Doctoral Dissertation

Distributed cloud bursting architecture based on peer-to-peer overlay for service provisioning

Supervisor: Prof. Norihiko Yoshida

Andrii Zhygmanovskyi
Department of Information and Computer Sciences
Graduate School of Science and Engineering
Saitama University
12DM055

June 3, 2015

Contents

Abstract **1**

1 Introduction **3**

1.1 Contributions 4

1.2 Structure of this thesis 4

2 Cloud computing and emergence of the Intercloud **6**

3 Peer-to-peer overlay for service sharing and discovery **11**

3.1 Overview 11

3.2 Related work 13

3.3 Differences between content and services in peer-to-peer networks . 14

3.4 Service description 14

3.5 Service description storing 18

3.6 Service discovery 19

3.7 Comparison analysis of (a, v)-graph structure 22

4 Peer-to-peer overlay inside the cloud **24**

4.1 Overview 24

4.2 Load balancing of service registrations and service queries 25

4.3 Queuing 27

4.4 Other issues 29

4.5 Simulator implementation 29
4.6 Experiment setup 31
4.7 Simulation results 33
4.8 Example domain 36

5 Decentralized architecture for cloud bursting 39

5.1 Current state-of-the-art 39
5.2 Model overview 40
5.3 Service descriptions 42
5.4 Service queries 42
5.5 Overlay formation and scaling 43
5.6 Platform components 44
5.7 Synchronization and conflict resolving in job request/service query queues 47
5.8 Considerations for evaluation and comparative analysis 50

6 Conclusion and future work 53

6.1 Conclusion 53
6.2 Achievements 53
6.3 Benefits 54
6.4 Future work 54

Acknowledgments 56

Publications 57

Bibliography 62

List of Figures

2.1 Cloud bursting. 8

3.1 Lifecycle of client request in peer-to-peer network. 14
3.2 (a, v)-graph for service description. 16
3.3 Merged (a, v)-graph. 19
3.4 Service description storing algorithm. 20
3.5 Generic search query format. 20
3.6 Example of search query. 21
3.7 Search query routing and execution algorithm. 22

4.1 Load balancing matrix. 27
4.2 Service discovery in the cloud: client request flow. 28
4.3 Service registration success rate comparison 33
4.4 Query success rate comparison 34
4.5 Service registration success rate comparison using LBM 35
4.6 Query success rate comparison w/replication 36
4.7 Node services data example. 38

5.1 Model components. 48
5.2 Decentralized cloud bursting architecture. 49

List of Tables

3.1 Differences between content sharing and service sharing in peer-to-peer networks. 15

4.1 Database tables used for storing services at storage node (description). 30

4.2 Database tables used for storing services at storage node (structure). 31

4.3 Parameters and notations used in the simulation. 32

5.1 (a, v)-graph types summary. 41

5.2 Centralized components eliminated in proposed model. 45

5.3 Intercloud taxonomy description of the system. 52

Abstract

Cloud computing is a technology that has gained extremely wide use in last several years. Initially embraced by major IT companies such as Amazon, Apple, Microsoft, Oracle and Google, which established themselves as top players in the cloud services market, it has became common for most companies to move their infrastructure to the cloud, both public and private. If properly applied, cloud computing not only can help lower IT costs for the enterprise, but also introduces many other benefits, such as effective management of peak-load scenarios by scaling the number of instances according to the real (or predicted) demand, dealing with natural disasters and system outages by seamlessly migrating to other available cloud resources, or serving as a inexpensive platform for the startups with innovative ideas for new services.

One of the concerns for the cloud-based solutions is the fact that the components responsible for service discovery, monitoring and load-balancing still employ centralized approaches. The presence of central authority entities like service brokers is often inappropriate, since such solutions lack satisfactory scalability, present a single point of failure and lead to performance bottlenecks and network congestion. On the other hand, considering distributed nature of cloud-based architecture, it is reasonable to use distributed approach to cloud service management and discovery as well, which in turn leads to the idea of using inherently decentralized, fault tolerant and scalable peer-to-peer paradigm.

One of ideas that can alleviate already existing and potential problems of centralized cloud is hybrid cloud architecture, that consists in combining both public and private clouds to get more scalable and robust cloud solution. In this thesis, we present an approach to building a hybrid cloud system by employing cloud bursting architecture — an approach that lies in using external cloud resources when local ones are insufficient. One of major use cases for cloud bursting is managing highly skewed request distribution for deployed services, mostly characterized by peaks in load which are sudden and unpredictable, planned but not long, and often exhibit seasonal behavior.

Our cloud bursting architecture is based on the peer-to-peer infrastructure for managing services, which is also an original idea presented in this thesis. This infrastructure is specifically designed to be an effective and scalable solution for storing, sharing and discovering services, and unlike most other peer-to-peer based approaches it allows flexible search queries since all of them are executed against internal database present at each overlay node. We also present several optimizations for peer-to-peer overlay which are necessary for utilizing it in the cloud environment. Evaluation of the peer-to-peer overlay is done by performing a set of experiments on a simulator that show it can be a viable solution to use in cloud setting and specifically in hybrid cloud. We also present some considerations about further cloud architecture evaluation and analysis.

Proposed approach is designed to address various issues of interconnecting several clouds, problems of resource provisioning, service deployment and provisioning in the hybrid cloud. Scalability of the approach is attained due to flexibility of service discovery mechanism, decentralized architecture and modular approach, which allows to leverage existing components. We argue that our approach present viable solution for managing abrupt peaks in the load and keeping service provider's QoS and SLA requirements on the desired level.

Chapter 1

Introduction

Cloud computing emerged as a novel approach allowing anyone to quickly provision a large scale IT infrastructure that can be completely customized to the user needs on a pay-per-use basis. This idea was equally well embraced by small and medium-scale companies, which frequently face problems of unpredictable IT service demand and infrastructure cost, as well as large enterprises due to the ease of provisioning computing and storage resources on-demand in a simple and uniform way, often involving multi-provider and multi-domain resources, and including integration with legacy services and infrastructures. In itself, the concept of utility computing was envisioned by early pioneers of computer science, such as John McCarthy, as early as in 60's and gained wide popularity in the form of time-sharing, operated by IBM and other leading mainframe providers. Nowadays, after the idea re-emerged as cloud computing in the late 2000s, the market is filled with companies providing cloud resources, from early adopters of the technology such as Amazon, Google and Microsoft to providers of traditional hosting services and other major IT players, such as Rackspace, Oracle or IBM.

Nowadays cloud computing includes not only the provision of resources to third parties on a leased, pay-per-use basis, but also the private infrastructures maintained and utilized by individual organizations. Those constitute two most widely used cloud deployed models (as per NIST definition [1]), the former case being referred to as *public cloud* and the latter as *private cloud*. Another emerging model is represented by so called *community clouds*, which is a cloud infrastructure exclusively used by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). Community clouds exhibit a wide range of possible owner models, including one or more of the organizations in the community, a third party, or some combination of them, and may exist on or off premises.

While being a considerable step forward in building reliable and scalable software solutions, widespread usage of the cloud brings about new challenges both for software engineers and cloud adopters. Ever increasing reliance to the cloud make applications more vulnerable to cloud power outages and malicious attacks, as well as creates a need for more flexible strategy for using cloud resources, especially in cases of highly skewed request distribution for deployed services, mostly characterized by peaks in load which are sudden and unpredictable, planned but not long, and often exhibit seasonal behavior. This thesis presents an attempt to tackle these problems by leveraging another, less known by now type of the cloud called *hybrid cloud*, and more specifically, one of the approaches based on it known as *cloud bursting*. While cloud bursting successfully addresses most problems mentioned above, this thesis main contribution is to propose a way to avoid another set of problems, which will inevitably arise due to centralized nature of cloud bursting solutions proposed so far.

1.1 Contributions

Principal contributions of this thesis include a) an architecture for cloud bursting that facilitates decentralized management of cloud resources and provides end-users with fault-tolerant and reliable services in large, autonomous, and highly dynamic environments; b) discovery architecture based on structured peer-to-peer overlay network, which utilizes platform-independent attribute-value based method for describing services and an algorithm for service discovery that allows execution of range and multidimensional queries; c) application of proposed service sharing and discovery approach to streamline the execution of single jobs and entire workflows in the cloud; and c) introducing a modular framework for composing cloud bursting solutions, allowing adoption of various standards and tools.

1.2 Structure of this thesis

This thesis is divided into 6 chapters and is structured as follows:

Chapter 1 provides an introduction and outlines main contributions done in this thesis.

Chapter 2 gives more detailed overview on intercloud and cloud bursting topics.

Chapter 3 presents a detailed description of peer-to-peer overlay for service sharing and discovery: an original idea which constitutes the basis for the cloud bursting

architecture.

Chapter 4 extends idea of peer-to-peer based service provisioning to the cloud scenario by addressing cloud-specific shortcomings of the original idea, and besides provides the experimental evaluation of this approach.

Chapter 5 presents a comprehensive description of cloud bursting architecture, and provides some analysis and guidelines and consideration for its experimental evaluation.

Chapter 6 contains ideas about future research topics and other concluding remarks.

Chapter 2

Cloud computing and emergence of the Intercloud

Nowadays cloud computing is already widely adopted by multiple kinds of clients, from small IT startups to big enterprises to governments. However, wide adoption and growing reliance on the technology are at the same time among the main reasons for situations where one cloud (or cloud provider) becomes not enough. Such cases include a) outages within cloud provider premises; b) insufficient or partially ineffective geographical distribution of cloud provider resources; c) malicious attacks which can partly or fully incapacitate the cloud; d) added complexity of migrating the infrastructure from one cloud provider to another, and as a result, e) the possibility of vendor lock-in. These problems are well known and constantly addressed by both researchers and enterprises. Yet, the biggest problem that emerge from using single cloud provider is the limits in scalability, which most often manifests itself in form of poor handling of abrupt fluctuations in the load, when permanent scaling out is not economically reasonable, and at the same time peaks in the load must be handled as prompt as possible. Flash crowds can be named as a prominent example [2].

Recently, there is an increasing research effort concerning concomitant use of two or more cloud services to minimize the risk of widespread data loss or downtime due to a failure in a cloud computing environment. Although the terminology is not quite fixed yet due to the freshness of the research domain, most authoritative sources [3–5] tend to use the term *inter-cloud computing* which is formally defined as "a cloud model that, for the purpose of guaranteeing service quality, such as the performance and availability of each service, allows on-demand reassignment of resources and transfer of workload through a interworking of cloud systems of different cloud providers based on coordination of each consumers requirements for

service quality with each providers SLA and use of standard interfaces.' [6]. This model, in turn, can be further divided into two more specific categories — *cloud federation* and *multi-cloud*. According to most definitions, cloud federation is achieved when a set of cloud providers voluntarily interconnect their infrastructures to allow sharing of resources among each other, while multi-cloud denotes the usage of multiple, independent clouds by a client or a service. Unlike federation, multi-cloud environment does not imply volunteer interconnection and sharing of providers infrastructures. One special case of multi-cloud, named *hybrid-cloud* is of particular interest. It is formally defined as "composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability" [1], and more informally as a composition of two or more different cloud infrastructures, for example, a private and a public cloud. This model currently gains popularity among cloud solutions. According to the recent report by Enterprise Strategy Group [7], share of hybrid cloud model usage among enterprises rose to 14% in 2014, while strictly on-premises model occupies only 2%. Current most popular use cases for hybrid cloud are as follows:

1. **Separation of workloads:** Client can deploy an on-premises private cloud to host sensitive or critical workloads, but use a third-party public cloud provider, such as Google Compute Engine, to host less critical resources, such as test and development workloads.

2. **Big data processing:** Company could use hybrid cloud storage to retain its accumulated business, sales, test and other data, and then run analytical queries in the public cloud, which can scale to support demanding distributed computing tasks.

3. **Managing peaks in load:** Applicable to systems that experiences significant demand spikes that are, however, rare, unpredictable or seasonal. Services could run in private cloud, but lease additional external cloud resources when computing demands significantly increase.

The last scenario is often referred to as *cloud bursting*. Though previously the term cloud bursting was used to describe an extension of grids and clusters by the means of clouds [8], currently its definition is extended to the private clouds. One of the characteristics that makes cloud bursting scenario increasingly appealing for enterprises is that it offers a reasonable alternative to leasing versus buying and combines the scalability and ubiquity of public cloud with high security and total control of the private one. Illustration of how cloud bursting works in a nutshell is shown in Figure 2.1

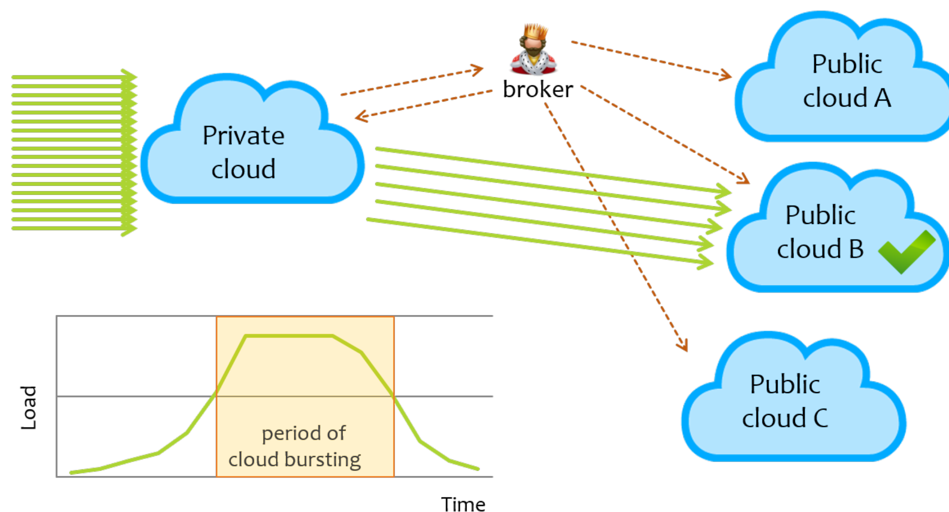

Figure 2.1: Cloud bursting.

