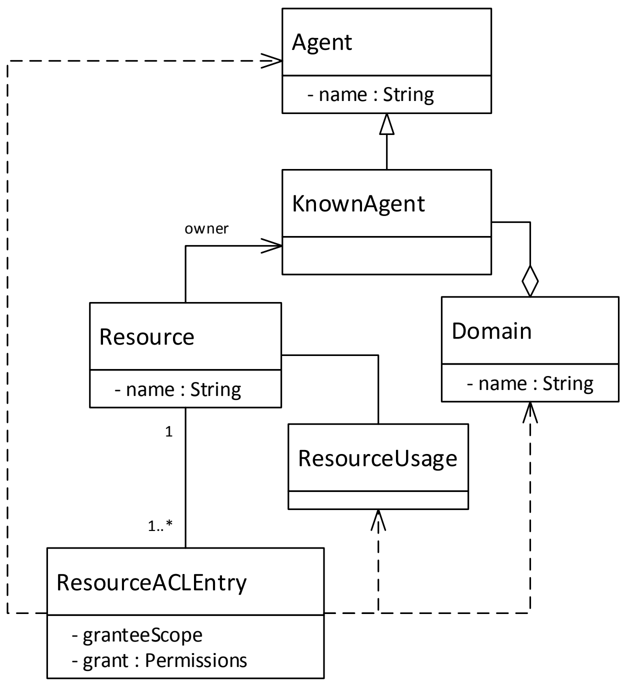

Figure 3.5: SDCP Agent Authentication and Access Control Diagram

the case of an unknown, public agent.

Access Control is a means to apply selective authorization to the use of resources from a service [69]. The agent authentication mechanism aims to identify special agents in the service, particularly the web application administrators, who are then granted additional operations on the service's cloud resources. To achieve this, SDCP allows for the creation of an ACL in the available resources. Technically, these ACLs are lists of tuples containing:

- An identification key of the resource that is being granted (or denied) to whichever entity is in the scope of the entry.
- A scope, defining the situations where the permissions in this entry are to be applied. The scope can mention a particular agent, agents in a domain or the set of all agents (including public agents identified during the session). The scope may also apply to a particular provider or the set of all providers.
- A set of permissions, which are predicates that define the granted operations or resource usage limits.

By default, the owner of a new resource is granted full access to them. Sharing a resource to another agent is as simple as adding a new entry to the ACL. How this action is made depends entirely on the remote connector's API. The existence of a cloud provider ID field when defining a scope is also relevant here, because of the possible usage limits: if, for instance, a cloud provider supports 2 GB of free blob storage, a limit may be applied to avoid taking additional costs. Naturally, unless another cloud provider was set for the same blobstore, operations that surpass such a limit would fail.

In the context of DaaS, ACLs serve as a means to specify whether the user can read from, write to or create new containers or blobs. Additional access control policies make sense at the level of the SDCP platform, due to its nature of provider orchestration, such as applying a maximum storage capacity of an agent's owned blobs to each provider. The implemented policies may apply to known users or public agents.

3.6.1 RESOURCE DELEGATION

Resource delegation can be done by a known agent wishing to share resources to other agents. This pattern is quite relevant in SDCP because of the nature of web applications having a client side and a server side. The main goal of resource delegation is to let the server application establish well defined restrictions on the client's resource access using SDCP. Although all clients will have access to this platform, they are not given the means to create or use existing resource unless permissions are given with the ACL mechanisms.

Client-side users of the web application will often take the role of public agents in SDCP, which cannot register cloud provider access credentials, nor have a resource hierarchy of their own. With resource delegation, the server of the web application can provide controlled resource access to these clients, without itself playing a constant middleware role, by applying new ACL entries to existing resources via the remote connector's interface.

Two means of achieving cloud resource delegation are considered, although the second one is shown to be more appropriate in most cases:

1. The client can create a session for itself and share its token with the server agent, as show in Figure 3.6. The client agent's name is not to be recognized by the application server in this situation. Instead, sending the client token alone will allow the SDCP system to obtain all session information required, including the translation of the token to an agent name. After the first session renewal however, the application server will no longer have a means of changing that

client's resource permissions, unless by being given the token again.

2. Alternatively, the server may choose to create a new public agent itself, grant resource permissions to the new agent, and then transfer the session token for cloud resource access on the client (Figure 3.7). This server-oriented approach of resource delegation will allow the server to keep track of all clients' agent names and continue changing resource access permissions, even after further session renewals. The public agent will not know its own agent name unless it is transferred by the server as well, but this information is usually unnecessary for the client, while being more useful for the application server. This delegation procedure is only possible when granting cloud resource access to public agents.

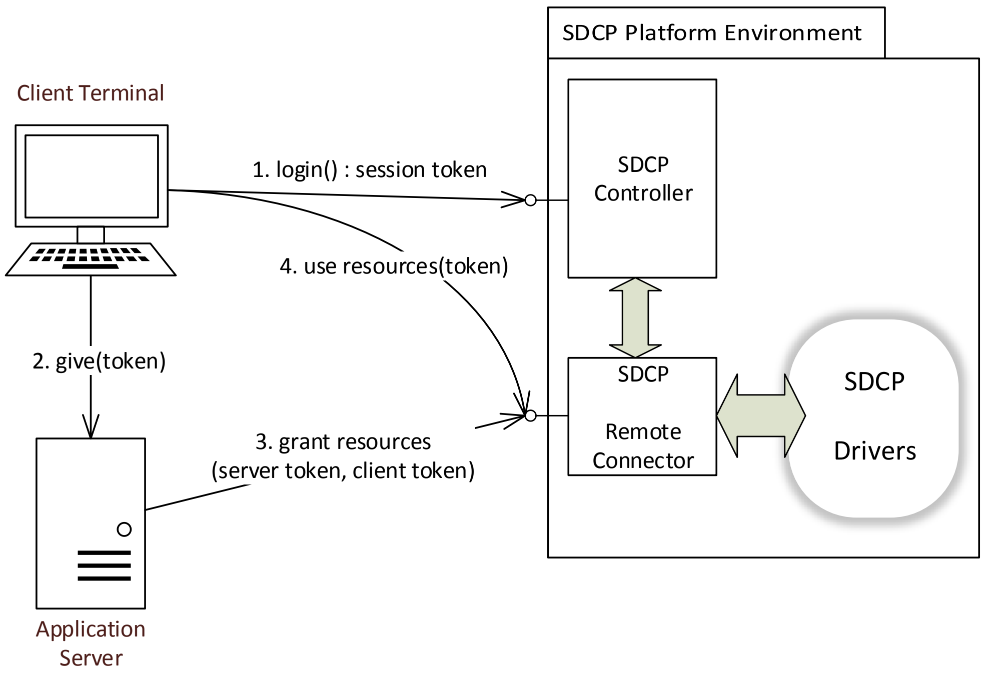

Figure 3.6: Resource delegation in SDCP, client-oriented approach.

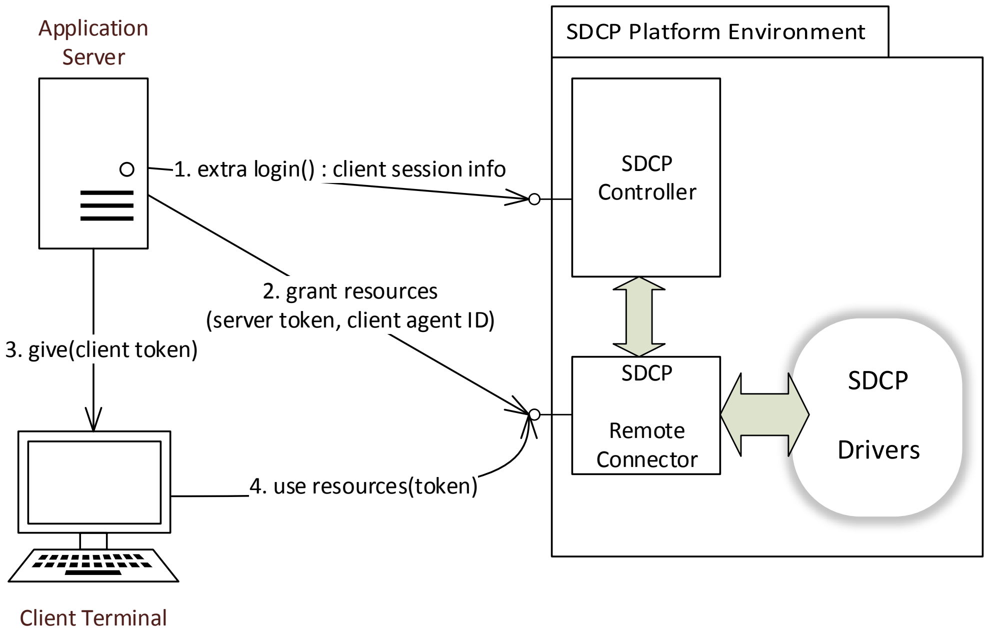

Figure 3.7: Resource delegation in SDCP, server-oriented approach.

3.7 THE API

The cloud controller's service includes operations for the login procedure mentioned in Section 3.6, along with the registry of cloud access credentials per agent, the listing of resource types and the specification of the cloud providers that may host the available resources.

Access to resources of a particular type are installed in a remote connector of its own, exposing a public web service. There are no restrictions on how the API should be specified, but it should be most adequate to the resource type that it applies to. In addition, each resource type should also have a software module for interfacing with the remote connector on the client side of the web application.

In situations where resources should rather not be used via HTTP, such as the notification service, the remote connector can rely on other protocols to communicate with the agents. The operation to perform on that resource, along with additional parameters, may be passed with the use of query string parameters, if so is supported in the protocol involved. Any other means defined in the resource type's API can be used, as long as they are well documented and the implemented client modules comply to them.

When creating new resources, users of the API can specify the cloud providers to use or let the entity choose one from the available cloud resource origins. If access to one of the providers becomes unavailable, the remote connector involved may automatically try a different origin when requested, for that resource.

3.8 THE PLUGIN SUPPORT

Modular design architectures offer the major advantage of being able to expand the system by plugging new modules, without changing the core components. The SDCP architecture follows this paradigm for supporting distinct resource types from several cloud providers. This is achieved with the concept of plugins, which implement the *facade*'s between the cloud agent and the cloud providers.

The previously designed platform relied on plugins for the extension of specific services to the system, such as the "PACS Cloud Gateway" for creating interoperability between the Digital Imaging and Communications in Medicine (DICOM) world and the cloud [70]. They were implemented and packaged as Java archives, which were loaded by a jar/class loader module in the SDCP-SDK. However, the translation of this model to the web raised an issue, because these plugins were deployed in the Cloud Gateway component, which could no longer be applied in the new architecture. Having the plugin system installed as part of the cloud controller was considered and deemed feasible, but the remote connector and cloud provider access driver model brings more advantages in terms of scalability, while only slightly crippling performance and resource usage, with the increase of system processes and inter-process communication channels.

The new plugin model considers two plugin types:

- Interface plugins specify a particular resource type (and related sub-resources), including what operations can be performed.
- Implementation plugins, also called cloud provider access plugins, make the bridge between a cloud controller's resource type and a specific cloud provider. The creation and deployment of these plugins enlarge the range of existing clouds for hosting the services.

This duality is meant to extend the platform in two aspects: create new abstractions for potential cloud resources; and the available cloud services that implement such resources.

This plugin model does not depend on a framework. Instead, each plugin is an active component with its own system process. Plugin deployment includes registering the component to the cloud controller, using an appropriate interface. Interface plugins take the form of remote connectors, and once deployed, they will make the cloud controller recognize the new resource type. Implementation plugins take the form of cloud access drivers, which will likewise make the cloud controller recognize new origins for cloud resources.

3.9 THE SERVICES

In the scope of SDCP, the service of each remote connector establishes an end-point for all resource requests of its type. Rather than taking into account the various cloud services' locations in the application logic, SDCP aggregates them as resources into a single service.

Some of the potential resource types to be implemented in the platform are blob/file storage, database (expanding to different types of databases) and notification system. Three of the types are further explained below.

FILE STORAGE

The model described in Section 2.5.1 regarding the previous approach to the Storage abstraction is still suitable for an implementation. Containers and files (blobs) are seen as resources that can be identified with a URI describing the full path from the service's root to the final blob or container of choice. Some of the operations can be identified: blobs can be created, written, read and removed, whereas containers can be created and removed. Other high-level operations may be implemented, such as copying and moving resources in the storage tree or between cloud providers. In addition, all resources have an access control list, defined with additional operations, which state reading and writing permissions over a file or a container, whether it be for accessing the resource's data or meta-data.

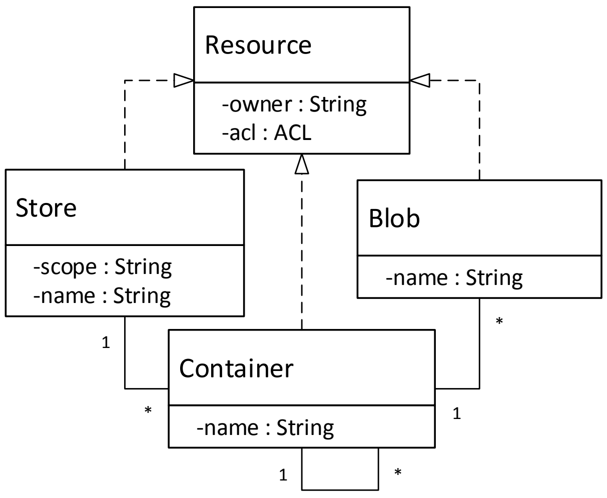

Figure 3.8: Filestore resource model.

This resource type may be provided by Amazon S3, Azure Blob Storage, Google Cloud Storage and many others. These services are not entirely identical to each other, but the middleware is able to establish a common operation set with the same results.

SIMPLE DATABASE

Non-Relational Databases have become a trend these days, bringing high performance, availability and flexibility. This simple database model assumes a set of domains / collections, each with a variable number of items and a variable number of attributes in each item (Figure 3.9).

Operations can be performed with simple `get` and `put` functions, or with the execution of queries written in a subset of Structured Query Language (SQL) . Reading

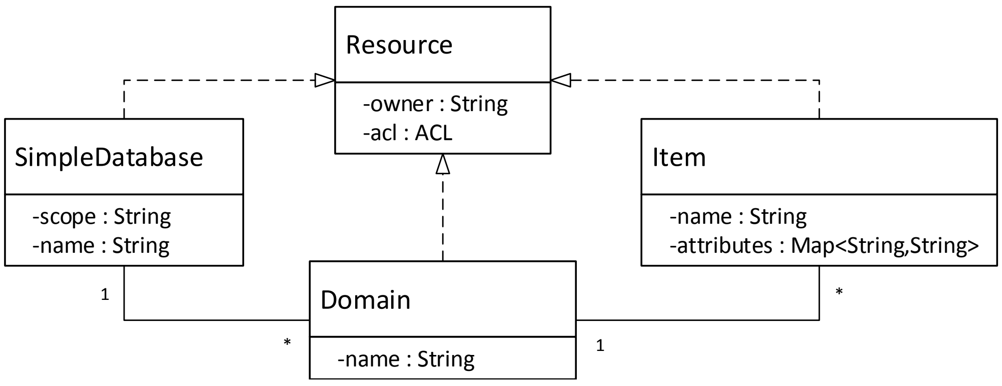

Figure 3.9: Simple database resource model.

and writing permissions are also applicable in this context: A simple analysis of such queries may identify the sort of operation involved over the database, thus evaluating whether the agent can perform it.

This resource type may be provided, for example, using Amazon SimpleDB or Azure Tables. Other non-relational database technologies may also be abstracted, with the appropriate middleware implementation. This task however, will make specialized features of a DBMS unreachable, as the common operation set becomes limited to basic operations. In addition, as explained in Section 2.3.2, not all schema-less database technologies function the same way. The creation of more than one resource type for non-relational database access would be another possible approach.

NOTIFICATION

The proposed model, represented in Figure 3.10, envisions a notification root resource as a set of channels. Each channel is a messaging entity to which agents subscribe for receiving published messages in real-time. The **publish** and **subscribe** are the most relevant operations, along with the creation of new channels. Access control lists would state whether an agent can create new channels in a channel aggregator or, in the case of a particular channel, whether the agent can subscribe or publish messages.

The remote connector may expose an interface based on HTTP, but more clever solutions can be implemented, by using protocols supporting asynchronous two-way communication, such as Extensible Messaging and Presence Protocol (XMPP) [71] or WebSockets [72].

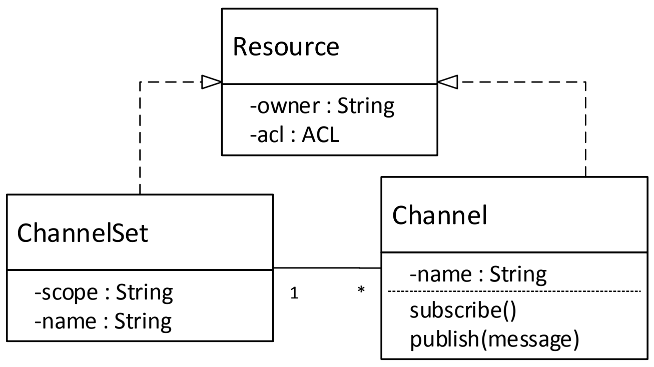

Figure 3.10: Notification service resource model.

CHAPTER 4

Platform Development

Once the architecture design of the platform was laid out, a proof-of-concept implementation ensued. This chapter guides the reader throughout the integration approach and the technical choices involved.

4.1 PRE-DEVELOPMENT CONSIDERATIONS

The initial research of a solution to the main thesis problem led to a different model from the one defined later on: the cloud controller was the monolithic component of the platform, and plugins would be attached to this component as software modules running in the same process. Taking advantage of the existing implementation of SDCP was considered, but there were issues that deemed too difficult to tackle: it was implemented for deployment in Google App Engine (GAE) , using snapshot versions of GAE that were in constant development back in that time. Being constrained to outdated libraries would make deployment a troublesome task.

Instead, additional research was taken for choosing a server application library or framework, with a few remarks stated in Table 4.1. The essential requirements for these technologies were support for HTTP Secure (HTTPS) and horizontal scaling. Furthermore, the use of programming languages that were out of the scope of Java or JavaScript was prevented. The final decision was to implement the cloud controller in Node.js, thus taking advantage of its network application nature and JavaScript environment.