In general, researchers analyze cloud bursting from two perspectives: **service provider perspective**, which concerns usage of public cloud resources when private cloud resources are insufficient, and **cloud provider perspective**, which concerns usage of other cloud provider resources by a cloud provider when its own are insufficient. Since we consider the latter as the special case of more general research on interclouds, in this thesis we will use term "cloud bursting" only in the former meaning.

There are the following concerns that can be identified for successfully deploying cloud bursting solutions:

- **Security and isolation** The pivotal point here is that enterprises are hesitant to trust a third party service provider to host applications or components, therefore service providers must prove that they meet compliance and audit requirements as specified by the enterprise customer. This particularly includes satisfactory level of security for data in transit, in use and at rest, the implementation of access control mechanisms, and establishing bilateral trust between cloud data centers.

- **Network performance and data architecture**. Since the connectivity between the clouds look like a "horizontal hourglass", even given the best WAN networks and WAN performance optimization, the throughput and latency usually have significant impact on application performance. Besides, the connectivity of choice for cloud bursting data centers is almost invariably

a VPN connection with encryption, which further increases the latency.

- **Data locality and compliance** The major concern in this category is the lack of transparency as to the exact physical location of clients' data, especially when it is subject to legal restrictions. Furthermore, even when the migration of workloads to a public cloud does not involve moving the data, possibility of failing to meet legal and regulatory requirements for handling sensitive information, such as guaranteed shutting down of previously instantiated virtual machines and secure wiping of attached temporary storage, cannot be ruled out completely.

- **Management and federation**. This group of concerns arguably present the widest area with the most ongoing research, including the one presented in this thesis. It comprises resource allocation and management, their optimization and life cycle management between a virtualized data center and the overflow capacity in a remote data center, as well as conceiving effective architecture for provisioning and orchestrating services, that are provided upon those resources. Given the difficulties that currently emerge when trying to move workloads quickly and easily across different cloud service providers due to the various manifestations of vendor lock-in, many researchers argue that first and foremost there is a need of common API standardization and providing third-party tools that are independent from software vendors.

However, as for inherently distributed system, cloud bursting potential becomes limited when such vital tasks as resource management and allocation, service provisioning and life cycle management are implemented in a traditional centralized way, usually utilizing some kind of brokering architecture, which introduces typical issues such as existing of single point of failure, performance bottlenecks, network congestion and synchronizing problems. This leads to the need of making at least part of the cloud bursting solution decentralized, therefore increasing its robustness, scalability and fault tolerance. Since our research interest mostly lies in the area of management and federation, architecture proposed in this thesis is designed to deal with the following problems: a) general approach to interconnecting several clouds; b) resource provisioning in the hybrid cloud; c) service deployment and provisioning in the hybrid cloud. This corresponds, by and large, to problems addressed in other proposed solutions related to building cloud bursting architecture. As we already mentioned, virtually all the solutions that we are aware of introduce certain centralized component that play the role of a broker, which usually combine multiple responsibilities, such as a) acting a marketplace where clouds can sell or advertise resources [6]; b) mapping user requests to cloud resources [9]; c) maintains the registry of collaborating clouds' services [10], etc. We argue that existence of such

component makes the system vulnerable to all kind of intrinsic failures to which centralized systems are susceptible to, as already described in the introduction.

Chapter 3

Peer-to-peer overlay for service sharing and discovery

In this chapter, we describe a service discovery architecture based on structured peer-to-peer overlay network, which utilizes platform-independent attribute-value based method for describing services and an algorithm for service discovery that allows execution of range and multidimensional queries. Among ideas that inspired us to propose this system, we can name PWSD (Peer-to-Peer based Web service discovery) architecture, presented in [11] and Intentional Name System, described in detail in [12]. We propose a lightweight approach that could be based on any DHT overlay, uses attribute-value based service description without resorting to complex data description frameworks (e.g. Semantic Web) and is designed with modular approach in mind. While being inherently a framework for building reliable and scalable network for providing, discovering and using services, the approach presented in this thesis is designed to be applied to the problem of efficient service provisioning in cloud-based solutions, that is, for building an overlay network that makes cloud-based services resilient, scalable and managed in a distributed manner.

3.1 Overview

The idea of building service sharing and discovery network based on peer-to-peer overlay itself is not innovative, since it allows avoiding many problems that arise in centralized scenarios, such as single point of failure, poor scalability or lack of robustness. Nevertheless, even nowadays most of peer-to-peer based systems deal with simple content sharing, which is fundamentally different from functionality of sharing services. Still, there is a range of problems in common that are present in

both cases, most important of them being appropriate descriptions of items shared (content or services), and creating flexible and efficient search functionality that provide results as relevant to the users' criteria as possible.

The pivotal point of the system is an original platform-independent format of service descriptions. This approach was first introduced in paper [13] with extended analysis of the related research, however, at that time we did not yet consider its possible application to the cloud environment. As was noted earlier, it is based on Intentional Name System naming approach described in [12] and utilizes abstract graph-based attribute-value description mechanism of services which is called (a, v)-*graph*. The underlying serialization format of the graph is actually not important, since it is not the part of the framework itself. We also propose several ways for building description graph, including utilization of existing descriptions, automatic description building and manual description input. While (a, v)-graph uses the attribute-value structure, it serves as basic format for the service description itself, which, among other ways, could be obtained by transforming corresponding service descriptions, including XML-based ones. Besides, (a, v)-graph structure contains only information meaningful for the service discovery, and also can be easily extended to contain the value type specification, so that search queries could be executed in a type-aware way. Also it is important to note that service owner and binding information are included in the (a, v)-graph as a special vertex[1] which is essential to the execution of discovered services. The main point which makes (a, v)-graph approach useful in distributed services storing and discovering is that the hash is computed only for attribute vertex and corresponding subgraph of the (a, v)-graph is copied to the responsible overlay node according to obtained hash value. This way the structure stored at the responsible node is still an (a, v)-graph, which allows for the queries to be more flexible, making possible using range and multidimensional queries in the application. Flexibility of the queries is also ensured by the fact that actual mechanism of (a, v)-graph storage is not defined, so different implementations can choose the best one for specific needs. While we have chosen to implement our idea using Chord DHT [14], the approach for storing and discovering service descriptions proposed in this thesis is actually overlay-independent. That is, the algorithms of storing, removing, updating and locating the services are defined in the layer more abstract than DHT one, and thus deal only with abstract notions such as "responsible node". This way it is possible to have multiple layers of overlays for service storage (for instance, to increase reliability or distribute the load), and those overlays, in fact, do not have to be based on the same DHT. Finally, we present abstract format of querying the distributed database of service descriptions which makes executing range and multidimensional queries possible.

[1]We use the term "vertex" meaning "node of a graph" in situations when there might be a confusion between terms "overlay node" or "cloud node".

The approach described above bears some similarities with distributed scalable content discovery system, proposed in [15] in that both use attribute-value based content registration and discovery mechanism (influenced by [12]), and propose similar mechanism of storing multidimensional data in the peer-to-peer overlay. However, the approach for resolving range queries in the paper mentioned above is based on Range Search Tree and due to the complexity of this structure includes various optimization and adaptation protocols, making the actual implementation of the approach difficult and error-prone.

3.2 Related work

Despite the fact that peer-to-peer approach is successfully used for content storage and management for about 15 years, service management systems based on it are still quite sparse and, as a general rule, they either remain within academia or even do not evolve further than proof-of-concept stage. The earliest successful peer-to-peer based service discovery networks include Hypercube [16] and Speed-R. Traditionally, many proposed solutions are based on Semantic Web approach for describing services. For instance, in the approach shown in [17], services are described using DAML-S, while service publishing and discovery mechanism is based on JXTA technology. Similar approach is used in [18], but Gnutella peer-to-peer overlay is used instead of JXTA. Generally speaking, all peer-to-peer based solutions for service management and discovery can be divided in two groups, depending on approach for building an overlay (structured or unstructured). While unstructured overlays are easy to implement and exhibit many interesting characteristics like small-world phenomenon, they suffer from significant drawbacks such as inefficient routing, unnecessary network congestion or inability to locate rare objects. Moreover, if we take into account the fact that cloud usually consists of a homogeneous set of hardware and software resources in a single administrative domain, we can argue that using DHT overlay presents an efficient approach due to its ability to adapt to dynamic system expansion or contraction, high scalability and autonomy features. However, since DHTs originally lack the ability to execute queries other than key-based lookup, their application in scenarios that require range and multidimensional queries was always a challenging task, leading to various approaches to storing and locating objects in DHT, such as locality preserving mapping based on Space Filling Curves in [19], sliding window partition method in [20] or tree-based solutions such as MX-CIF quad tree in [21] to name a few.

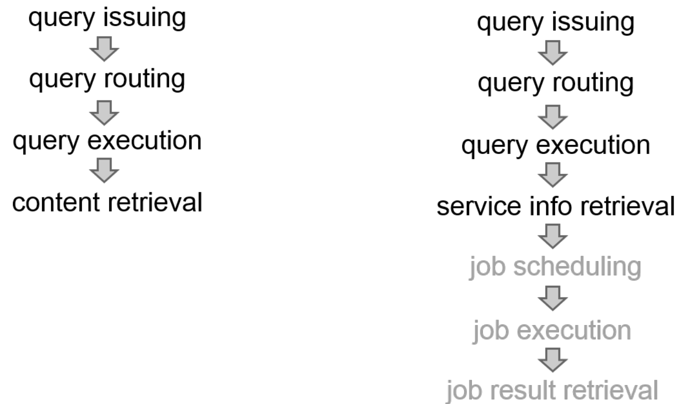

Figure 3.1: Lifecycle of client request in peer-to-peer network.

3.3 Differences between content and services in peer-to-peer networks

In this section I present a brief outline of the differences between content sharing and service sharing in peer-to-peer networks. In my opinion, proper understanding of these differences is a key to designing efficient and scalable service sharing and discovery architecture. First, let us look on the process of typical client request that happens in the network. While content request always ends with discovering and passing the desired content to the requester (or its absence), service request's lifecycle is considerably more complex in that discovering the service ends only the first, preliminary phase of the request, which is then must be continued with job scheduling, service execution and returning the result to the client. Besides, all these steps can be considerably distant in time. You can see two flows compared in Figure 3.1. Other differences are summarized in Table 3.1.

3.4 Service description

In this and next several sections we show how services are described, stored and discovered in the proposed peer-to-peer based architecture. Each service in the network is described using attribute-value format, forming so called (a, v)-*pairs*, where attribute stands for the arbitrary property of a service. While attributes can be only strings, value types can be of any type which is queryable, serializable and can be efficiently stored by the underlying DHT storage mechanism. All (a, v)-pairs form a connected graph, called (a, v)-*graph*, which always includes one extra vertex,

Table 3.1: Differences between content sharing and service sharing in peer-to-peer networks.

	Content sharing	Service sharing
Nature of data	various types of files	services description and services endpoint description
Data volatility	content changes frequently	services are stable and usually provided for a long period time
Data volume	• usually there is a lot of files being shared • content may be (and usually is) very large (GB, TB)	• usually peer exposes only up to several services • service descriptions are text-based and therefore not large in size
Metadata	• exact metadata depends on type of file shared (MP3, JPEG, PSD, ZIP etc) • usually not human readable, most often in binary format (unless input by the user himself or extracted from the file using some transformation) • may be absent at all or be present only as the name of the file • owner is not particularly concerned	• done by the service owner using some well-known format (WSDL, OWL-S) • more accurate and complete service description makes it more discoverable and usable, so owner is highly concerned • text-based
Peer identity	• most file sharing peer-to-peer networks users prefer to remain anonymous • there is a strong emphasis on a means to protect user privacy and anonymity	services are provided by some well-known identity

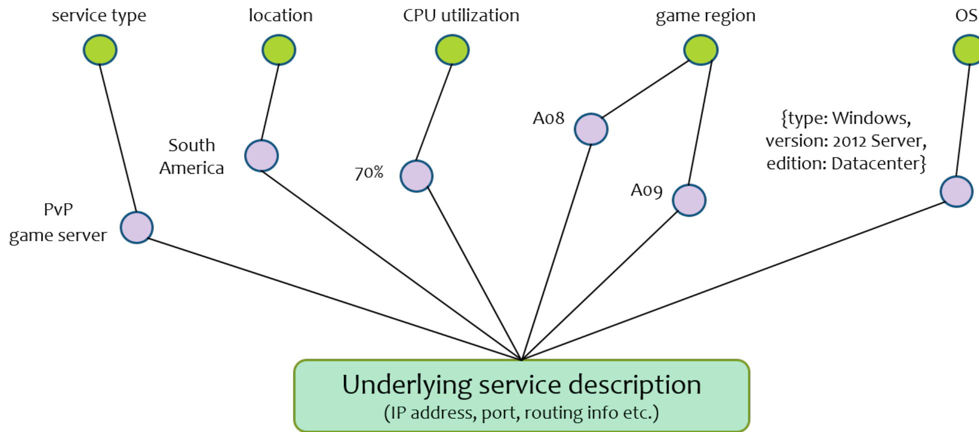

Figure 3.2: (a, v)-graph for service description.

that represents routing and binding information for the service itself. In real life situations it is not uncommon when one node provides access to several services, and each service is described using (a, v)-graph. Example of (a, v)-graph is shown in **Figure 3.2**. The service is described using the following attributes: *service type*, *location*, *CPU utilization*, *game region* and *OS*. The rest of graph vertices consist of values of attributes mentioned above and one more special vertex which contains low-level service info, such as IP address, port and network protocol.

From the viewpoint of the multidimensional objects representation theory, indexing methods can be divided into two broad categories [22]: *relatively low-dimensional data*, which usually arises in domains such as geographic information systems, spatial databases, solid modeling, computer vision, computational geometry, and robotics; and *high-dimensional data*, which is seen as a direct result of trying to describe objects via a collection of features (also known as feature vector). In our case, the feature vector is represented by a set of (a, v)-pairs. The queries, which are to be supported in the application that uses such kind of data, usually fall into the following categories [22]:

1. Finding objects having particular feature values (*point queries*).
2. Finding objects whose feature values fall within a given range or where the distance from some query object falls into a certain range (*range queries*)
3. Finding objects whose features have values similar to those of a given query object or set of query objects (*nearest neighbor queries*). In order to reduce the complexity of the search process, the precision of the required similarity can be an approximation (*approximate nearest neighbor queries*).

4. Finding pairs of objects from the same set or different sets sufficiently similar to each other (*all-closest-pairs queries*). This is also a variant of a more general query commonly known as a *spatial join query.*

The algorithm for service discovery ensures that all query types mentioned above are possible to execute and that the efficiency of answering to such queries, in fact, depends only on the efficiency of the node's internal database, therefore being independent from the service discovery approach itself.

One of the most challenging issues for any new approach or framework is to provide compatibility with existing technologies, which are already widely used. It is important to note once more that (a, v)-graph approach is just an abstract model of service description, therefore it should be seen as an output of some model transformation function. Possible ways of obtaining the resulting (a, v)-graph model are at least as follows:

Manual Input: the simplest case where user enters all service description information manually, usually with some kind of GUI, though batch information input (for instance, by uploading the file with multiple attribute-value pairs) is also possible. Regardless of how the data is put into the system, it always can be processed and transformed to the underlying (a, v)-graph serialization format.

Transformation of Existing Service Descriptions: to address the issue of compatibility, the system must have a way to obtain (a, v)-based descriptions from existing ones. In our case this is achieved by applying necessary model transformations, exact nature of which depends on the representation of existing models. But since in most cases existing services are described using XML-based industry standards like WSDL or OWL-S, Extensible Stylesheet Language Transformations (XSLT) language seems to be the most appropriate choice for model transformation in this case, due to its high expressive power and ability to output virtually any kind of data format using XML data as input. It is important to note that in most cases existing service description need to be transformed to (a, v)-based one only partly, since low-level details of a service (like protocol name, input parameters enumeration, IP address etc) are generally not searched upon. However, those low-level details can still be present in (a, v)-graph in the service description vertex to facilitate the process of binding and routing when the service is actually used.

Automatic Augmentation: among service description properties there are often ones that are highly important as a search criteria but normally are neither

supposed to be input by humans nor present in traditional static service descriptions. Examples of such properties are current geographical location of the service, status of the job queue, various QoS data (like performance or latency) and sharing network data (like current node's reputation). All those data can be obtained automatically using various means, including automatic location detection, internal monitoring functionality or network monitoring and QoS protocols. However, storing and querying such kind of values correctly requires additional efforts because of their dynamic nature. We will elaborate more on that in Section 4.4.

3.5 Service description storing

In the original set of experiments we conducted to support this approach, Chord DHT peer-to-peer overlay was used to store service descriptions in a distributed manner. Again, since this idea does not depend on actual DHT implementation, it can be changed anytime (as we did later by switching to Kademlia DHT). As usual, in order to store content in DHT we need to define what will act as a key and what will be stored at the node. In our system, we apply hash function to each attribute name and decide the node which is responsible for this attribute — in case of Chord DHT it is successor of a key. In terms of our approach, this node is called *responsible node* for the attribute and therefore will store subgraphs of a form *[attribute, value, service description]*, that is, subgraphs based on given attribute, of all (a, v)-graphs in the network. In the result, we obtain a structure called *merged* (a, v)*-graph* in each node that is responsible at least for one attribute. Example of merged (a, v)-graph is shown in **Figure 3.3**. Here, the node responsible for attributes A_1, A_2 and A_3 holds merged (a, v)-graph which consists of those attributes, all their values found in the network and respective service descriptions, which include routing information about service owner node.

Next, we present formalized version of service description storing algorithm. Each node P in the overlay owns two graphs, namely, $P.OS$ — graph for the services it owns, and $P.SS$ — graph for the services from other nodes it stores. The formal definition of both graphs is as follows: $P.OS = (V_V \cup V_A \cup \{sd\}, E_{A,V} \cup E_{V,SD})$ and $P.SS = (V_V \cup V_A \cup SD, E_{A,V} \cup E_{V,SD})$, where V_V — set of vertices that correspond to the attributes of the service, V_A — set of vertices that correspond to the values of the attributes, sd — vertex which contains the low-level service description (including information about owner node), SD — set of sd vertices, $E_{A,V}$ — set of edges $(v_A, v_V)|v_A \in V_A, v_V \in V_V$ and $E_{V,SD}$ — set of edges $(v_V, sd^*)|v_V \in V_V, sd^* \in SD \vee sd^* = sd$. The only difference between formal definitions of $P.OS$ and $P.SS$ given above is the number of service definitions, included as vertices in the graph:

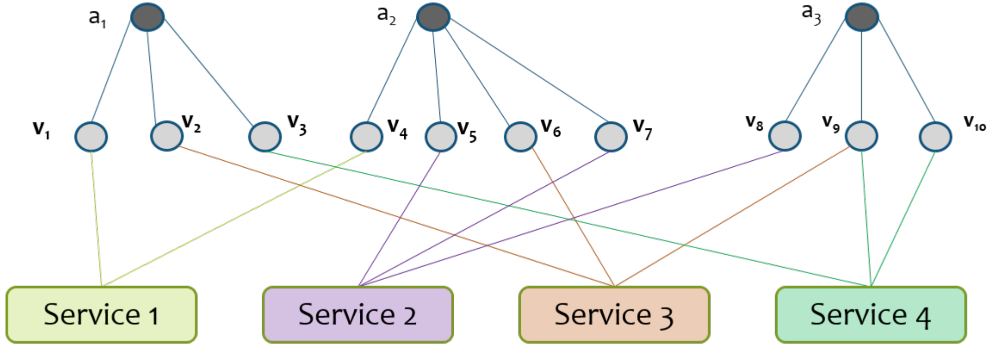

Figure 3.3: Merged (a, v)-graph.

in $P.SS$ there potentially will be multiple service definitions corresponding to owner nodes of each stored service, while $P.OS$ contains one and only one service definition vertex, which contains information of a node that owns $P.OS$ itself. Then, we assume that for each node in the overlay the following two functions are defined: ***h*** — hash function used to build an overlay, and ***find*** — function that returns the node from the overlay by hash value. The pseudocode for service description storing algorithm is shown in **Figure 3.4**. First, we evaluate hash function against each attribute name in the service description. Having found the node, which is responsible for storing that value, we send the graph which is comprised of the attribute, its values and service description vertex to responsible node. When the graph is received, we join it to merged (a, v)-graph (if it is already present at the node).

It is not essential to the architecture of the system how exactly merged (a, v)-graphs are stored at the node, but storage mechanism should be chosen in a way that makes possible answering range and multidimensional queries. Therefore, the most appropriate choices for storage mechanism are relational databases or document-oriented databases, although both of them have their own advantages and drawbacks.

3.6 Service discovery

Among one of the most significant drawbacks of DHT-based peer-to-peer overlays is that the principles of content storage and its association with a key usually allows processing only exact match queries when searching. There are some elaborate approaches addressing this issue, but they usually offer only wildcard matching level queries. The approach we propose in this thesis is based on the way service

```
foreach service s from P.OS
  foreach v_A ∈ s.V_A
    H := h(v_A);
    P := find(H);
    if not exists v*_A ∈ P.SS.V_A where v*_A = v_A
      P.SS.V_A := P.SS.V_A ∪ {v_A};
    end if
    sd* := get sd* from P.SS.SD
      where sd* = s.sd;
    if sd* is NULL
      P.SS.SD := P.SS.SD ∪ {s.sd};
      sd* := s.sd;
    end if
    V := {v_V ∈ s.V_V | (v_A, v_V) ∈ s.E};
    foreach v ∈ V
      P.SS.V_V := P.SS.V_V ∪ {v};
      P.SS.E := P.SS.E ∪ {(v_A, v)};
      P.SS.E := P.SS.E ∪ {(v, sd*)};
    end foreach
  end foreach
end foreach
```

Figure 3.4: Service description storing algorithm.

descriptions are stored in the overlay network, that is, using merged (a, v)-graph.

$$op_1(A_1) \quad LOP_1 \quad op_2(A_2) \quad LOP_2 \ \ldots \ LOP_{n-1} \quad op_n(A_m)$$

Figure 3.5: Generic search query format.

Firstly, we assume that each search query in the system is submitted in the form, shown in **Figure 3.5** or can be represented as such. Here we let $op_i(A_i)$ be operators $=, \neq, contains, >, <$ etc., defined on the sequence of attributes A_i as parameters, A_i — sequence of attributes $[a_1, a_2, \ldots, a_n]$ (where n is arity of the operator op_i) and LOP_i denote logic operators OR or AND. This format itself is very generic, so we argue that it represents most of meaningful search queries submitted in peer-to-peer networks. You can see a concrete example of the query in **Figure 3.6**. Here, we search for a game server, running on the machine with at least two processor cores, having two possible operating systems installed and located somewhere in South America.

The algorithm of search query routing and execution is formalized below. Note that the algorithm actually does not require any extensions for the underlying DHT

$$
\begin{array}{c}
servicetype =' GameServer' \\
AND \\
(OS =' Windows\ 2008\ Server' \quad OR \quad OS =' Windows\ 2012\ Server') \\
AND \\
processorcores \geq 2 \quad AND \quad location =' South\ America'
\end{array}
$$

Figure 3.6: Example of search query.

algorithm. The query issued by the node in the form shown in **Figure 3.5** can be represented as a graph $QU = (Q \cup LOP, E)$, where $Q = \{q_i = (a_i, op_i) | i = 1 \dots n\}$ — the set of query terms, $LOP = \{lop_i | i = 1 \dots n\}$ — the set of logical operators, and $E = \bigcup_{i=1}^{n} \{\{(q_i, lop_i)\} \cup \{(lop_i, q_{i+1})\}\}$. In addition to node functions ***h*** and ***find*** introduced in the Section 3.5, we assume that each node have function ***evaluate***, which returns the result of evaluating a query term again internal database of the node. Similarly, we define graph $RES = (R \cup LOP, E)$, where $R = \{r_i | i = 1 \dots m\}$ — initially empty set of results obtained from nodes after evaluating a query term, $LOP = \{lop_i | i = 1 \dots n\}$ — the set of logical operators, and $E = \bigcup_{i=1}^{n} \{\{(r_i, lop_i)\} \cup \{(lop_i, r_{i+1})\}\}$. Also we define the set of pairs $C = \{(OR, \cap), (AND, \cup)\}$ which contains the one-to-one correspondence between logical operators and set-theoretical operations. Pseudocode for the algorithm is shown in **Figure 3.7**. To put it simply, the process of search query routing and executing basically consists of the following steps:

1. Query is split by logical operators, forming a set of attribute expressions.
2. Attribute expressions are grouped according to the hash function value of each attribute present in the group, thus forming attribute groups.
3. Each attribute group is sent to the node, responsible for it in the peer-to-peer overlay.
4. Each attribute expression in the attribute group is evaluated against the data in merged (a, v)-graph stored at responsible node and the result (i.e. list of service descriptions) is propagated back to the originating node.
5. Originating node merges all query results from responsible nodes according to the logical operators and forms final result.

```
foreach q ∈ QU.Q
   H := h(q.a);
   P := find(H);
   R := P.evaluate(q);
   RES.R := RES.R ∪ {R}  for  corresponding  lop ∈ LOP;
end foreach
return r_1  C[lop_1]  r_2  C[lop_2] ... C[lop_{n-1}]  r_n
     where r_i ∈ RES.R, lop_i ∈ RES.LOP;
```

Figure 3.7: Search query routing and execution algorithm.

3.7 Comparison analysis of (a, v)-graph structure

Despite the fact that NVTree used in [11] and (a, v)-graph have much in common, it is the differences that make (a, v)-graph structure more flexible. In order to get a better view on advantages of the approach proposed in this thesis let us outline those differences more clearly with NVTree as an example. As can be seen from the way of building the NVTree, it is basically a tree representation of a XML-based (actually, WSDL-based) service description which is further split into atomic attribute-value pairs, where value nodes contain typed data (in case type information is not available, the value is assumed to be of string type). Besides, the information present in NVTree comprises all data found in the original service description, including the routing, binding and service owner information about the service itself. On the other hand, while (a, v)-graph uses the same attribute-value structure, it serves as basic format for the service description itself, which, among other ways, could be obtained by transforming corresponding service descriptions, including XML-based ones. Besides, (a, v)-graph structure contains only information that is meaningful for the service discovery, and also can be easily extended to contain the value type specification, so search queries could be executed in a type-aware way. Also it is important to note that service owner and binding information are included in the (a, v)-graph as a special node which is essential to the further execution of discovered services.

Another difference concerns the point of how those attribute-value pairs are used in the hashing process. The approach in [11] is as follows: attribute and value string are concatenated and passed as an argument to the hash function which determines the place where this piece of information is stored in the overlay network. The approach in the (a, v)-graph is similar but instead of hashing attribute and value altogether, the hash is computed only for attribute node and corresponding subgraph of the (a, v)-graph is copied to the responsible node according to obtained hash value. This way the structure stored at the responsible is still an (a, v)-graph,

which allows for the queries to be much more flexible, which includes fuzzy and range queries. Flexibility of the queries is also stipulated by the fact that actual mechanism of (a, v)-graph storage is not defined, so different implementations can choose the best one for the specific needs.