Technology	Language	Observations
Node.js	JavaScript	Low-level, but the language is flexible and many higher-level modules are available. The fact that it runs JavaScript may help the creation of new web SDCP plugins and make reusable server and client code.
Google App Engine	Java, Python, ...	Previously used, but the implementation became outdated quickly due to new versions coming up.
RedHat WildFly (JBoss AS)	Java EE 7	Easy to start with, and provides good administration tools. Supports horizontal scaling since JBoss AS 7, with `mod_cluster`.
Oracle GlassFish	Java EE 7	Provides administration console. Supports horizontal scalability since v3.1.
Apache Tomcat (TomEE)	Java EE 6	No additional installation required in deployment. Supports horizontal scaling since Tomcat 5.
Eclipse Jetty	Java	Supports horizontal scaling with proper extensions (e.g. `mod_proxy_balancer`).

Table 4.1: A simple comparison of web server applications and technologies.

4.1.1 MOSAIC CONSIDERATIONS

Throughout the research of cloud-based platforms, the mOSAIC project grabbed some attention that led to pondering whether it could be used as a starting point for an implementation of SDCP, so as to not only contribute to the project, but also to catalyze the implementation process. Section 2.6 shows the study made on the project, mentioning its most important features to the context of cloud service delivery. The final decision was to make use of mOSAIC, which brought a different level of challenges and ideas for the final architecture design of SDCP.

ARCHITECTURAL GOALS AND CONSTRAINTS

mOSAIC aims to be a solution to application developers not only wishing to leverage their applications to the cloud, but also to enhance existing applications into a managed sky computing environment. The application's composition into multiple independent components also grants greater scalability. Such applications must follow specific guidelines in order to be deployed as a mOSAIC application, which includes relying on the API for internal and external communication purposes. Once deployed, the application may use the resource providers that were previously established in the application descriptor.

An essential issue arises from the given development pattern: What if the application itself is not sitting in a cloud? There are several implications that may lead to avoiding the deployment of the entire application, but simply relying on some cloud resources. In the case of web applications, its client side program may contain the essential business logic for both interacting with the user and consuming a broad variety of services.

The interface plugins defined in the new design of SDCP are analogous to the concept of mOSAIC Connectors, with two major exceptions:

- SDCP interface plugins must define and implement a remote API, with a relatively simple translation to an in-process API for JavaScript.
- Since only the mOSAIC application is meant to be granted access to such resources, no access control policy specifications for the resource type are contemplated. This means for instance, that nothing states what set of operations on a blobstore would be allowed by a user with read-only access. Additional operations would have to exist in order to grant new permissions to clients, by manipulating the resource's ACL. These operations can however, be conceived in a generic, connector-independent manner, as long as the permission types involved are well outlined for each resource type. Often, the *read* and *write* permission types would be used, but they may not make sense in other resource types: *publish* and *subscribe* can establish permission types for a notification service.

Vendor-specific solutions to cloud resources provide a remote API, which can be invoked by a JavaScript application running in a web browser, but they also require an authentication process that must be performed by the server. This leads to requiring a component implementing controlled exposure to cloud resources in order to let applications use such resources directly. SDCP keeps provider access credentials, registered by authorized user agents in deployment time and later transparently used for resource usage. In the context of mOSAIC, this logic would have to be implemented in the application layer, regardless of whether the rest of the application was deployed in it.

INTEGRATION PROPOSAL

With the analysis of the mOSAIC platform, it is made clear that it solves a relevant set of problems regarding cloud interoperability and sky computing. However, the project did not target some of the goals of SDCP.

Access to cloud resources is contemplated with connector-driver capabilities, and the given design of the API could be implemented in JavaScript. However, the connector API is only accessible inside the boundaries of mOSAIC as a user component, and are not exposed in the form of web services. In order to fulfill the requirements of SDCP, a user must be able to authenticate itself, possibly by logging in to a credentials system, and remotely create, share and use the available range of cloud resources remotely, regardless of the origin of requests.

The previously identified components of SDCP still apply in the integration. The mOSAIC drivers fulfill the same goals of cloud provider access plugins, with the advantage of being platform independent: driver components are pieces of software that only need to show concerns of native cloud resource access and exposure, without having to be implemented in a specific programming language. The initial version of SDCP supported plugins implemented in Java only. In addition to implementing a bridge to a cloud provider, these drivers must be registered at the cloud controller as cloud providers. This procedure can be done manually after its deployment, or by making a custom Driver that automatically registers itself to the controller.

For the remote connectors, the mOSAIC connectors can implement the abstraction to the cloud provider access drivers. To fulfill the remaining goals of SDCP, each remote connector has to augment the connector's interface with access control operations that modify the resources' meta-data, and expose a web service that allows the execution of such operations by authenticated and authorized agents. The design concepts considered in Chapter 3 are to be reflected in the implementation, including the resource hierarchy model and the remote connector's registering process to the cloud controller.

4.2 DATA MODELS AND PROTOCOLS

Due to its web environment nature, most of the objects transmitted from the platform are formatted as JavaScript Object Notation (JSON) documents. The use of this notation makes objects smaller in size than with XML and facilitates their usage in JavaScript applications, which contain built-in support for JSON parsing and manipulation.

Table 4.2 lists the considered key entries of the JSON objects used throughout the interaction with the cloud controller. The `RTDescriptor` is a document transmitted by a remote connector in order to register itself to the platform.

The `endpoint` of the remote connector specifies the effective public address that clients should connect to in order to gain access to resources, whereas the `endpoint` of a cloud provider descriptor is an internal address that should only be visible and accessible by the platform itself. In either case, the document may rely on Internet Protocol (IP) addresses or domain names, and should specify both a schema (`http`, `https`, or another) and an access port number.

Considering what was discussed as part of the cloud controller model (Section 3.4), the use of a data type ample enough to support all cloud provider access credentials is sufficient for the objectives of SDCP. In practice, the Object used for the `credentials` entry is a simple string-to-string map.

Some of the entries are optional, such as the description of the resource type. The `protocol` entry is also not required, but when combined with addition information, such as the set of `operations` and the set of `permissions`, they could be used for automatic interface access module generation. An alternative to these entries would be a description of the web service with an IDL.

All `id` values must be distinct among the same data type. This only applies to the error `code` when interfacing with the same cloud controller service (see 4.3.1), which will always show the same error `message` related to that code.

Data	JSON Model
RTDescriptor	`{ id : string, name : string, description : string, protocol : string, endpoint : string, client_module_url : string, operations : Object[], permissions : Object[] }`
CloudProvider	`{ id : string, name : string, rt_id : string, credentials_format : string, endpoint : string }`
CloudAccess	`{ agent_name : string, provider_id : string, credentials : Object }`
AgentSession	`{ name : string, token : string, created : Date, expires : Date }`
ACLEntry	`{ type : string, scope : ACLEntryScope, grants : string[] }`
ACLEntryScope	`{ type : string agent_name : ?string, domain_name : ?string, provider_id : ?string }`
Controller Error	`{ code : number, message : string }`

Table 4.2: JSON document definitions in SDCP.

4.3 CLOUD CONTROLLER

The SDCP Cloud Controller was implemented as a stateless Node.js application. The Express.js library was used for implementing its set of web services (described in Section 4.3.1).

Agent sessions are created with a `login` operation. The agent session token is a string with a fixed length and a Base-64 character set, which is randomly generated whenever a session is created or renewed. Agent sessions live until after a fixed amount of time (five minutes by default), unless a renewal procedure is requested before it expires.

If no name and password are provided, a temporary public agent is created that will exist for as long as that agent session does. In order to distinguish them from known agents, all public agents have their names preceded by an exclamation point (`!`) wild card. Renewing a session will change the token, but the agent name is left unmodified, which means that public agents do not lose resource grants in the process.

All persistence of the component relied on MongoDB [26], an easy to use document-oriented database with horizontal scaling capabilities. By keeping all state in the database, the platform administrator can deploy several instances of the cloud controller, or redeploy malfunctioning ones, to increase service availability and resilience, without compromising the consistency of the service.

4.3.1 CLOUD CONTROLLER WEB SERVICES

The Cloud Controller is composed of four web services based on REST, accessible via HTTP and supporting a JSON based protocol. This choice of communication protocols allows for an easy service consumption from a web application, without the need for external libraries. For the access to these services from another technology (e.g. Java), many REST client libraries are available that a developer can create service access wrappers with.

Another issue taken in consideration when developing the services was the existence of same-origin restrictions on user agents. These restrictions prevent a client-side web application running from one origin from obtaining data retrieved from another origin, and also limit unsafe HTTP requests that can be automatically launched toward destinations that differ from the running application's origin. This is critical to the well functioning of SDCP, because the platform is meant to support any client web application, regardless of where it comes from. The use of Cross Origin Resource Sharing (CORS) solved this issue, by letting the services accept any application origin. In practice, this was achieved by adding the "`Access-Control-Allow-Origin: *`" HTTP header to service responses [73].

The **main** service (Table 4.3) is public and provides a set of operations for logging in, renewing / revoking a session and obtaining resource type information. Since this is a public service handling confidential data, a secure connection must be used.