The next step for services discoverability after descriptions is the way how they are stored in a distributed manner in peer-to-peer overlay network. Similarly to [11] we chose Chord overlay for this purpose, but while the authors of [11] decided to extend Chord DHT algorithm, scheme for storing and discovering service descriptions proposed in this article is overlay-agnostic. That is, the algorithms of storing, removing, updating and locating the services are defined in the layer completely disconnected from the DHT layer and deal with only with abstract notions like "responsible node". This way it is possible to have multiple layers of overlays for service storage (for instance, to increase reliability or distribute the load), and those overlays, in fact, do not have to be the same. This approach will be described in more detail in the next sections. Finally, we present abstract format of querying the distributed database of service descriptions which makes possible wide range of conditions including fuzzy conditions, range queries and complex values lookup.

Chapter 4

Peer-to-peer overlay inside the cloud

4.1 Overview

In this chapter we outline a service provisioning approach which utilizes service sharing and discovery mechanism described in Chapter 3. Generally, the process of deploying application services on clouds is known as *cloud provisioning* and consists of three primary steps [23]

1. *Virtual machine provisioning:* instantiation of one or more VMs that match the specific hardware characteristics and software requirements of the services to be hosted

2. *Resource provisioning:* mapping and scheduling of VMs onto distributed physical cloud servers within a cloud

3. *Service provisioning:* deployment of specialized application services within VMs and mapping of end-user's requests to these services

Although all main players in the cloud computing market provide solutions that control scalability and reliability of cloud instances, they all rely on traditional centralized model of operation, thus becoming subject to usual problems of this approach, such as network congestion, performance bottlenecks and existence of the single point of failure. At the same time, in order to deliver expected Quality of Service to customers, minimize maintenance costs and ensure that given cloud solution is robust and reliable, large-scale cloud systems need scalable and reliable

service provisioning architecture, which can be attained only if it is built in a decentralized manner, eliminating all disadvantages of centralized approach. Specifically, two major advantages of using peer-to-peer overlay in cloud service provisioning can be named as follows:

1. Absence of centralized entity that answers and routes client requests (sometimes called "cloud broker") and potentially presents a single point of failure
2. Layer of abstraction that eliminates differences between cloud providers and instances

Since (a, v)-graph mechanism was originally designed as a generic approach for storing arbitrary services in a non-specialized peer-to-peer overlay, adapting it to cloud-based solution presents several challenges, most important of them are:

- Proper technique for balanced storing attribute-value pairs that are naturally skewed in the cloud scenario, since most nodes expose a set of standard attributes, such as operating system, available memory, processor architecture etc.
- Algorithm for load balancing, i.e. an act of uniformly distributing workload across one or more service instances in order to achieve performance targets such as maximize resource utilization, maximize throughput, minimize response time, minimize cost and maximize revenue [23]
- Scalable and robust queuing approach (such as publish/subscribe) for requests that cannot be satisfied in the current moment, either due to overload of existing service providers or due to altogether shortage of service provider nodes at the moment
- Dealing with dynamic attributes of the cloud service owners such as current load and network congestion level

4.2 Load balancing of service registrations and service queries

To address the problem of storing skewed attribute-value pairs, we consider using an approach called *Load Balancing Matrix (LBM)* originally proposed in [15] with slight alterations taking into account specifics of storing service descriptions

in (a, v)-graph. Here we give only an overview of the approach; for the complete description, analysis and evaluation of the LBM please refer to original paper [15].

In essence, authors propose using a set of nodes, rather than one, for storing popular attributes. Those nodes are organized into a logical matrix called *Load Balancing Matrix*. Each node in the matrix has a column and row index (p, r), and responsible node ID is determined by applying the overlay hash function to the triple (a_i, p, r) (while the approach in the original paper uses a 4-tuple that consists of the attribute with corresponding value, column index and row index). Each column in the matrix stores one subset, or *partition*, of the (a, v)-pairs that correspond to the attribute a_i. Nodes in the same column are replicas of each other, since they host the same subset of (a, v)-pairs.

The matrix dynamically expands and shrinks along its two dimensions depending on the load it receives. Due to this, matrices may end up in different shapes. For instance, a matrix may have only one row, when only the registration load is high, or one column, when only the query load is high. Each matrix uses a node, called *head node*, to store its current size and to coordinate the expansion and shrinking of the matrix. A head node is only responsible for its own matrix, and different matrices will likely have different head nodes, which are distributed across the network. Therefore, head nodes will not become the bottleneck of the system. However, when a head node leaves or crashes, vital information about its matrix, such as the size will be lost. To prevent this from happening, live nodes in the matrix send infrequent messages with their indices to the head node. According to routing properties of DHT overlay, a new node whose ID is close to the old head node's ID will receive these messages and become the new head node.

Though the approach described above will positively need some enhancements to be efficiently used in the cloud environment, the experiments in the original paper have shown its soundness and effectiveness for dealing with storing multidimensional data with skewed attributes distribution. Schematic depiction of load balancing matrix can be found in Figure 4.1.

Approach to the load balancing of the services search in the system is as an extension of the load balancing matrix method. Indeed, for the popular services there will be a LBM with large number of rows, containing replicas of the same service, and for the frequent (a, v)-pairs there will be a LBM with large number of columns (partitions). In addition, there are some optimizations that can be done to improve the efficiency of the search further. First, we can leave out services that are busy or unavailable at the moment. This step may be absent (or performed with certain delay) in case query processing node waits for the matching services to become available (this approach is described in the Section 4.3). Next, the node must decide which of the remaining services is returned to the requestor based on the principle

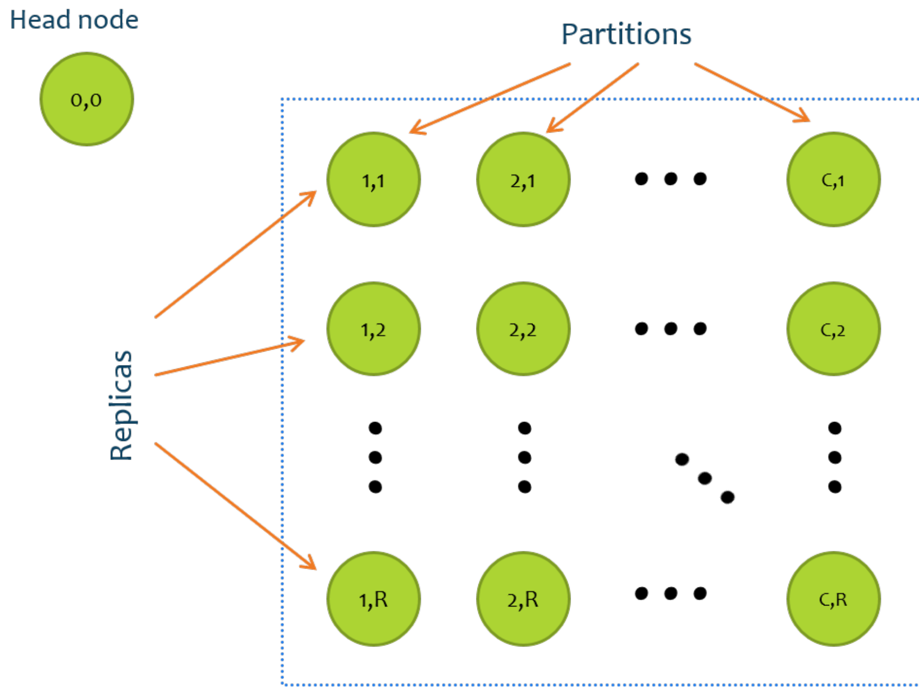

Figure 4.1: Load balancing matrix.

that it should not lead to over-provisioning of the concerned service. Also, there might be additional requirements, some of them are posed by the requestor, that need to be taken into the account, such as minimizing network congestion, minimizing client financial overhead, restricting geographical location of the instances etc.

4.3 Queuing

If we depict the flow from a client submitting a request to the completion (either successful or not) of this request in the form of a simplified flowchart (see **Figure 4.2**), it follows that there are several activities that can be performed by the node which originated the search (by the client's request) after the query is executed

1. In case when the service(s) exist and are available, return success message to the client along with the necessary data for the actual service usage.
2. In case where there are no available services for the given job at the moment (but they are available in principle), there is an option to wait for some service to become able to accept requests again and then proceed to the action

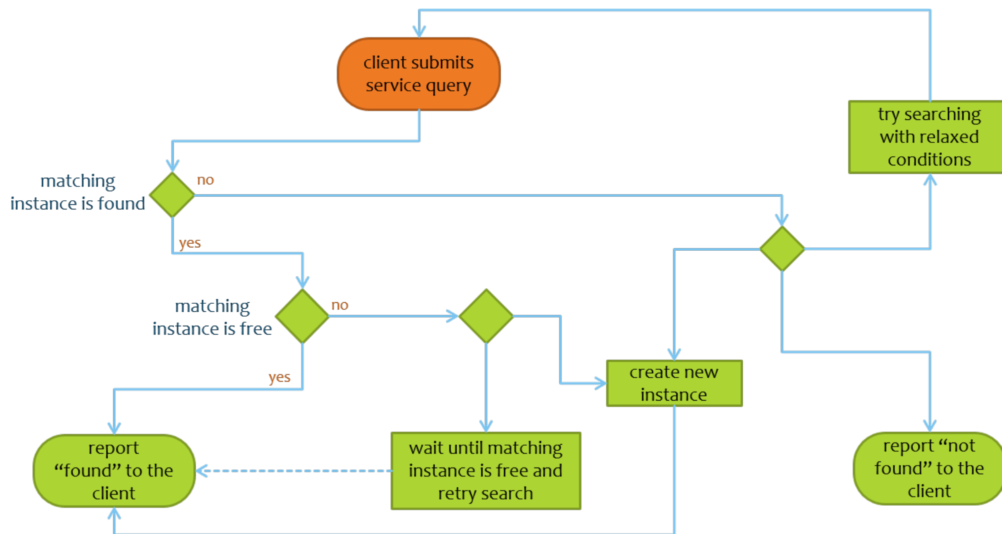

Figure 4.2: Service discovery in the cloud: client request flow.

described in 1. This scenario allows avoiding unnecessary instance creation and its subsequent shutting down, but it clearly must be used only when delaying the incoming job request will not lead to the system instability, in other words the wait must not create the bottleneck.

3. If there is no such service that corresponds to the submitted query, the node might ask cloud infrastructure to create new instance and proceed to action 1. This approach is usually used in case when all existing instances are busy and is the most common scenario in the cloud-based environments.

4. As an alternative solution in case when no services match submitted query, the node can try searching with less strict criteria (for instance, replacing point queries with range queries or searching for the service that satisfy only part of the query terms). Note that the attribute value domain in the query must adhere to certain requirements in order that obtained range queries be still meaningful and not change the meaning of the query too much. On the other hand, omitting some components of the query requires certain kind of weight distribution assigned to the each attribute of the query to avoid neglecting really important query terms.

4.4 Other issues

Among other issues that arise when applying service sharing and discovery approach to the cloud environments, there is yet another challenging one: dealing with dynamic attributes, such as current load, available disk space or network congestion level, which are naturally present in every dynamic environment. Although finding an efficient solution to this problem makes up a part of our future research, the most promising approaches are the following:

- Use execution history to create a probability distribution that would characterize predicted value of the parameter of interest
- Send messages to the neighbor nodes in a periodic manner in order to establish current values of desired parameters and propagate updates values in the overlay

Furthermore, considerable part of restrictions based on such dynamic attributes can be applied later, on the load balancing stage, as briefly described in Section 4.2.

4.5 Simulator implementation

The proof-of-concept implementation of the service description and sharing framework described in the Chapter 3 is done in C♯ programming language using Microsoft.NET framework (http://www.microsoft.com/net) and consists of the following main parts:

1. **DHT overlay:** a prerequisite for the efficient service sharing and discovery approach implementation. In our case we chose Chord DHT [**?**] since it meets necessary criteria, such as scalability of key location algorithm, efficient node joins and departure processing, formally proven statements concerning location of a key and overlay stabilization, and besides, is a well-known DHT overlay widely used both in academia and industry. The consistent hash function used in the overlay must generate values uniformly distributed in the name space and be not input-sensitive. In current implementation we decided to use SHA-1 as the system-wide hash function.
2. **Service storage:** an efficient data storage which supports all common types of queries, allows storing typed data and can be easily deployed on each node in the overlay. In current implementation we decided to use document-oriented

database MongoDB (http://www.mongodb.org/), which satisfies all the requirements stated above and is more flexible than many traditional relational databases, since it is a lightweight solution which has little deployment overhead, do not require predefined schema and do not include unnecessary at this point functionality, such as transactional processing.

3. **Overlay and statistics visualizing:** both are implemented as a web application, using HTML5 and JavaScript/jQuery (http://jquery.com/). Some visualizations are done using Flot (http://www.flotcharts.org/) JavaScript library.

Network for the experiment is implemented using .NET based WCF (Windows Communication Foundation) framework, which allows considerable flexibility as to specifying various connection types, network delays, message serialization formats etc.

During the implementation of the storage layer it became clear that at least three database tables per node are needed for the full coverage of basic data manipulation functions, that is (1) *putting service to the storage*, (2) *updating service stored at the overlay*, (3) *removing service from the storage* and (4) *getting service from the storage based on the query.* Tables and their structure are described in detail in **Tables 4.1** and **4.2**.

Table 4.1: Database tables used for storing services at storage node (description).

Local service data	Remote service data	Service attributes data
stores comprehensive data about given node services locally	stores attribute data for services in the overlay network; all attributes the node is responsible of are stored	stores information about responsible nodes for attributes of given service; service, responsible for storing this information, is determined by hashing service name itself

Table 4.2: Database tables used for storing services at storage node (structure).