The **administration** service (Table 4.4) contains operations for creating/removing agents and domains. It is private and meant to be only accessed in a strict administrative scope.

The **cloud access** service (Table 4.5) contains operations for known agents to add and remove cloud provider access credentials. It is meant to be used by an application developer before or during deployment time, and a secure connection must be used, along with a known agent login procedure prior to the usage of this service.

The **internal** service (Table 4.6) contains operations required for the well functioning of SDCP plugins, namely the remote connectors and the drivers. It provides operations for obtaining session information, registering connectors and drivers, and retrieving cloud access credentials. It is private and meant to be accessed in a scope where it is certain that only genuine, legitimate components of the platform can use this service.

REST operation	Description
POST /login?name={name} &password={password}	Login to the platform, obtaining an agent session token.
PUT /renew?token={token}	Renew an agent session.
POST /logout?token={token}	Log out, revoking the session token.
GET /rt/{rt_id}	Request for resource type information and available access point(s).

Table 4.3: Main SDCP cloud controller web service.

REST operation	Description
POST /agent?name={name}&password={password}	Register a new agent.
GET /agent	List all agent names.
DELETE /agent/{name}	Remove an agent.
POST /domain/{name}	Register a new domain.
GET /domain	List all domains.
PUT /domain/{name}/{agent_name}	Add an agent to a domain.
GET /session	List all agent sessions.

Table 4.4: Administration SDCP cloud controller web service.

REST operation	Description
GET /providers?rt_id={rt_id}	List basic information of registered cloud providers.
GET /cloudaccess?token={token}	List registered cloud provider access entries of an agent.
PUT /cloudaccess?token={token} &provider_id={provider_id}	Add a cloud access point to the agent's list of cloud provider access credentials. The credentials are sent in the request body.
DELETE /cloudaccess?token={token}	Clear all credentials.

Table 4.5: Cloud access SDCP cloud controller web service.

`REST operation`	**Description**
`GET /session?token={token}`	Get session information.
`PUT /rt?endpoint={endpoint}`	Register a remote connector, along with its resource type. The resource type descriptor is sent in the request body.
`DELETE /rt/{rt_id}/{endpoint}`	Revoke a remote connector, making it no longer usable.
`GET /domain/{domain_name}`	Get the agents registered to a given domain.
`GET /provider?rt_id={rt_id}` `&provider_id=provider_id`	List all cloud providers that provide resources of the given type.
`GET /cloudaccess?token={token}` `&rt_id={rt_id}`	List all cloud provider access data for resources of the given type that an agent can access to.
`POST /provider`	Add a cloud provider to the list of available cloud providers. The provider descriptor is sent in the request body.
`DELETE /provider/{id}`	Remove a cloud provider.

Table 4.6: Internal SDCP cloud controller web service.

4.4 REMOTE CONNECTOR SDK

The cloud controller only serves as the glue between SDCP components. The most relevant features of the platform are expressed in the remote connectors, where resources are effectively registered, identified and shared. This means that each component representing a remote connector will have to implement these topics:

1. Resource meta-data persistence and manipulation : this includes creating a federated view of all resources of the same type, and registering each cloud resources' effective origin, along with other meta-data.
2. Access Control policies based on ACLs.
3. Resource monitoring capabilities.
4. A web service and API for exposing all resource type operations.
5. The means of connecting and interfacing with the cloud provider access drivers of that resource type.

Some of these topics are so generic that a significant part of its business logic can be kept as part of the remote connector SDK, without knowing the resource type in advance. The most important pieces of resource meta-data are the URI, the ACL and the list of cloud providers where it resides (provider-specific options could also apply). Access control policies can also be previously conceived as ACL persistence and verification mechanisms. Resource monitoring often depends on a choice of metrics for that resource type, but the same resource meta-data persistence can be used. Lastly, the remote connector developer will have to define and expose a web service using a web server library of his/her own choice.

The most relevant integration of mOSAIC to the project happens in each remote connector: considering an existing implementation of a mOSAIC connector and a set of at least one driver for that cloud resource, a remote connector can be made by integrating and managing a pool of mOSAIC connectors. Most of the connectors already implemented by the consortium are made for Java mOSAIC applications. Abiding to the fact that connectors are language-specific, the remote connector SDK was implemented in Java as well, so as to let developers use the existing Java connectors.

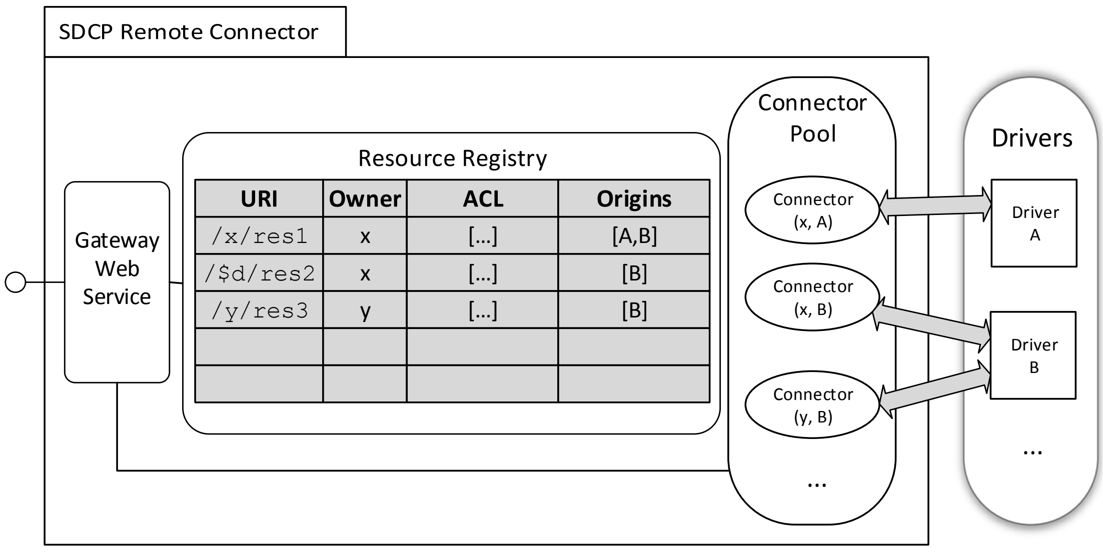

Figure 4.1: Remote connector internal structure

4.4.1 INTERNAL ARCHITECTURE

The remote connector SDK suggests an internal structure for the component, which is generic enough to be applied in many different resource types. The architecture is illustrated in Figure 4.1, and contains three main subcomponents:

- The **Gateway Web Service** exposes a web interface and interacts with the remaining subcomponents in order to translate requests to cloud provider actions, and to send messages back to the client when needed. The SDK does not provide an implementation of this module, since it highly depends on the resource type involved.
- The **Resource Registry** contains the means of access and persistence of resource meta-data. Figure 4.1 shows a simple example of the contents of each entry in the registry. More meta-data can be stored if so is desired. The owner column represents the agent that created the resource, which may not always be identifiable from the URI, in case of it residing in a domain. The first element in the URI represents the resource scope, which is either the owner agent name or a domain name, in which the dollar sign (`$`) wild card is used for indicating that the scope represents a domain name. For resource monitoring capabilities, their usage data can also be added to the registry. A resource can have more than one origin, thus why a list of cloud providers is used (`A` and `B` are examples of cloud provider IDs in the figure). An implementation of a resource registry was made available in the SDK.

- The **Connector Pool** manages a set of connector instances, indexed by agent name and cloud provider ID. If an agent's connector to a particular cloud provider does not yet exist, a new one is automatically created and registered. The connector pool class provided in the SDK library is abstract, only requiring a means of creating connectors to be implemented for each resource type.

If, for example, agent `x` requested a write operation on resource `/x/res1`, the gateway would look up the URI on the registry. Since a write operation changes the state of the resource, it must be performed with both origins **A** and **B** taken into account. This logic is hard-wired to the remote connector and would often make a replication of the same operation to each origin. Afterwards, both connectors are retrieved and the necessary operations are applied to each one of them. Once the task is complete, the server may update the resource registry and send a proper response to the client.

These procedures require interaction with the cloud controller: the client will always send an agent session token for authentication, which must be translated to an agent name by using the cloud controller's internal service. Furthermore, the driver endpoints and cloud access credentials may have to be retrieved from the same service when creating new connectors.

The web interface of the remote connector should also allow establishing new ACL entries to resources, so that other agents may be granted permission to access them (Section 3.6.1). The SDK provides data types and mechanisms for evaluating the result of an ACL (whether an operation can be invoked or not). Owners of a resource are free to perform any operation, including deleting and changing resource meta-data.

The internal web service of the cloud controller can be easily accessed from the remote connector with an interface based on Remote Procedure Call (RPC) , implemented in the SDK. A resource registry that can be used for any resource type is made available as well. Finally, an abstract connector pool was implemented with operations for retrieving connectors by origin / agent pair, which will automatically retrieve cloud access entries from the cloud controller when needed.

4.5 CLOUD PROVIDER ACCESS DRIVERS

With the integration of mOSAIC explained in Section 4.1.1, implementing a cloud provider access driver is quite similar to conceiving a mOSAIC driver. The main

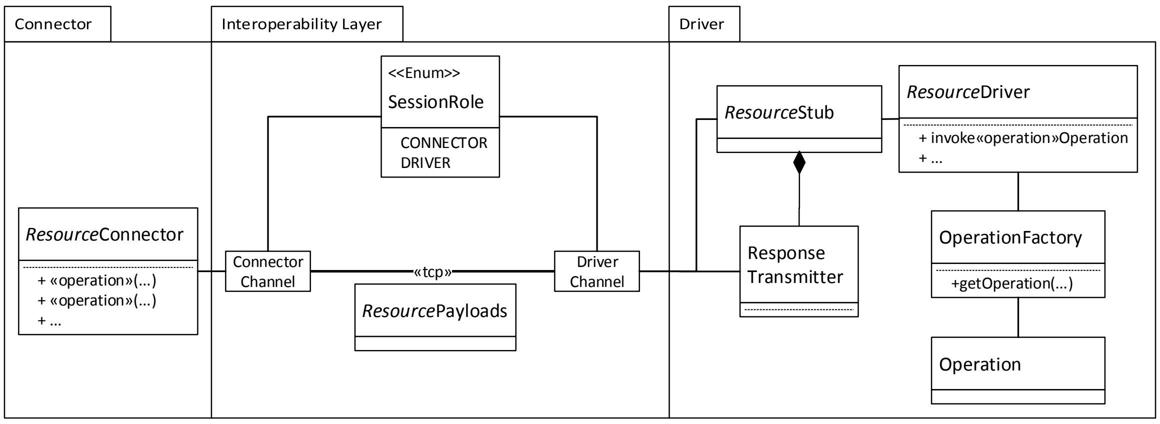

Figure 4.2: Internal structure of a mOSAIC connector-driver pair implementation

difference is that the driver must register itself to the cloud controller with the internal web service. Alternatively, the registration process can be performed separately by an integration tool or the SDCP system administrator. By sending driver information such as provider ID, resource type ID and access endpoint, the remote connectors can use this information to create mOSAIC connectors in runtime. Without the mOSAIC integration, the registration process is the same. Furthermore, other RPC solutions can be chosen for connector-driver communication.

Figure 4.2 shows an abridged generic model diagram of subcomponents that the connector and driver implementations may follow. Instances of a connector are programmatically created using factory methods and configuration objects, which will automatically establish a session to the driver. All communication messages exchanged between them are described and generated with a `Payloads` module. The *Resource Stub* component receives the messages and translates them to asynchronous operation invocations. The *Resource Driver* attends to these invocations by creating operation objects and executing them. Once the operation is completely handled and an outcome is received, the *Response Transmitter* encodes the result into a message and sends it to the corresponding connector. Although it is not required to follow this model, all of the previously mentioned capabilities must be present.

The perspective of a mOSAIC connector / driver pair is similar to that of the SDCP client and a remote connector, with the exception that the exposed interface is conceived to be fast, and only for being accessed by an existing implementation of a connector. Requests are simply converted to appropriate cloud provider invocations, disregarding the existence of SDCP agents, access control policies or more than a single origin for that resource. Cloud operations can then be executed asynchronously on demand.

4.6 SDCP RUNTIME

The client runtime is a JavaScript module for interfacing with the cloud controller's main service and eventual remote connectors. The browser's built-in AJAX capabilities were used for sending HTTP requests and retrieving responses from the cloud controller. Depending on the web browser, the `XmlHttpRequest` JavaScript object or another object available (`XDomainRequest` in the case of Internet Explorer) is used.

The module was also implemented with additional boiler-plate that makes it compatible with Node.js applications. The synchronous `require` operation will let Node.js applications retrieve the sdcp module, which functions as expected from a browser application. Besides the file exporting the module, the HTTP request code explained in the previous paragraph was also adapted to use the `http` module in case of being run in Node.js. In practice, discovering whether the script is being executed in this framework is done by verifying a variable that is always defined in web browsers (and not in Node.js).

The interface of the SDK was kept in a JavaScript module export, one of the common development patterns in this programming language where a global variable is assigned to the result of a function that returns all of the module's implementation. This pattern keeps private methods and properties invisible from the outside, while remaining visible under the closure's scope. An abridged example can be seen in Listing 4.1. All public functions were kept in the `sdcp` global variable, and are described in Table 4.7.

```
var sdcp = (function () {
  var m = {},
    agent_name = undefined, // private property
    session_token = undefined; // private property

  function renewSession() { // private method
    // ...
  }

  m.login = function (name, password, callback) { // public method
    // ...
  };

  // ...

  return m;
}());
```

Listing 4.1: Abridged example of a JavaScript module export

Function	Description
`init(url, token)`	Initialize the module. This must be called once before any subsequent operations in the module. If an agent token is given, it will be used for the following operations.
`login(name, password, callback)`	Log in to SDCP, obtaining an agent session token. If no name is given, a public agent session is created.
`logout(token, callback)`	Log out, revoking the session token.
`rt_access(rt_id, callback)`	Obtain a resource type information object.
`getSessionToken()`	Getter for the current agent session token.
`getAgentName()`	Getter for the current agent name.

Table 4.7: Client Runtime API description.

An initialization procedure is done beforehand to identify the cloud controller origin and possibly define a token of a session that was previously created by the application server, thus making server-oriented resource delegation possible, as explained in Section 3.6.1. The remaining functions contained in the module naturally reflect the API of the cloud controller's **main** service, where a REST operation is translated to an asynchronous function, containing a `callback` parameter for retrieving the results and a possible error object. Additional getter functions are available for the client to retrieve the session token and agent name.

It is also to be noted that not all operations of the main service are considered, as the interface does not contain the `renew` operation. Once logged in, the client module will automatically schedule the renewal of the agent session token, keeping the task of maintaining the agent session away from the web application's main logic.

When access to a resource type is requested, the cloud controller provides its resource type descriptor. With this document, the client can retrieve the registered client module Universal Resource Locator (URL) and the remote connector's endpoint dynamically. The client module is then transferred and dynamically loaded into the application. Alternatively, the application developer can manually add the script file to the web page. The module's API will also depend on the requested resource type. The token passing on resource access can be hidden from the web developer by letting the module rely on the SDCP module to retrieve the current session token. This also makes the SDCP runtime module required for resource type specific modules to work.

4.7 RESOURCE TYPE: SIMPLE DATABASE

As part of the proof-of-concept implementation, a new mOSAIC connector and respective remote connector were implemented for supporting a simple database kind of cloud service. Developing the entire remote connector solution involved creating many important parts:

- The simple database **interoperability layer** is shared by the connector and the driver, and contains the payload descriptors and message builders for all communication between them.
- The mOSAIC simple database **connector** is the means of creating and configuring access points to drivers of this resource type.
- Finally, at least one complete mOSAIC **driver** for this resource type.

The main difference of the new simple database connector from the existing mOSAIC key-value connector is the possibility of accessing more than a single *domain* (bucket) using the same driver connection, and allow passing multiple credentials to the driver in access time, which was previously not contemplated. In addition, this connector supports a `select` operation that may be used for running SQL-like queries in the database. Its interface is simply based on the proposal presented in the mOSAIC Deliverable 1.3, which is a copy of AWS SimpleDB API.

The remote connector relied on the remote connector SDK for managing the connector pool, only requiring the specification of how the connectors should be created, as well as the resource registry. The implemented resource type considers the same model proposed in Section 3.9. The database domain, also named a collection, used for describing a set of items, is not to be confused with the concept of SDCP domains, which represents a set of agents. Database domains exist in one or more cloud providers, and belong to one database that must be previously created in SDCP. In order to avoid database domain name collisions, the domains created at the cloud providers are effectively named with a combination of the scope, database and domain names. This is to prevent, for instance, the resource `/Agent/A/B` from colliding with resource `/Agent/C/B`, which reside in different databases in SDCP's point of view. An alternative to this approach would be generating random domains names and saving them on a translation table.

A simple REST service (Table 4.8) was created using Jersey [74], containing similar operations of the mOSAIC connector, plus a few more for manipulating the resources' meta-data, such as adding ACL entries. Operations may receive optional arguments for a tighter control of cloud resources. For instance, if a cloud provider ID is passed to the `createDomain` operation, the remote connector will create the database domain in that specific provider. If more than one cloud provider ID is given, the database domain will be evenly replicated across all providers.

`REST operation`	**Description**

`POST /db/{scope}/{db}`
`?token={token}`
	Create a new database.

`DELETE /db/{scope}/{db}`
`?token={token}`
	Delete a database.

`POST /db/{scope}/{db}/{domain}`
`?token={token}`
	Create a new database domain.

`GET /db/{scope}/{db}`
`?token={token}`
	List domains in a database.

`DELETE /db/{scope}/{db}/{domain}`
`?token={token}`
	Delete a new domain.

`PUT /db/{scope}/{db}/{domain}/{item}`
`?token={token}`
`attributes in the request body as a JSON object`
	Put attributes in an item. The item is created if it does not exist yet.

`GET /db/{scope}/{db}/{domain}/{item}`
`?token={token}&attribute={attribute...}`
	Get the attributes of an item.

`DELETE /db/{scope}/{db}/{domain}/{item}`
`?token={token}&attribute={attribute...}`
	Delete attributes in an item.