Local service data	Remote service data	Service attributes data
• service name • all service attribute-value pairs with responsible node for each attribute • attribute value type (optional)	• attribute name • attribute value • service owner node • service name	• service name • all service attribute names with responsible nodes info

4.6 Experiment setup

In all following experiments we consider a network that consists of $N_C = 1000$ nodes. There exist $N_A = 100$ possible attributes in the system and $N_V = 100$ possible values, which result in $N_{AV} = 10000$ possible (a, v)-pairs. Dataset for service registrations is described as a 7-tuple $DSet_R = (N_S, N_A, N_V, E[a], Var(a), E[v], Var(v))$, where N_S — overall number of services in the dataset, $E[a]$ — mathematical expectation of number of attributes per service description, $Var(a)$ — its variance, $E[v]$ — mathematical expectation of number of values that attribute can have, and $Var(v)$ — its variance. Parameters $E[v]$ and $Var(v)$ allow us to introduce multiple-valued attributes into service descriptions. We generate two kinds of datasets: *uniform* and *skewed*. The former represents an ideal situation where attributes for the given service are chosen randomly using uniform probability distribution. This dataset is primarily used for comparison with more realistic skewed scenario. The latter is generated with a frequency of attributes of a service defined by discrete Zipf probability distribution with a fixed scale $s = 5$.

For service discovery evaluation we generated two query datasets, which are described as 5-tuple $DSet_Q = (N_Q, N_A, N_V, E[a], Var(a))$, where N_Q — overall number of queries in the dataset, $E[a]$ — mathematical expectation of number of attributes used in a query and $Var(a)$ — its variance. Again, we generate two kinds of query datasets: uniform and skewed, which are constructed similar to the service registration ones.

Both service registration and query arrival times are modeled using Poisson distribution with expected value (frequency) λ. Each node has three threshold parameters, namely R_r — maximum registration rate, R_q — maximum query rate and T_r — maximum (a, v)-pair registrations that node can hold in its internal database.

Table 4.3 shows all parameters and notations used in the simulation.

Table 4.3: Parameters and notations used in the simulation.

Symbol	Meaning
Network	
N_C	number of nodes in the overlay network
N_A	number of possible attributes in the system
N_V	number of possible values for each attribute in the system
N_{AV}	number of possible (a, v)-pairs
Service registrations dataset	
$DSet_R$	dataset
N_S	overall number of services in the dataset
$E[a]$	mathematical expectation of number of attributes per service description
$Var(a)$	variance of $E[a]$
$E[v]$	mathematical expectation of number of values that attribute can have
$Var(v)$	variance of $E[v]$
Service queries dataset	
$DSet_Q$	dataset
N_Q	overall number of queries in the dataset
$E[a]$	mathematical expectation of number of attributes used in a query
$Var(a)$	variance of $E[a]$
Thresholds	
R_r	maximum registration rate
R_q	maximum query rate
T_r	maximum (a, v)-pair registrations that node can hold in its internal database
Other parameters	
λ	expected value of Poisson distribution that is used to model service registration and query arrival times

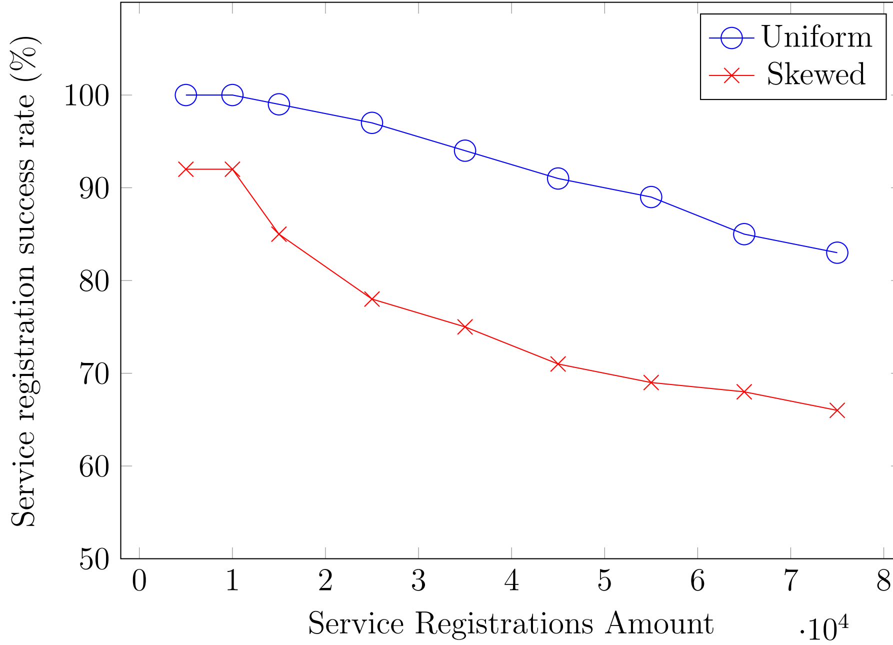

Figure 4.3: Service registration success rate comparison

4.7 Simulation results

First, we examine success rate of registrations as the number of registrations increase and perform experiments both for uniform and skewed service datasets. Service registration arrival time is modeled with Poisson distribution with frequency $\lambda = 50\ reg/s$. Maximum (a, v)-pair registrations per node is chosen to be $T_r = 100$ and threshold $R_r = 30\ reg/s$.

Figure 4.3 shows how registration success rate changes depending on how many services were fed to the system. According to Section 4.6, there can be two possible factors that cause failure in service registration — exceeding the maximum registration rate R_r, which is influenced by parameter λ of the Poisson distribution used to model service registration arrival times (and the overall amount of service registrations, since they are executed in parallel). Exceeding the value of T_r is the other possible cause of failure.

We observe that for the uniform dataset registration success rate curve gradually drops as the services amount N_S increases. Moreover, for the large values of N_S the drop in the curve become more prominent, since more and more nodes reach the threshold T_r and are not able to process new (a, v)-pair registrations, resulting in service being not registered in the system at all. However, in the case of uniform

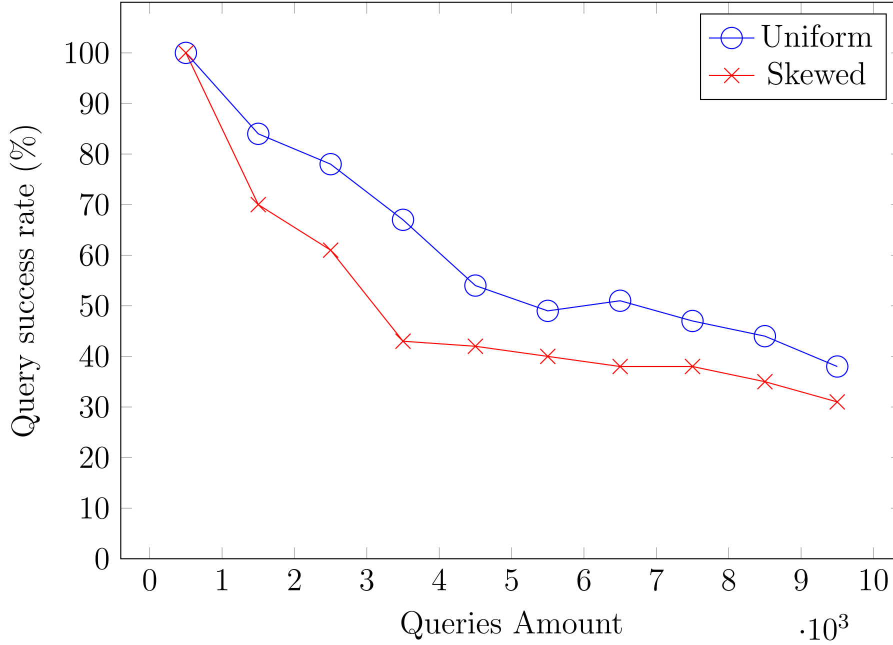

Figure 4.4: Query success rate comparison

dataset, the main factor that causes service registration failures is node saturation with regard to threshold R_r, therefore it is possible to lessen its impact by regulating the arrival rate or increasing overlay node processing capabilities and network bandwidth.

When we compare the ideal scenario above with more realistic skewed dataset, the drop in the registration success rate becomes more significant, since relatively small amount of nodes, which hold most popular attributes, become saturated very quickly, meaning that T_r becomes major bottleneck in the system. On the other hand, role of R_r becomes much less significant, as it mostly affects nodes with most popular attributes when they are already close to saturation with regard to T_r.

Next, we study how the system behaves as the query load increases. For this, we measure the rate of queries failed because of reaching maximum possible query rate at the overlay node $R_q = 100\ reg/s$. Again, we perform the measurements for the uniform query dataset, where all attributes can appear with the same probability, and for skewed one, where attributes are assigned weights according to the Zipf distribution. The amount of services registered in the system $N_S = 5000$ is fixed and queries arrival time is modeled with Poisson distribution with frequency $\lambda = 10\ reg/s$.

As can be seen from **Figure 4.4**, even for the unrealistic uniform query dataset case, query success rate is 100% only for the small amount of queries and it steadily

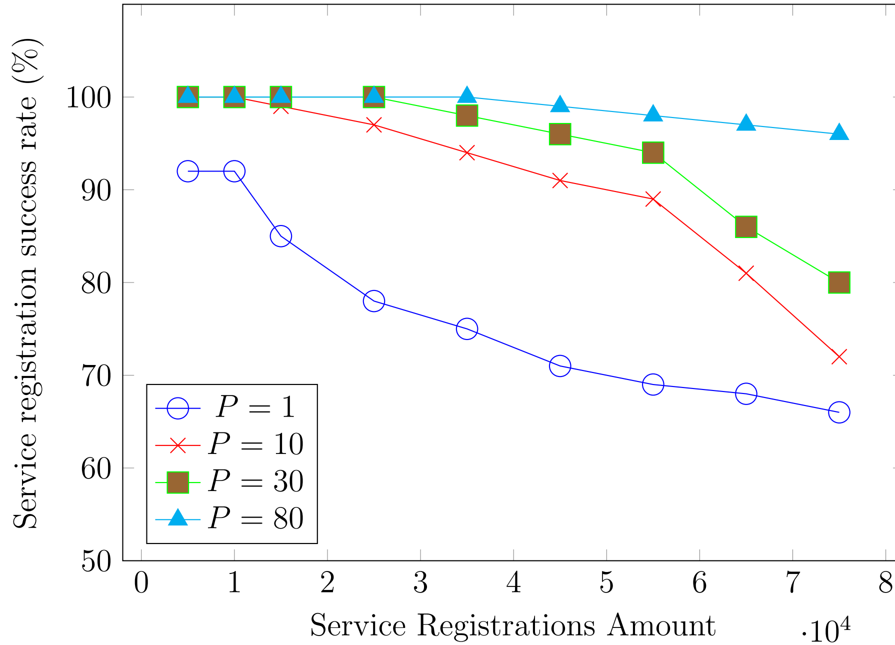

Figure 4.5: Service registration success rate comparison using LBM

drops down to the 50% as the query load increases. When we compare it with the skewed scenario, the drop becomes more evident, especially for relatively small query amount, drawing closer to the uniform scenario afterwards. This can be explained by the fact, that almost every query in the real-world scenario will contain one or more popular attribute, and the query execution will fail all the same, even when other parts of it (that contain non-frequent attributes) succeed. This can be somewhat mitigated with the technique proposed in Section 4.3 (item 4), but actual cases when the popular query can be omitted are relatively few.

These experiments show that the system needs proper load balancing mechanism to be scaled in an efficient way. According to more experiments shown in [15], LBM (Load Balancing Matrix) approach has desired features in this regard, therefore we tried to use it as a load-balancing mechanism for our approach. For this, we executed another set of experiments for service registrations, now with load balancing matrix for several values of P — amount of partitions. This time only skewed datasets were used. Graph with $P = 1$ corresponds to the system without load balancing matrix. Results are shown in Figure 4.5. It can be seen that increasing the number of partitions leads to less service registration failures. In a similar way, we compared how system behaves for service queries from skewed query dataset with and without load balancing matrix replication. The result is shown in Figure 4.6

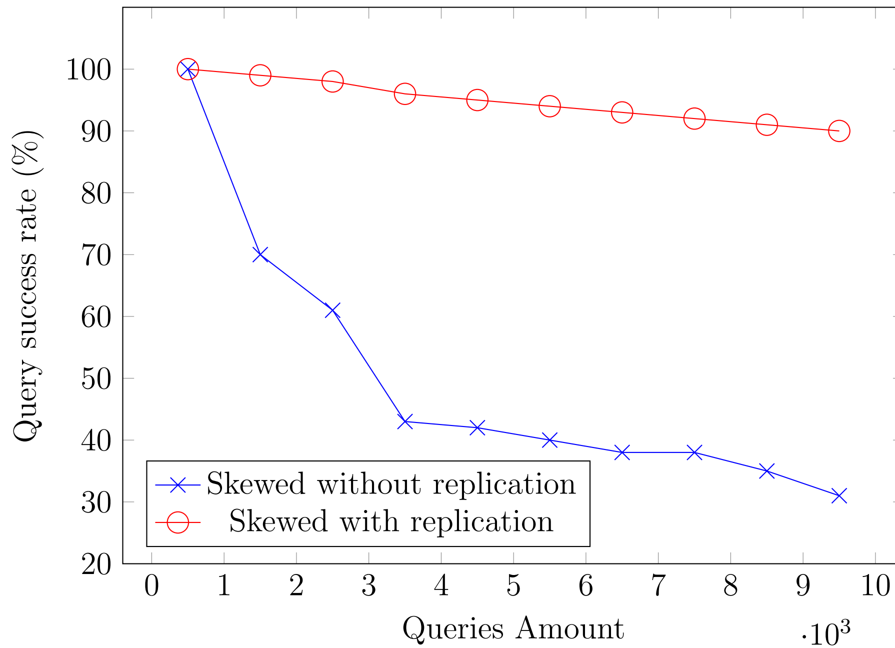

Figure 4.6: Query success rate comparison w/replication

4.8 Example domain

To further demonstrate the system operation, we chose a domain of dedicated gaming solution based on Google Cloud Platform [24], since in our opinion it represents one of the typical scenarios found in distributed systems, requiring elaborate strategies for load-balancing and efficient service provisioning.