`PUT /meta/{resource_uri}?token={token}`
`&metakey={metakey}`
`ACL entry in request body as a JSON object`
	Insert meta-data in a resource.

`GET /meta/{resource_uri}`
`?token={token}`
	Insert meta-data in a resource.

`DELETE /meta/{resource_uri}?token={token}`
`&metakey={metakey}&index={index}`
	Insert meta-data in a resource.

Table 4.8: SDCP Simple Database remote connector web service description.

4.7.1 AWS SimpleDB

A new mOSAIC driver for access to Amazon SimpleDB was implemented for use in SDCP, following the same structure as the remaining drivers with only a few differences. The implemented mOSAIC drivers contemplated in-cloud services, which were to be deployed and accessed under the same cloud infrastructure. For instance, a Riak key-value storage driver would communicate with a deployed Riak database instance. In this situation, the drivers were not prepared to access the same provider with different credentials. The implemented SimpleDB driver accepts a hash map, containing the expected access key and secret key, when initializing a channel session. This also means that only one driver per cloud provider service is needed in the platform.

The driver was implemented in Java, following the model as seen in Figure 4.2. The official AWS Java SDK was used for the actual cloud provider access.

4.7.2 client module

In order to include SDCP simple database support for web applications, a client module was implemented as well. The Simple Database client follows similar conventions to the implementation of the SDCP runtime, providing an interface with a function for each available REST operation in the remote connector's web service. This module only works alongside the base SDCP client module. Like in the client runtime module, it was made both browser and Node.js compatible by containing boilerplate code for both platforms (as explained in Section 4.6).

CHAPTER 5

Results and Discussion

This chapter presents the benchmarking results of SDCP, comparing the use of the platform with direct access to cloud services. Afterwards, various discussion topics of the platform follow.

5.1 Benchmarking

The use of a middleware system to access cloud resources will introduce an overhead that can be hardly analyzed empirically. It is known that two additional components are included in order to achieve the same logical channel between the client and the cloud service: the remote connector and the cloud provider access driver. And although all communication between these components uses speed-optimized protocols, the involved payload transfers may not be negligible.

Hence, the establishment of benchmarking procedures and the analysis of results are fundamental to understanding the overhead associated with the use of SDCP for performing cloud resource operations.

5.1.1 Environment conditions and sequence of operations

The benchmark extracts the execution time values of simple database cloud operations via SDCP to AWS SimpleDB, which are compared from the same operations applied directly on SimpleDB. A client benchmarking program was created in Node.js, relying on the SDCP client runtime, the SDCP simple database client module and the official Node.js AWS SDK. An additional utility program was created for processing the resulting data into Comma Separated Values (CSV) , for further analysis and visual observation.

The complete SDCP platform, consisting of the cloud controller, the simple database remote connector and the AWS SimpleDB driver, were deployed in a local machine. The benchmarking program was executed in the same machine as the platform's. The cloud controller had a cloud access entry to the SimpleDB provider. The benchmarking program also had the means to access the credentials from a local file.

All domain names, item names, attribute names and respective values were randomly generated in the form of Universally Unique IDentifiers (UUIDs), a format often used for universal identification of resources in several different contexts [75]. The chances of a duplicate UUID in the benchmarking process are too low to be considered.

All cloud operations performed access to one cloud resource at a time (one for creating a domain, one for creating an item, and so on), always waiting for the outcome before starting the next operation. The strict sequence of operations, in both cases of testing, were as follows:

1. Create 5 database domains;
2. Insert 100 items, each with 10 attributes, in one of the domains;
3. Retrieve all attributes of each item in the domain;
4. Delete each item in the domain;
5. Delete each database domain.

5.1.2 RESULTS

The first results of several tries of the benchmark showed that the use of SDCP was faster than the direct use of the AWS SDK, which was considered controversial. With the main suspicion that the Node.js AWS SDK had a weak performance, the same benchmarking procedure was reimplemented in Java, using the same SDK as the one used by the cloud provider access driver. The execution times of this new benchmark program were later found to be more reasonable and trustworthy.

Table 5.1 shows the average and maximum execution times (in milliseconds) of each operation type performed in the benchmark, including the total benchmark execution time. Figure 5.1 shows a graph containing the process sequence in the form of accumulative execution time values after each operation performed in the benchmark.

In other words, the Y value of each line is the amount of time passed after the execution of X cloud operations. Both items are categorized into 4 modes of execution:

- **Direct (Node.js)** shows the results obtained from the Node.js benchmarking application, using direct AWS SimpleDB cloud service access.
- **Direct (Java)** shows the results obtained from running the benchmarking application made in Java, using direct AWS SimpleDB cloud service access.
- **SDCP** shows the results from running the Node.js benchmarking application via SDCP for the first time since the deployment of the simple database remote connector.
- **SDCP (warmed up)** shows the results from running the Node.js benchmarking application via SDCP a second time after the simple database remote connector was deployed.

	Direct (Node.js)		**Direct (Java)**		**SDCP**		**SDCP (warmed up)**	
	avg	max	avg	max	avg	max	avg	max
create domains	2140.2	2920	1900.2	2864	3263.6	9787	1640	1678
put items	473.8	3619	232.48	1165	304.35	1562	255.43	1234
get items	589.76	3972	184.12	1349	633.15	1266	630.91	1112
delete items	621.35	1887	200.56	242	288.4	934	211.08	334
delete domains	1940	2684	1126.2	2595	1256.8	1359	1271	1313
execution time	188901		76857		145198		124298	

Table 5.1: The benchmarking results containing average and maximum execution times of each operation type, in milliseconds.

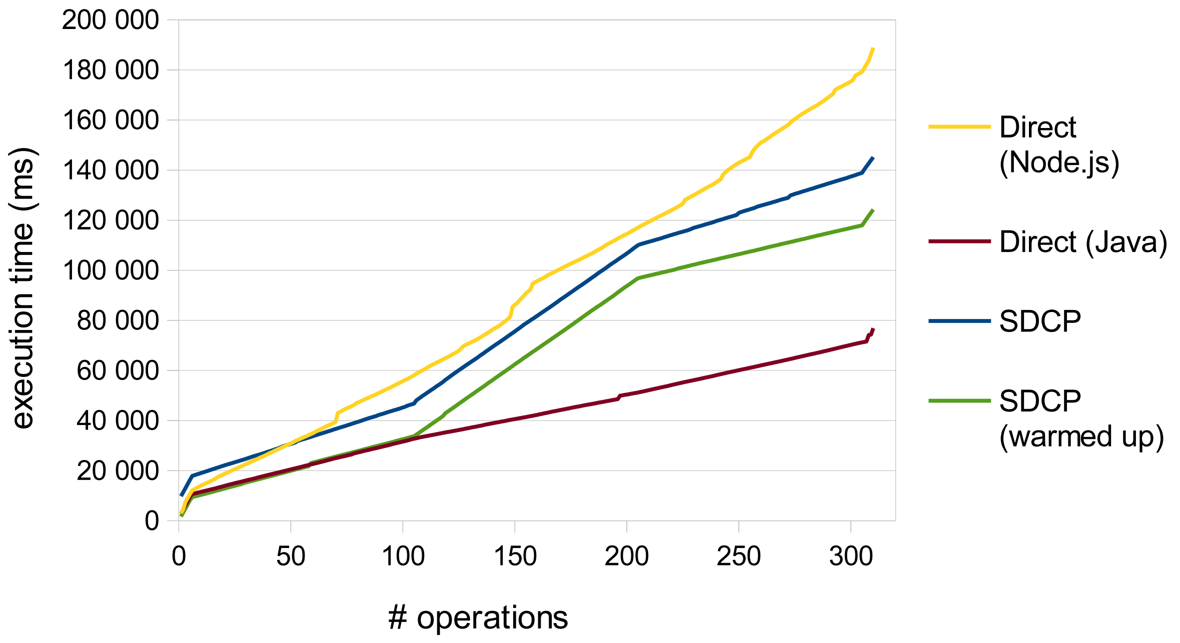

Figure 5.1: A comparison of performance between the use of SDCP for simple database operations and direct SimpleDB access.

5.1.3 OBSERVATIONS

The most costly operations in terms of execution time are the domain creation and deletion. Other operations occur much faster in the remaining cases. Database item retrieval happens to be significantly slower in SDCP, which can be observed by a steeper upwards slope in the graph's SDCP lines. This observation suggests that the `getAttributes` operation is not very well optimized in either the connector or the driver implementations involved. Along with the fact that the operation was performed on 100 items with 10 attributes, the performance hit appears more significant.

It is also observable that the first cloud operation performed via SDCP contains a significant delay. This is because the operation triggered the creation of a new connector to AWS SimpleDB for the agent, which takes a few seconds to complete. The "warmed up" SDCP results do not show this overhead, as the connector remained available for the agent at the beginning of the benchmark. The graph shows that the difference between both benchmarks of SDCP are nearly constant for every number of operations.

The slight "warm up" overhead and the slow `getAttributes` operations are the most significant bottlenecks in simple database cloud resource access. When casting these issues aside, the platform actually shows a good performance, achieving similar execution times for the remaining cloud operations. In addition, is it to be noted that this benchmark does not address the existence of an intermediate application server. Situations where the server behaves as a controlled gateway to cloud resources involve redirecting requests and responses from / to the client, whereas this already happens in SDCP.

5.2 DISCUSSION TOPICS

5.2.1 mOSAIC INTEGRATION

The choice of integrating SDCP with mOSAIC came with the premise that it would serve as a means of contributing to the project and attain results faster. However, the implementation of connectors and drivers for mOSAIC using the existing mOSAIC libraries is very complex, leading to a slowdown of the project that was not predicted beforehand. Still, the development of more connectors and drivers leads to a contribution to the mOSAIC project. Furthermore, the choice of this integration approach alone led to a better architecture model for SDCP, which was previously based on software plugins that would be attached to the cloud controller.

With the current integration of mOSAIC, SDCP relies on the connector-driver mechanisms of the system as independent components, which means that the full SDCP solution does not even need to be deployed in a proper mOSAIC environment. If the integration was to go a step further, it would involve all of the components of SDCP being manageable by the mOSAIC platform, including the cloud controller and each remote connector. Also, with mOSAIC's core system, SDCP could find existing remote connectors and retrieve their information without needing the internal web service.

5.2.2 SECURITY

Some security measures are to be taken seriously when maintaining and developing solutions using SDCP. Firstly, the cloud controller manipulates a database containing other agents' cloud provider access credentials, which are extremely confidential. Database access must be restricted to administrative tools (utilized by authorized personnel) and trusted cloud controller components.

Most web service operations of the cloud controller contain confidential data, which must not be exposed to other entities. The *main* web service involves agent authentication credentials. The *cloud access* and *internal* services are used to transfer cloud provider access credentials. The use of secure connection with Transport Layer Security (TLS) is a viable solution to the matter.

Remote connectors are active entities in the platform that require confidential information from the cloud controller, which made the internal web service a requirement to the platform. However, security measures must be taken in order to only admit system components that were legitimately and consciously deployed. Otherwise, “evil” components of the system could retrieve cloud provider credentials and misuse them (exposing them to other services or performing unintended operations with them). On the other hand, remote connectors cannot obtain SDCP agent credentials on their own, but any entity can impersonate an agent by sniffing and using the agent’s session token. A significant number of third-party components is to be expected, thus why the use of component certification methods would become crucial for the security of SDCP systems.

5.2.3 BENEFITS

The platform has been designed for creating resources among similar cloud services for use in web applications, following a provider-independent API. With the service and resource abstractions, web application developers can focus on creating and using resources at the client side to complement the application, with a complete, federate view of all resources. The automatic choice of the real location to keep the application service’s cloud resources favors the abstraction level of the API by lifting unnecessary complexity from the application developers. Manually choosing the cloud provider to use can still be done, which is important in cases where resources must be kept in safer locations, such as a private cloud.

External services can be combined, decorated and orchestrated to fulfill a service logic that may not be supported in a cloud provider. An example is the storage of encrypted data on the cloud, where the decryption key would stay in the platform, thus preventing even cloud companies or intruders from possessing the clear data.

Cloud resources from the platform can be used directly from the client program without relying on the application server, and it does not need agent credentials of its own: the application server, seen as an agent, can create and grant access to particular cloud resources on the fly, leaving part of the application’s logic to the cloud.

The platform can be extended to support more cloud providers and resource types without redeploying the entire platform. This also reduces the need to migrate the application to other cloud or multi-cloud platforms. Although still having to be implemented, new remote connectors may rely on existing mOSAIC connector community-made implementations, and its associated drivers will be immediately compatible with SDCP.

5.2.4 DRAWBACKS

The solution depends on an active middleware entity, which will naturally induce a few considerations.

A slight overhead is to be expected, which was roughly analyzed in Section 5.1.3. The performance of the platform can change with a few other aspects not considered in the benchmark. On the other hand, the access delay to some resources may be lowered by optimizing the implementations of the platform in general, along with pre-fetch and cache mechanisms not studied in this thesis.

The cloud controller will contain sensitive information, making it preferably deployed in a private cloud infrastructure. If such an infrastructure does not exist, the controller can be deployed in a public cloud, which will, on the down side, prevent the future access to private cloud services via the platform. The deployment of SDCP in a single private server would make the platform a single point of failure.

CHAPTER 6

Conclusion

SDCP provides a seamless integration of cloud services by focusing on a resource type abstraction and the use of plugins to support more services and providers to the application developer. By keeping cloud resource access over the abstraction, the platform prevents vendor lock-in by supporting cloud resource manipulation mechanisms that do not depend on the specific cloud provider.

The objectives of the framework are not limited to cloud interoperability and integration issues. The platform aims to be a practical, all-in-one framework for application development, supported in cloud resources. Although having a particular focus on web applications, the platform can be used outside of this context, simply as long as the service APIs are well defined. This is true for the Cloud Controller's web service, and so should be in each remote connector implemented. SDCP implements access control policies, which allow user agents to share resources with other agents, to well defined extents. Features that currently already existed in the mOSAIC project were lifted in order to augment the platform with the capabilities of SDCP, rather than rebuilding such mechanisms from scratch. Therefore, simple resource sharing applications can be made by developing a thin client application to the cloud controller and required remote connectors. With the concept of public agents, anonymous users may be granted direct access to resources during the application's session, even without agent credentials. The delegation of resources is deemed as a convenient pattern in a web application, where direct access from the web client to the cloud controller has been made possible. Such policies would also involve resource usage restrictions and monitoring, configurable by the web application's administrator, in a particular context.

Conclusively, SDCP is considered a practical solution to cloud service delivery, with a special focus on web applications and based on mOSAIC, a renowned project of international effort.

6.1 FUTURE WORK

This work of significant effort resulted in a well designed solution to heterogeneous cloud access, with a special focus on web applications. However, this work leaves room for a few other concepts not tackled in detail by this thesis. Some of these concepts are described below:

- Automatic service orchestration and decoration: The current architecture of the platform, following the SOA model, can take advantage of external orchestration services. Further work in SDCP can involve the creation of an automatic service orchestration module as part of the platform's architecture.
- The platform could benefit from **load balancing** and better **resource replication** techniques. A more efficient control of resources residing in multiple clouds can make resource access and migration faster and more resilient.
- Automatic interface module generation: with the combination of an IDL, a tool for generating server and client-side implementations of a remote connector interface would accelerate the development process. This includes the partial creation of a gateway service on the server side, a stub for the cloud provider access driver and a service access module on the client side, supporting JavaScript and possibly other languages for use by the server application.

References

[1] W3C, *HTML5*, 2014. [Online]. Available: http://dev.w3.org/html5/spec/.

[2] Amazon, *Amazon Web Services*, 2014. [Online]. Available: http://aws.amazon.com.

[3] Microsoft, *Windows Azure*, 2014. [Online]. Available: http://www.windowsazure.com.

[4] Rackspace, *Rackspace*, 2014. [Online]. Available: http://www.rackspace.com.

[5] *Heroku*, 2014. [Online]. Available: http://www.heroku.com.

[6] PubNub, *PubNub*, 2014. [Online]. Available: http://www.pubnub.com.

[7] L. A. Bastião Silva, C. Costa, and J. L. Oliveira, "A common API for delivering services over multi-vendor cloud resources", *J. Syst. Softw.*, vol. 86, no. 9, pp. 2309–2317, Sep. 2013, ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.04.037. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.04.037.

[8] H. He, "What is service-oriented architecture", *Publicação eletrônica em*, vol. 30, pp. 1–5, 2003. [Online]. Available: http://www.nmis.isti.cnr.it/casarosa/SIA/readings/SOA_Introduction.pdf (visited on 2014).

[9] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the web services web: an introduction to soap, wsdl, and uddi", *IEEE Internet computing*, vol. 6, no. 2, pp. 86–93, 2002.

[10] R. T. Fielding, "Architectural styles and the design of network-based software architectures", Doctoral dissertaion, University of California, Irvine, 2000, ch. 5.

[11] P. Mell and T. Grance, "The NIST definition of cloud computing", 2011.

[12] L. Youseff, M. Butrico, and D. Da Silva, "Toward a unified ontology of cloud computing", in *Grid Computing Environments Workshop, 2008. GCE '08*, 2008, pp. 1–10. DOI: 10.1109/GCE.2008.4738443.

[13] Rackspace, *Load Balancing*, 2014. [Online]. Available: http://www.rackspace.com/cloud/load-balancing.

[14] Amazon, *AWS Elastic Load Balancing*, 2014. [Online]. Available: http://aws.amazon.com/elasticloadbalancing/.

[15] K. Nuaimi, N. Mohamed, M. Nuaimi, and J. Al-Jaroodi, "A survey of load balancing in cloud computing: challenges and algorithms", in *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*, 2012, pp. 137–142. DOI: 10.1109/NCCA.2012.29.

[16] *AWS Simple Storage Service*, 2014. [Online]. Available: http://aws.amazon.com/s3/.

[17] Google, *Google AppEngine*, 2014. [Online]. Available: http://developers.google.com/appengine.

[18] Dropbox Inc., *Dropbox*, 2014. [Online]. Available: www.dropbox.com.

[19] Amazon, *AWS Relational Database Service*, 2014. [Online]. Available: http://aws.amazon.com/rds/.

[20] Microsoft, *SQL Azure*, 2014. [Online]. Available: http://www.windowsazure.com/en-us/services/sql-database/.