First, according to the overview, key components of the proposed solution are the following:

1. Game server selection
2. Player's game client connecting to dedicated game server
3. In-game requests and Google Compute Engine instance health checks
4. Autoscaling game servers
5. Storing logs for analysis and MapReduce
6. Analysis of massive user and game datasets using Google BigQuery

In our example, we decided to outline four main activities that occur in the gaming scenario and are worth considering from the cloud computing point of view:

1. Search for the best matching game server
2. In-game requests which utilize separate game components, specifically micro-payments for game items or services
3. Dedicated player versus player or team versus team requests
4. Requests for game or user data storage, extraction and analysis

Consequently, the most straightforward scenario we can elaborate in the given setting would require four types of services, each corresponding to the one of the activities described above. Each of the four services is characterized by the set of the attributes, including ones specific to that service type. The possible example of service descriptions in given domain is shown in **Figure 4.7**.

```xml
<data>
  <node type="Game Server" owner-node="166">
    <cpu-utilization type="percent">30</cpu-utilization>
    <architecture>64</architecture>
    <os type="custom:osdata">
      <type>windows</type>
      <version>2012 Server</version>
      <edition>datacenter</edition>
    </os>
    <processor-cores type="integer">4</processor-cores>
    <clock-rate type="float">2.7</clock-rate>
    <location>North America</location>
    <game-region type="custom:gameregion">S14</game-region>
  </node>

  <node type="Game Server" owner-node="31">
    <cpu-utilization type="percent">70</cpu-utilization>
    <architecture>64</architecture>
    <os type="custom:osdata">
      <type>windows</type>
      <version>2012 Server</version>
      <edition>datacenter</edition>
    </os>
    <processor-cores type="integer">2</processor-cores>
    <clock-rate type="float">2.3</clock-rate>
    <location>South America</location>
    <game-region type="custom:gameregion">A08</game-region>
  </node>

  <node type="Micropayment Server" owner-node="37">
    <payment-provider>PaypPal</payment-provider>
    <encryption>AES-256</encryption>
    <authentication-mode>token</authentication-mode>
    <location>South America</location>
  </node>

  <node type="Analysis Server" owner-node="37">
    <cpu-utilization type="percent">20</cpu-utilization>
    <architecture>64</architecture>
    <os>
      <type>windows</type>
      <version>2012 Server</version>
      <edition>datacenter</edition>
    </os>
    <processor-cores type="integer">2</processor-cores>
    <clock-rate type="float">2.3</clock-rate>
    <db type="custom:ddata">
      <type>microsoft sql server</type>
      <version>2012</version>
    </db>
    <location>Europe</location>
  </node>
</data>
```


Figure 4.7: Node services data example.

Chapter 5

Decentralized architecture for cloud bursting

5.1 Current state-of-the-art

In this section, we examine the current state-of-the-art and identify main research directions that deal with the resource provisioning and management in the inter-cloud and specifically in the hybrid cloud which present significance to the problem of building an effective cloud bursting architecture. Due to the relative novelty of the topic, amount of proposed frameworks is scarce and many of them extend previous research activities on Grids interconnection and interoperability. Among them are Contrail [9] that introduces a centralized composite entity that acts as a single entry point to a federation of cloud providers. Components of the solution provide single sign on, mapping using requests to cloud resources, support observing the states of cloud providers and facilitate the federation-wise SLA. By using either internal or external adapters for different cloud providers, Contrail can support both centralized cloud federation as well as federation-agnostic structure. In mOSAIC [25], unlike most other inter-cloud technologies, there are some assumptions regarding the architecture of the deployed application. For instance, the application architecture must be service oriented and use only the mOSAIC API for intercomponent communication; also, developers must specify the resource requirements in terms of computing, storage and communication. Some solutions are positioned as peer-to-peer based, such as OPTIMIS [26] and RESERVOIR [27]. Former is built around special software agents that are deployed within cloud providers and application brokers. Those agents perform cloud providers discovery and negotiation (mostly based on SLA data), deploy the service by providing appropriate virtual machines, taking into consideration current workload, posed restrictions and evaluation of po-

tential profit. However, OPTIMIS architecture is designed in order to anticipate a variety of architectures for simultaneous use of multiple clouds (federated, multi-cloud, aggreagtion of resources by a third party broker, hybrid cloud), providing all necessary components and therefore is, in fact, a fully integrated multipurpose solution. While it can be argued that such multipurpose solutions are quite powerful, on the other hand, it can become another point of vendor lock-in and lacks necessary modularity and loose-coupledness. On the other hand, RESERVOIR project relies on the usage of elasticity rules of the type “trigger-action”, which facilitate custom application brokering. As a result, the system attempts to create an elasticity model that optimizes the control strategy without violating the SLA constraints. In this model, all deployed applications are required to be agnostic of the hosting data center location, which allows transparent deployment anywhere within the federation. However, as was already mentioned before, it have similar concerns as to not addressing specifically cloud bursing scenario and having all components integrated. More detailed analysis of inter-cloud frameworks can be found in [6].

As for cloud bursting research, the results here are even sparser and mostly include the following two prominent frameworks for building inter-clouds that explicitly tackle cloud bursting related problems. First, there is the architecture described in [28], which is built on top of Aneka [29] — a software platform and a framework for developing distributed applications in the Cloud managing computing resources in a heterogeneous network. Another approach is taken by OpenNebula [30] which is positioned as feature-rich and flexible solution for the comprehensive management of virtualized data centers to enable private, public and hybrid IaaS clouds. There are also slightly more niche research topics, such as investigating a decision support model for cloud bursting [31] or developing a reference design for secure cloud bursting, which gives special attention to the problems of trust and security in this model.

5.2 Model overview

At the beginning of the description of our model, we provide details about participating entities. In case of cloud bursting scenario, they usually include *service provider* (SP), which provides one or several services, which it operates on premises using private cloud, and *cloud provider* (CP), which is a company that offers a cloud computing solution in the form of IaaS, PaaS or SaaS (or any combination thereof) to other businesses or individuals. Cloud bursting scenario typically involve several CPs, either on the evaluation stage, during which SP choose the CP it would use for cloud bursting, or during actual multicloud stage if SP uses several public clouds. Chosen CP is called *cloud bursting target*. In our case, scenario also

includes another role, called *client*, which represents an entity that has some work to perform in the cloud (either in the form of the workflow instance or a single job request). While client in general might be a separate entity, in most scenarios SP acts in this role. Lastly, we will call a request from the client a *job request* from now on, but it should be noted that this definition differs from the similar terminology that exists in Grid systems.

Job request does not always have one-to-one relationship to a service query (see Section 5.4), since the query might be executed several times for the same job request while waiting in the job request queue. Each job that is submitted to the cloud describes the list of services that need to be called, input and output parameters, required service properties and certificates, QoS requirements, etc. Alternatively, job requests that are submitted to the cloud can be the part of some workflow, which is described using existing workflow description languages such as BPEL [32] or YAWL [33]. Either way, service query is constructed by transforming these descriptions to the format described in Section 5.4.

Table 5.1: (a, v)-graph types summary.

(a, v)-**graph**	**Description**	**Attribute examples**
S-(a, v)	contains properties of service provided by SP	differ by actual service
I-(a, v)	contains properties of virtual machine instance where service is deployed	CPU information, maximum available memory, maximum available hard disk space, OS type and version, available licenses, installed certificates
L-(a, v)	contains lease-specific properties of cloud bursting target virtual machine instance where service is temporarily deployed	maximum lease time, lease slot duration, resource price

5.3 Service descriptions

Services provided by SP are described using (a, v)-graph notation, introduced in Section 3.5. It is important to note that while (a, v)-graph descriptions for each service can be created from scratch, more realistic approach, which takes into account reusing existing service data, would be the transformation of existing descriptions, possible using XSLT for model transformation. We call the resulting (a, v)-graph a *service* (a, v)*-graph*, or simply *S-*(a, v)*-graph*. Next, each virtual machine instance in the cloud is characterized by its own (a, v)-graph based description, which includes hardware characteristics such as CPU information, maximum available memory, maximum available hard disk space, operating system type and version, available licenses and installed certificates, etc. This graph is built similarly to the S-(a, v)-graph and is called *instance* (a, v)*-graph*, or simply *I-*(a, v)*-graph*. Virtual machine instances leased from the external CP add more (a, v)-pairs to the descriptions of the services they host. Those (a, v)-pairs contain information such as maximum lease time, lease slot duration or resource price, and form so called *leased resource* (a, v)*-graph* or simply *L-*(a, v)*-graph*. Finally, service, instance and leased resource (a, v)-graphs are merged into one graph called *full-*(a, v)*-graph*, which is stored at the overlay node. Full-(a, v)-graph contains the node that holds low-level routing information about the instance. Table 5.1 shows the summary of all types of (a, v)-graphs used in current model.

5.4 Service queries

Job request is a crucial part of cloud bursting approach, since it acts as the mechanism which determines the status of the internal cloud and verifies to what degree provisioned services fulfill the established service level agreement and other auxiliary requirements. In case the job that is being submitted to the cloud cannot be executed with only internal resources, leasing additional resources from external cloud must be performed. In our model we construct such mechanism using the set of *service queries* $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_s\}$ that formally describe each job request J and return the set of available service instances $\mathcal{I}$ that are provided by either SP or one of the CPs that currently act as a cloud bursting target. The service query is expressed in a format introduced in Section 3.6.

The node, which handles the query, is called *brokering node*. Apart from service properties, query can include special terms, called *provisioning policy terms*, which help regulate the rate and scale of overlay growth. These terms include

- **Cost restricting terms:** For example, current job request has low priority and should either lead to restricted leasing of external resources or result in no leasing at all.
- **Queue wait time:** This term regulates how long the request should wait in the queue for local SP resources before it switches to leasing external ones.

5.5 Overlay formation and scaling

Next, we describe the procedure of forming a peer-to-peer overlay which is the pivotal element in the proposed architecture. It is formed with cloud virtual machine instances as nodes, each of which has at least one service deployed, however some services may be deployed on several instances as well, depending on the capabilities they require and estimated initial demand. We consider it reasonable to assume that services of SP form an overlay network of considerably large amount of nodes, since if the opposite were true SP most probably would not need to resort to building cloud bursting solution. The form and the scale of the overlay is defined by the combination of the following scenarios:

1. **Initial configuration scenario:** Starting point of building cloud bursting architecture. Here SP first deploys its services into own private cloud with minimum needed amount of instances for each service, therefore establishing sufficient infrastructure for executing necessary jobs at the moment.
2. **Vertical scaling scenario:** This scenario consists of expanding or contracting (increasing or decreasing the number of instances) within the bounds of the SP private cloud and does not incur extra-corporate expenses.
3. **Horizontal scaling scenario:** This scenario represent the actual cloud bursting process: temporary leasing virtual machine instances from cloud bursting target and eventual contracting to the original state of having only private cloud instances. Choosing one or several CPs that will act as a cloud bursting target is also performed here, but can be seen as optional, because in the real-word scenario SP will not perform full CP selection process each time the horizontal scaling occurs.

By utilizing peer-to-peer overlay, we achieve certain level of homogeneity, which allows us to abstract from actual nature of scaling scenario (whether it is vertical or horizontal), since in both cases new virtual machine instances are represented as overlay nodes with corresponding service descriptions that are linked to the actual

instance routing information. Given this model, each job request will result in one of the following situations:

1. Requested service is available and job request satisfies restrictions introduced by resource provisioning policies.
2. SP infrastructure lacks either power (expressed in CPU, memory etc) or capabilities (expressed in operating system kind, software, license, protocols support, certification etc).

In the first situation, the job is submitted to corresponding instance(s). For the second one, there are two alternatives:

1. Brokering node puts the request in the job request queue, performing periodical re-querying.
2. Scaling is performed, resulting in increased amount of instances and/or adding required capabilities.

5.6 Platform components

In this section we describe the components which compose proposed solution. Due to modular approach to designing and implementing the solution, components are loosely coupled, allowing replacing or refactoring them if needed. This approach also makes possible using a wide range of third-party solutions for queues or databases. We argue that proposed solution eliminates most centralized components, that usually appear in many other publications related to cloud bursting or multicloud organization, as shown in Table 5.2.

The components that are used in the proposed model are as follows. Dependencies between components are shown in the Figure 5.1.

Job request processor: Gateway component which accepts a job request from a client. The job request is put into the job request queue, and then a service query is formed and sent to the service query processor, which returns the result together with further action. Depending on this action, job request processor either contacts provisioning manager and eventually return available instance(s) information to the client or repeats service query later. When job is finished, its results (if they are present) are returned to the client. In many cases, clients may receive not the actual result value, but a set of pointers to the component where the result data is stored.

Table 5.2: Centralized components eliminated in proposed model.

Component	Examples
Resource pool	FCM Repository [34], Cloud Computing Resource Catalogue [35]
Service request processor	Federation Runtime Manager [9], Generic Meta-Broker Service [34], Resource Broker [25]
Resource provisioner	CloudBroker [34], Cloud Optimizer [26], Client Interface [25]
Multicloud broker	Primary Cloud Provider [10], Cloud Coordinator [3], Intercloud Exchange [35]

Service query processor: Component which is responsible for processing the service query, which was formed by job request processor based on the job request from a client. Since service query processing itself is a multi-stage process, the component performs the following functions:

- Accepts service query and puts it to the service query queue.
- Processes the query according to procedure described in [36].
- After getting the query result, returns it to the job request processor along with the action that is supposed to be taken (such possible actions are described in detail in Section 5.5).

Since proposed solution is distributed, any overlay node can act as job request processor and service query processor for the incoming query. In practice, actual node is chosen using round-robin algorithm in order to distribute the load and eliminate unnecessary large request queues. Furthermore, while the architecture actually does not prescribe that those two components must be located at the same overlay node, it seems reasonable to do so for given job request and service queries associated with it in order to decrease network traffic and improve reliability. Refer to the Figure 5.2 for more detailed description of decentralized processes in cloud bursting architecture.

Job request queue: Component which holds job requests with their respective service queries being processed or scheduled to do so. Physically, the queue is organized as a distributed queue where master node corresponds to the overlay node that acted as job request processor, and several other nodes in the overlay act like a slave nodes with duplicate queue instances.

Service query queue: Component which holds service queries which are currently being processed. This queue can be organized in a distributed way similar to job request queue, but in reality due to reliability concerns it is coupled with service request processor and located at the same node.

Provisioning manager: Component which is responsible for provisioning of resources in case the current ones are not sufficient to process a job request. It is contacted by job request processor in case service request result indicated that new instances must be created for given job request. On this stage, provisioning manager becomes the point from where cloud bursting process starts in case local resources are not enough. First, it contacts cloud bursting target searcher component to obtain the information about most suitable cloud bursting target. Having established the target, provisioning manager sends the request to external authentication manager, which provide necessary credentials and other authentication data needed to start provisioning process from external cloud. It uses appropriate adapter components to interact with external CP, obtain routing information for leased resources and return it to job request processor.

Resource pool: Since most cloud providers offer billing plans based on fixed time blocks (e.g. X dollars for Y hours of uptime), it is reasonable to maintain a pool of resources that once was leased from CP with job that requested them is finished but the dedicated time slot is not over yet, so they can be reused. In contrast to most centralized solutions, this pool does not exist as a standalone component — the nodes just remain as the part of the overlay with respective services deployed. Nodes are removed from the pool (and therefore from the overlay) based on their L-(a, v)-graph information. As for the removal, there are two ways to implement it. First one implies that provisioning manager monitors all resources leased from external CP and releases them when needed, communicating with job request processor if necessary. This approach is currently adopted in out model. Alternatively, leased resources can be managed in the distributed way when overlay nodes check their neighbors and send corresponding "release" message to provisioning manager. This way is preferable considering distributed nature of the framework and we plan to switch to it during our future research.

External authentication manager: Component which stores and manages various authentication information (credentials, authentication tokens, security certificates, etc.), that are necessary to successfully perform resource leasing from external CP. This component is contacted by provisioning manager during cloud bursting phase. This component uses adapter components for interacting with external CP.

Cloud bursting target searcher: Component which provides information about cloud bursting target when horizontal scaling is performed. Possible cloud bursting targets are first listed in the initial configuration of the component in the

order of preference. This configuration is created based on open and available data about CPs, which include pricing plans, available geographical locations, supported platforms and services, etc. In addition, data from cloud monitoring system component is used to change this initial preferences order. The design of proposed solution also allows the administrator to participate in cloud bursting target selection process by manually performing selection via GUI.

Cloud monitoring system: Component which collects, aggregates and stores diagnostic and monitoring data obtained from hybrid cloud instances. While every CP provides its own cloud performance monitoring system (for example, Amazon CloudWatch [37] or Cloud Monitoring [38]), they are not designed to be cloud-agnostic, which renders their usage in multicloud scenarios difficult. Notwithstanding that, there are some systems which are oriented for multicloud usage [39] and some of them can be used in our solution. The data that is being obtained by a monitoring system is highly dynamic (for instance, current load) and possibly requires aggregation or join operators when being queried, therefore it is difficult to efficiently store it inside the overlay itself. However, while it does not seem reasonable to make such system truly decentralized, it can benefit from certain techniques such as replication. Monitoring system output can be used to decide whether we need to perform scaling in some particular situation. For example, if we take the scenario when there are no available instances that match service query, the monitoring data can be used to decide the period for retrying the query, and if the estimated period length exceeds certain threshold, perform scaling. In addition, cloud monitoring system is responsible for collecting statistics regarding previously leased external CP resources, which provide an input for cloud bursting target searcher.

Adapter components for interacting with external CP: While there are some efforts to provide a common standard API for the cloud [40], at the moment each cloud provider still uses vendor-specific interfaces for cloud programmatic access and communication with their cloud resources. This brings forward the need to maintain separate driver components for each possible cloud bursting target. In our solution, those components constitute the part of the application, that is deployed on every instance.

5.7 Synchronization and conflict resolving in job request/service query queues

While being an effective way for improving scalability and robustness of the system, distributing job request and service query processing (we use "job requests" term only for the rest of this section for brevity, but this discussion concerns both

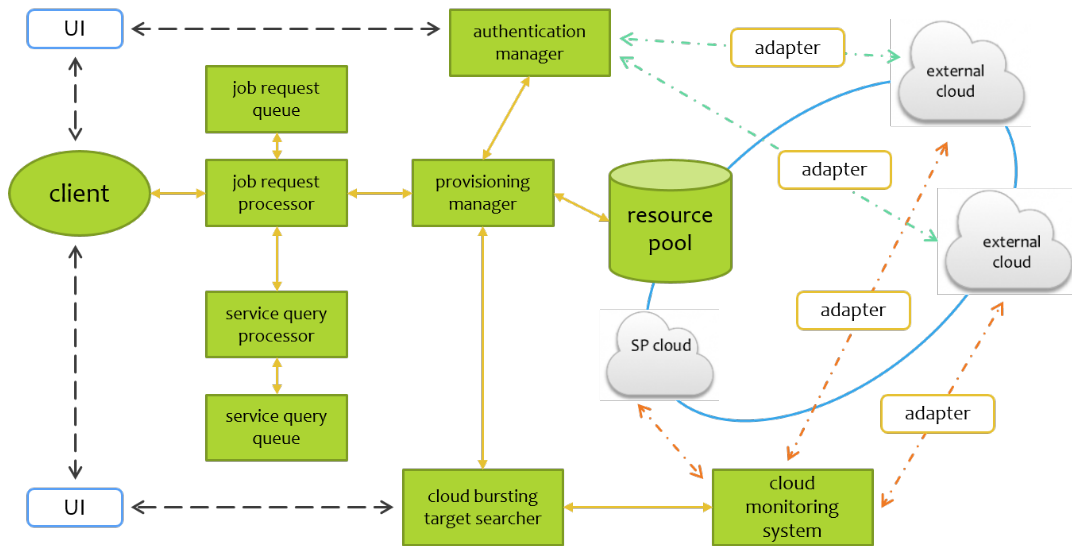

Figure 5.1: Model components.

job requests and service queries) presents another problem: handling simultaneous requests and overall synchronizing the queues. More specifically, let us consider the situation when several (say, two of them, for the sake of more determinate analysis) job requests for the same service independently arrive at different nodes which act as job request processors. Next, they are transformed to service queries and sent to the overlay (acting here as a resource pool) where they both return some resource R as a result to the job request processor. Supposing that R can serve only one more client at this time, how do we ensure that the following exceptional situations will not occur for R:

1. Racing condition when both requests claim the resource
2. Providing a resource when it is already invalid, because it was occupied by another requestor
3. Both requestors reporting that a resource is unavailable, one of those being a false information

First, let us consider traditional methods for resolving this issue. One of them is to introduce one more queue (we can name it *synchronization queue*), which will act as a synchronizing element for all job requests in the system. Since there is not need for actual copying of job request content, only their identifiers and timestamps may be present in this central queue. However, introducing such a component brings on another problems: that is, now we need reliable clock synchronization across the

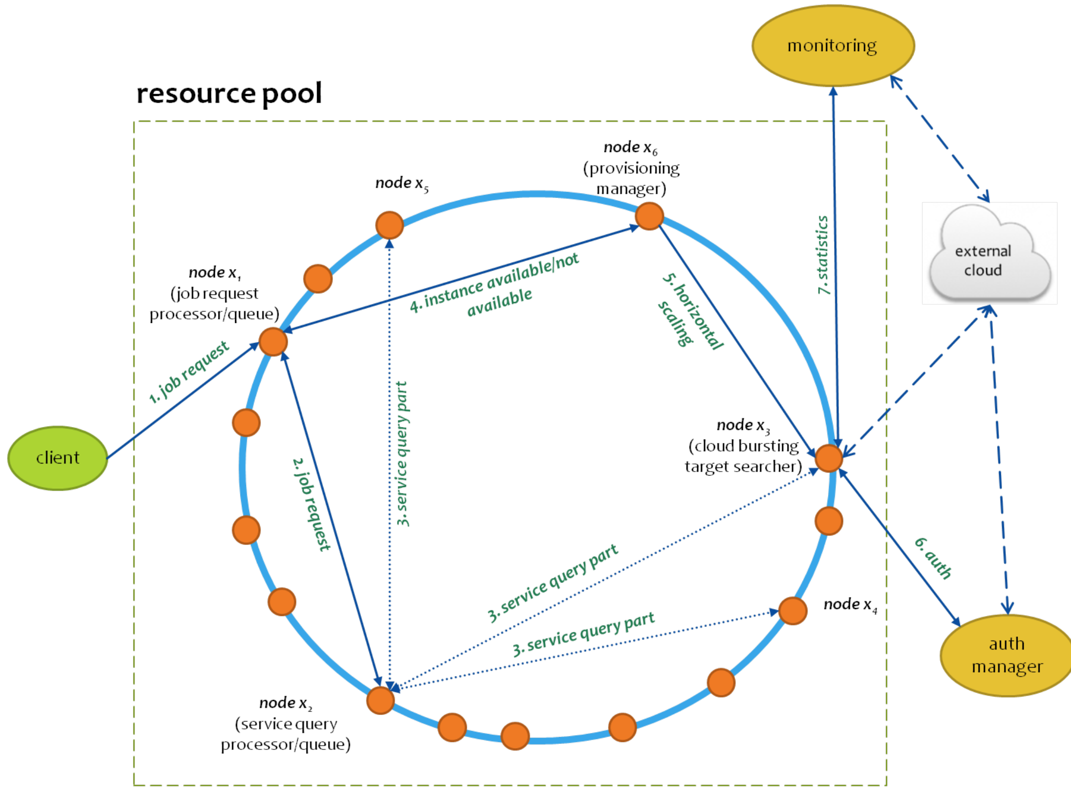

Figure 5.2: Decentralized cloud bursting architecture.

system, including parts of hybrid cloud that are leased from external cloud provider on a temporary basis. Examples of such synchronization methods are Network Time Protocol [41], Precision Time Protocol [42] or clock synchronization using GPS. Apart from considerably increasing the system complexity, this approach presents yet another problem: synchronization queue becomes yet another bottleneck and single point of failure in the system, thus defeating the whole purpose of cloud bursting architecture decentralization.

Now let us analyze an alternative which consists in resolving conflicting job requests inside an overlay. From this point of view, we have two queries that are satisfied by the same resource located on some node N_R. To resolve an issue when both requestors claim the same resource we can introduce a *status* flag of a resource which indicates whether it is free or claimed by some requestor. This property is saved a part of S-(a, v)-graph on the owner node only and not propagated to the network and other nodes, since doing so will bring on another kind of synchronization problem. When node which executes the part of the query that is satisfied by this resource, it contacts the owner node and checks the flag value. If the value is "free", it sets the flag to "claimed" state and assumes that query part is satisfied.

Otherwise, execution flow for this query part excludes given resource from the result set. Since accessing the database of a node (for manipulation with "status" flag) is already governed by internal synchronization primitives such as locks, we take it that no synchronization conflicts could occur here. Resetting of the flag back to "free" state is done by job request processor instance responsible for that particular job.

5.8 Considerations for evaluation and comparative analysis

In this section we will present metrics and other considerations for evaluation and comparative analysis of our proposed approach which is done as a part of future research on this topic. In order to test behavior and performance of the system we are currently building a simulator for cloud bursting scenario, which contains all components described in Section 5.6 and mocks necessary interfaces that are present in real cloud setting.

First, we propose several basic performance metrics as follows.

- **Average time of job request in job request queue**. This metric shows overall performance of the system from client's perspective, that is how long does it take in average for its job requests to be executed. It defines the overall responsiveness and effectiveness of the system.
- **Average amount of service query executions (reformulations) for given job request**. This metric reflects the responsiveness of the system and its sensitivity to the private cloud overload. It follows from the definition that the value of this metric should not ever exceed certain acceptable threshold.
- **Average lifetime of a resource in the resource pool**. This metric estimates the size of the hybrid cloud. It can be measured separately for SP and external CP instances to show how heavily SP depends on leased resources.
- **Average cost per job request**. This is a metric that shows how costly is job request processing from SP perspective. If we assume that intra-cloud processing costs are constant, the value of this metric will grow according to how much external cloud resources are used to process incoming job requests. The smoother is this growth the more optimal and precise is the cloud bursting process, including reusing already leased resources and optimal waiting time for incoming service queries.

- **Network load**. This metric shows how efficiently the system is using network resources. This influences scalability of the system and, more directly, its cost effectiveness.

Next, there is a need for comparing proposed solution with other research results in this area. While there already are multitude of comparative analysis research for traditional cloud systems, intercloud and cloud bursting systems presents certain complexity in this aspect because of novelty of the topic. Therefore, we try to introduce several ways of performing such analysis for your solution. First of all, Table 5.3 shows how our solution fall into intercloud systems taxonomy introduced in [6], therefore making possible its comparison to already existing systems similarly described in the aforementioned paper. In particular, we can compare based on the following characteristics: a) level of adhering to client SLA requirements; b) how optimal is the use of SP costs; c) level of cloud monitoring and how far this data is leveraged to optimize bursting target selection; d) how system reacts to abrupt/planned peaks in the load; e) robustness in case of partial cloud failure. In cases when these and other criteria are hard to measure using quantitative methods, we intend to perform detailed analysis to show pros and cons of our approach from the qualitative aspect.

Table 5.3: Intercloud taxonomy description of the system.