[21] N. Leavitt, "Will nosql databases live up to their promise?", *Computer*, vol. 43, no. 2, pp. 12–14, 2010, ISSN: 0018-9162. DOI: 10.1109/MC.2010.58.

[22] Amazon, *AWS SimpleDB*, 2014. [Online]. Available: http://aws.amazon.com/simpledb/.

[23] Basho, *Riak*. [Online]. Available: http://basho.com/riak (visited on 2014).

[24] *Redis*. [Online]. Available: http://redis.io (visited on 2014).

[25] Apache, *Cassandra*. [Online]. Available: http://cassandra.apache.org (visited on 2014).

[26] MongoBD Inc., *MongoDB*, 2013. [Online]. Available: www.mongodb.org (visited on 2014).

[27] OASIS, *AMQP*. [Online]. Available: http://www.amqp.org (visited on 2014).

[28] *AWS Simple Queue Service*, 2014. [Online]. Available: http://aws.amazon.com/sqs/.

[29] *AWS Simple Notification Service*, 2014. [Online]. Available: http://aws.amazon.com/sns/.

[30] K. Keahey, M. Tsugawa, A. Matsunaga, and J. A. B. Fortes, "Sky computing", *Internet Computing, IEEE*, vol. 13, no. 5, pp. 43–51, 2009, ISSN: 1089-7801. DOI: 10.1109/MIC.2009.94.

[31] *IEEE P2302 working group (Intercloud)*, 2014. [Online]. Available: http://grouper.ieee.org/groups/2302 (visited on 2014).

[32] *Global Inter-Cloud Technology Forum*, 2014. [Online]. Available: http://www.gictf.jp/index_e.html (visited on 2014).

[33] *Open Grid Forum*, 2014. [Online]. Available: www.ogf.org (visited on 2014).

[34] *Cloud Computing Interoperability Forum*, 2014. [Online]. Available: https://groups.google.com/forum/#!forum/cloudforum (visited on 2014).

[35] *Cloud Audit*, 2014. [Online]. Available: http://www.cloudaudit.org/CloudAudit/Home.html (visited on 2014).

[36] Open Cloud Computing Interface, *Open Cloud Computing Interface*, 2014. [Online]. Available: http://occi-wg.org (visited on 2014).

[37] OASIS, *Cloud Application Management for Platforms*, 2014. [Online]. Available: https://www.oasis-open.org/committees/camp (visited on 2014).

[38] *Unified Cloud Interface project*, 2014. [Online]. Available: https://code.google.com/p/unifiedcloud/ (visited on 2014).

[39] SNIA, *Cloud Data Management Interface*, 2014. [Online]. Available: http://www.snia.org/tech_activities/standards/curr_standards/cdmi (visited on 2014).

[40] *Cloud Standards Wiki*, 2014. [Online]. Available: http://cloud-standards.org (visited on 2014).

[41] D. J. Abadi, "Data management in the cloud: limitations and opportunities.", *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 3–12, 2009.

[42] S. Chaisiri, B.-S. Lee, and D. Niyato, “Optimal virtual machine placement across multiple cloud providers”, in *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, 2009, pp. 103–110. DOI: 10.1109/APSCC.2009.5394134.

[43] Gartner, *A CIO Primer on Cloud Services Brokerage*, 2012. [Online]. Available: https://www.gartner.com/doc/2201415 (visited on 2014).

[44] C. Gonçalves, D. Cunha, P. Neves, P. Sousa, J. P. Barraca, and D. Gomes, “Towards a cloud service broker for the meta-cloud”, in *CRC 2012: 12ª Conferência sobre Redes de Computadores*, 2013, pp. 7–13.

[45] The Apache Software Foundation, *jclouds*, 2014. [Online]. Available: http://jclouds.apache.org.

[46] T. A. S. Foundation, *libcloud*, 2014. [Online]. Available: http://libcloud.apache.org.

[47] *Simple Cloud API*, 2014. [Online]. Available: http://www.ibm.com/developerworks/library/os-simplecloud/ (visited on 2014).

[48] T. A. S. F. [website], *Deltacloud*, 2014. [Online]. Available: http://deltacloud.apache.org (visited on 2014).

[49] *mOSAIC project*, 2014. [Online]. Available: http://www.mosaic-cloud.eu.

[50] D. Petcu, C. Craciun, M. Neagul, I. Lazcanotegui, and M. Rak, “Building an interoperability API for Sky computing”, in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, 2011, pp. 405–411. DOI: 10.1109/HPCSim.2011.5999853.

[51] D. Petcu, G. Macariu, S. Panica, and C. Crăciun, “Portable cloud applications—from theory to practice”, *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1417 –1430, 2013, ISSN: 0167-739X. DOI: http://dx.doi.org/10.1016/j.future.2012.01.009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X12000210.

[52] *RESERVOIR*, 2014. [Online]. Available: http://reservoir-fp7.eu/.

[53] *OPTIMIS project*, 2014. [Online]. Available: http://optimis-project.eu.

[54] S. Nair, S. Porwal, T. Dimitrakos, A. Ferrer, J. Tordsson, T. Sharif, C. Sheridan, M. Rajarajan, and A. Khan, “Towards secure cloud bursting, brokerage and aggregation”, in *Web Services (ECOWS), 2010 IEEE 8th European Conference on*, Dec. 2010, pp. 189–196. DOI: 10.1109/ECOWS.2010.33.

[55] A. Brogi, A. Ibrahim, J. Soldani, J. Carrasco, J. Cubo, E. Pimentel, and F. D’Andria, “SeaClouds: a european project on seamless management of multi-cloud applications”, *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 1, pp. 1–4, Feb. 2014, ISSN: 0163-5948. DOI: 10.1145/2557833.2557844. [Online]. Available: http://doi.acm.org/10.1145/2557833.2557844.

[56] MuleSoft, *CloudHub*, 2014. [Online]. Available: http://www.mulesoft.com/cloudhub/ipaas-cloud-based-integration-demand.

[57] *InterCloud*, 2014. [Online]. Available: www.intercloud.com.

[58] K. Fogarty, “Cloud computing standards: too many, doing too little”, *CIO Magazine*, 2011. [Online]. Available: http://www.cio.com/article/679067/Cloud_Computing_Standards_Too_Many_Doing_Too_Little (visited on 2014).

[59] G. Lewis, “Role of standards in cloud-computing interoperability”, in *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, 2013, pp. 1652–1661. DOI: 10.1109/HICSS.2013.470.

[60] S. Subashini and V. Kavitha, “A survey on security issues in service delivery models of cloud computing”, *Journal of Network and Computer Applications*, vol. 34, no. 1, pp. 1 –11, 2011,

ISSN: 1084-8045. DOI: http://dx.doi.org/10.1016/j.jnca.2010.07.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804510001281.

[61] D. Benslimane, S. Dustdar, and A. Sheth, "Services mashups: the new generation of web applications", *Internet Computing, IEEE*, vol. 12, no. 5, pp. 13–15, 2008, ISSN: 1089-7801. DOI: 10.1109/MIC.2008.110.

[62] The mOSAIC Consortium, "D1.1 - Architectural Design of the mOSAIC's API and Platform", SUN - IEAT, Deliverable, version 02, 2011, 86 pp.

[63] ——, *D3.1 - Description of mOSAIC's API*, G. Echevarría, Ed., Deliverable, 2011.

[64] iMatrix corporation, *zeromq*. [Online]. Available: zeromq.org (visited on 2014).

[65] Google, *Protocol Buffers - Google developers*. [Online]. Available: https://developers.google.com/protocol-buffers/ (visited on 2014).

[66] *The evolution of the web*, 2014. [Online]. Available: http://www.evolutionoftheweb.com.

[67] G. Lawton, "Developing software online with platform-as-a-service technology", *Computer*, vol. 41, no. 6, pp. 13–15, 2008, ISSN: 0018-9162. DOI: 10.1109/MC.2008.185.

[68] A. Gopalakrishnan, "Cloud computing identity management", *SETLabs briefings*, vol. 7, no. 7, pp. 45–54, 2009.

[69] R. Shirey, *Internet Security Glossary, Version 2*, RFC 4949 (Informational), Internet Engineering Task Force, 2007.

[70] L. A. Bastião Silva, C. Costa, A. Silva, and J. Oliveira, "A PACS gateway to the cloud", in *Information Systems and Technologies (CISTI), 2011 6th Iberian Conference on*, 2011, pp. 1–6.

[71] P. Saint-Andre, *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*, RFC 3921 (Proposed Standard), Obsoleted by RFC 6121, Internet Engineering Task Force, Oct. 2004.

[72] I. Fette and A. Melnikov, *The WebSocket Protocol*, RFC 6455 (Proposed Standard), Internet Engineering Task Force, 2011.

[73] W. W. W. Consortium *et al.*, "Cross-origin resource sharing", *W3C Working Draft*, vol. 3, 2012.

[74] Oracle Corporation, *Jersey*. [Online]. Available: https://jersey.java.net (visited on 2014).

[75] P. Leach, M. Mealling, and R. Salz, *A Universally Unique IDentifier (UUID) URN Namespace*, RFC 4122 (Proposed Standard), Internet Engineering Task Force, 2005.