Feature	**Explanation**
SLA-based brokering approach	application developers specify the brokering requirements in an SLA in the form of constraints and objectives
Singular jobs	executed only once, unless they are rerun by the users or automatically rescheduled upon failure
Periodical jobs	repeatedly executed over a period of time.
Compute-and-data-intensive	facilitate user access to a persistent storage or perform massive computations
Geo-location aware	requests should be scheduled near the geographical location of their origin to achieve better performance
Pricing aware	detailed and up to date information about providers prices and policies to perform fiscally efficient provisioning
Legislation/policy aware	application broker should take into account legislative and political considerations upon provisioning and scheduling
Local resources aware	usage of local in-house resources with higher priority than that of external ones

Chapter 6

Conclusion and future work

6.1 Conclusion

In this thesis, we have presented the decentralized approach for efficient service sharing and discovery based on peer-to-peer overlay. In order to adapt existing peer-to-peer DHT overlays we introduced formal description of the problem of service annotation, their distribution in the network as well as their discovery mechanism. We presented algorithms based on aforementioned formal definitions that cover the process of service sharing and discovery in peer-to-peer overlay. As practical application of this approach, we applied it to the problem of service sharing and discovery in the cloud. We also introduced several enhancements of the original approach that allow using it more efficiently in cloud scenarios. We have shown the soundness and robustness of this approach by constructing simulation environment and conducting the set of experiments. This work eventually lead us to even more challenging area of application, namely building cloud bursting architecture based on peer-to-peer overlay for service provisioning. We presented an approach of building decentralized cloud bursting system, outlined scenarios and workflows that emerge in it, and provided detailed description of the components that comprise the system. We also conducted its evaluation by comparative analysis with similar system and outlined guidelines for further, more detailed evaluation.

6.2 Achievements

- Introduced and formally described decentralized service sharing and discovery approach in peer-to-peer overlay

- Applied the above approach to the problem of service provisioning in the cloud and made necessary adjustments and enhancements, including load balancing for service registrations and service queries
- Verified our approach by simulation using existing DHT overlays (Chord, Kademlia) and cloud simulator (CloudSim)
- Introduced decentralized architecture for cloud bursting using the approach above, including detailed description of scenarios, workflows and system modules
- Conducted evaluation of the above by comparative analysis with similar system and outlined guidelines for further, more detailed evaluation

6.3 Benefits

- Utilizing of well-known DHT, which guarantees the correctness and soundness of underlying peer-to-peer overlay
- Partitioning of multidimensional service description space over the set of overlay nodes, that naturally allows execution of range queries
- Introducing decentralized brokers to cloud bursting architecture, which eliminates single point of failure and increases robustness of the system
- Eliminating separate resource pool component from cloud bursting architecture and instead forming it in form of cross-cloud peer-to-peer overlay, therefore making the system highly scalable
- Modular framework architecture, which allows using wide range of existing queue or storage components

6.4 Future work

- Automate decision making process for choosing most suitable cloud bursting target, possibly by using detailed analysis of historical data to build a knowledge database, which then could be used to infer optimal choice
- Adapting proposed model for hosting business workflows
- Enhance load balancing of underlying peer-to-peer overlay

- Build (or integrate existing) system statistics and monitoring subsystem which can be using for more granular tuning of an overlay

Acknowledgments

First and foremost I would like to thank my supervisor, Prof. Norihiko Yoshida. His expert guidance helped me in achieving the goal of my research, and I greatly appreciate his contributions of time, ideas and every advice he gave me during my research. It has been a great honor to be his Ph. D. student.

I also would like to thank my secondary supervisors and reviewers: Profs. Cheng, Yoshiura, Koshiba and Goto for their valuable advices and support. Also I would like to thank all my lab mates for their help and support allowing me to be motivated and productive during my research. I will always remember the time we spent together. Especially I would like to thank Assistant Professor Noriko Matsumoto for her help and efforts she makes to create lively and comfortable atmosphere at the lab.

I also would like to thank my family and friends for their support and encouragement. For my parents who always trust in me and give me their love and support. For my sister who was always there when I needed her. For all my friends who helped me to keep going through many difficulties. Thank you.

Publications

Papers related to the thesis (journals):

- Andrii Zhygmanovskyi and Norihiko Yoshida, "Peer-to-peer Network for Flexible Service Sharing and Discovery", Lecture Notes in Artificial Intelligence, No.8076, Springer (Proceedings of 11th German Conference on Multiagent System Technologies, Koblenz, Germany), pp.152-165 (September, 2013)
- Andrii Zhygmanovskyi and Norihiko Yoshida, "Cloud Service Provisioning Based on Peer-to-Peer Network for Flexible Service Sharing and Discovery", Journal of Computer and Communications, Vol.2, No.10, SCIRP, pp.17-31 (August, 2014)

Papers related to the thesis (conference):

- Andrii Zhygmanovskyi and Norihiko Yoshida, "Distributed Cloud Bursting Model Based on Peer-to-Peer Overlay", Proceedings of The International Workshop on Data-Driven Self-Regulating Systems (in The IEEE 3rd International Conference on Future Internet of Things and Cloud), Rome, Italy, accepted (August, 2015)

Other papers (journals):

- Takuya Yamaguchi, Andrii Zhygmanovskyi, Noriko Matsumoto, and Norihiko Yoshida, "Similarity-Based Content Retrieval in Self-Organizing Peer-to-Peer Networks", British Journal of Mathematics & Computer Science, Vol.5, No.2, pp.456-470 (February, 2015)
- Masaya Miyashita, Andrii Zhygmanovskyi, Noriko Matsumoto, and Norihiko Yoshida, "Mobile Thread Migration for Dynamic and Adaptive Load Distribution in Grid", IAENG International Journal of Computer Science, submitted

Other papers (conferences):

- Yoshihisa Okano, Andrii Zhygmanovskyi, Noriko Matsumoto, and Norihiko Yoshida, "Folksonomy-Based Improvement of Extraction-Based Automatic Summarization", Proceedings of 2nd International Conference on Intelligent Control, Modelling and Systems Engineering, pp.96-102, Boston, U.S.A. (January, 2014)
- Takeshi Moriai, Andrii Zhygmanovskyi, Noriko Matsumoto, and Norihiko Yoshida, "Dynamic Load Balancing in Skip Graph", Proceedings of 2nd International Conference on Intelligent Control, Modelling and Systems Engineering, pp.212-217, Boston, U.S.A. (January, 2014)
- Koki Taguchi, Andrii Zhygmanovskyi, Noriko Matsumoto, and Norihiko Yoshida, "Context Dependent Messages in Delay/Disruption/Disconnection Tolerant Networks", Proceedings of 6th International Conference on Emerging Network Intelligence, pp.1-5, Rome, Italy (August, 2014)
- Kazuki Ono, Andrii Zhygmanovskyi, Noriko Matsumoto, and Norihiko Yoshida, "Resilient Live-Streaming with Dynamic Reconfiguration of P2P Networks", Proceedings of 6th International Conference on Emerging Network Intelligence, pp.6-11, Rome, Italy (August, 2014) **(Best paper award)**

Bibliography

[1] Peter M. Mell and Timothy Grance. The NIST definition of cloud computing. Technical report.

[2] Ismail Ari, Bo Hong, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. Managing flash crowds on the internet. In *Proceedings of the 11th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '03)*, pages 246–249, December 2003.

[3] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N. Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *Proceedings of the 10th International Conference on Algorithms and Architectures for Parallel Processing, Volume Part I*, ICA3PP'10, pages 13–31. Springer-Verlag, 2010.

[4] David Bernstein, Erik Ludvigson, Krishna Sankar, Steve Diamond, and Monique Morrow. Blueprint for the intercloud — protocols and formats for cloud computing interoperability. In *Proceedings of the 2009 Fourth International Conference on Internet and Web Applications and Services*, ICIW '09, pages 328–336. IEEE Computer Society, 2009.

[5] Katarzyna Keahey, Mauricio Tsugawa, Andrea Matsunaga, and Jose Fortes. Sky computing. *IEEE Internet Computing*, 13(5):43–51, 2009.

[6] N. Grozev and R. Buyya. Inter-Cloud Architectures and Application Brokering: Taxonomy and Survey. *Software: Practice and Experience*, 44(3):369–390, 2012.

[7] The demand for hybrid online file sharing solutions. `http://www.esg-global.com/infographics/the-demand-for-hybrid-online-file-sharing-solutions/`. Accessed: 2015-05-19.

[8] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 43–52. IEEE Computer Society, 2010.

[9] Emanuele Carlini, Massimo Coppola, Patrizio Dazzi, Laura Ricci, and Giacomo Righetti. Cloud federations in Contrail. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7155 of *Lecture Notes in Computer Science*, pages 159–168. Springer Berlin / Heidelberg, 2012.

[10] Mohammad Mehedi Hassan, Biao Song, and Eui nam Huh. A market-oriented dynamic collaborative cloud services platform. *Annales des Télécommunications*, 65(11-12):669–688, 2010.

[11] Yin Li, Futai Zou, Zengde Wu, and Fanyuan Ma. Pwsd: A scalable web service discovery architecture based on peer-to-peer overlay network. In *APWeb*, volume 3007 of *Lecture Notes in Computer Science*, pages 291–300. Springer, 2004.

[12] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley.

[13] Andrii Zhygmanovskyi and Norihiko Yoshida. Peer-to-peer network for flexible service sharing and discovery. In *Multiagent System Technologies*, volume 8076 of *Lecture Notes in Computer Science*, pages 152–165. Springer Berlin Heidelberg, 2013.

[14] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160. ACM, 2001.

[15] Jun Gao. *A Distributed and Scalable Peer-to-peer Content Discovery System Supporting Complex Queries*. PhD thesis, Pittsburgh, PA, USA, 2004.

[16] Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. Hypercup: Hypercubes, ontologies, and efficient search on peer-to-peer networks. In *Proceedings of the 1st International Conference on Agents and Peer-to-peer Computing*, pages 112–124. Springer-Verlag, 2003.

[17] Darije Ramljak and Maja Matijašević. Swsd: A p2p-based system for service discovery from a mobile terminal. In *Proceedings of the 9th International Conference on Knowledge-Based Intelligent Information and Engineering Systems - Volume Part III*, pages 655–661. Springer-Verlag, 2005.

[18] Massimo Paolucci, Katia P. Sycara, Takuya Nishimura, and Naveen Srinivasan. Using daml-s for p2p discovery. In *ICWS*, pages 203–207. CSREA Press, 2003.

[19] Cristina Schmidt and Manish Parashar. A peer-to-peer approach to web service discovery. *World Wide Web*, 7(2), June 2004.

[20] Guanling Lee, Sheng-Lung Peng, Yi-Chun Chen, and Jia-Sin Huang. An efficient search mechanism for supporting partial filename queries in structured peer-to-peer overlay. *Peer-to-Peer Networking and Applications*, 5(4), 2012.

[21] Egemen Tanin, Aaron Harwood, and Hanan Samet. Using a distributed quadtree index in peer-to-peer networks. *VLDB Journal*, 16:165–178, 2007.

[22] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[23] Rajiv Ranjan and Liang Zhao. Peer-to-peer service provisioning in cloud computing environments. *The Journal of Supercomputing*, 65(1):154–184, 2013.

[24] Google Cloud Dedicated Gaming Solution. `https://cloud.google.com/developers/articles/dedicated-server-gaming-solution`. Accessed: 2015-05-19.

[25] Dana Petcu, Beniamino Di Martino, Salvatore Venticinque, et al. Experiences in building a mOSAIC of clouds. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(12), 2013.

[26] Ana Juan Ferrer, Francisco Hernández, Johan Tordsson, et al. OPTIMIS: a holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66–77, January 2012.

[27] B. Rochwerger, D. Breitgand, E. Levy, et al. The Reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):535–545, July 2009.

[28] Michael Mattess, Christian Vecchiola, Saurabh Kumar Garg, and Rajkumar Buyya. Cloud bursting: Managing peak loads by leasing public cloud services. In *Cloud Computing: Methodology, Systems, and Applications*. CRC Press, Taylor and Francis Group, LLC.

[29] Christian Vecchiola, Xingchen Chu, and Rajkumar Buyya. Aneka: A Software Platform for .NET-based Cloud Computing. High Speed and Large Scale Scientific Computing, pages 267–295. IOS Press, 2009.

[30] OpenNebula.org. `http://opennebula.org/`. Accessed: 2015-05-19.

[31] Markus Lilienthal. A decision support model for cloud bursting. *Business & Information Systems Engineering*, 5(2):71–81, 2013.

[32] Diane Jordan and John Evdemon. Web services business process execution language version 2.0. Technical report, OASIS, 2007.

[33] YAWL. `http://yawlfoundation.org/`. Accessed: 2015-05-19.

[34] A. Marosi, G. Kecskemeti, A. Kertesz, and P. Kacsuk. FCM: an architecture for integrating IaaS cloud systems. In *Proceedings of the Second International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 7–12, Rome, Italy, 2011.

[35] David Bernstein, Deepak Vij, and Stephen Diamond. An intercloud cloud computing economy — technology, governance, and market blueprints. In *Proceedings of the 2011 Annual SRII Global Conference*, SRII '11, pages 293–299. IEEE Computer Society, 2011.

[36] Andrii Zhygmanovskyi and Norihiko Yoshida. Cloud service provisioning based on peer-to-peer network for flexible service sharing and discovery. *Journal of Computer and Communications*, 2(10):17–31, 2014.

[37] Amazon CloudWatch. `http://aws.amazon.com/cloudwatch/`. Accessed: 2015-05-19.

[38] Google Cloud Monitoring. `https://cloud.google.com/monitoring/`. Accessed: 2015-05-19.

[39] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, June 2013.

[40] Jingxin K. Wang, Jianrui Ding, and Tian Niu. Interoperability and standardization of intercloud cloud computing. *The Computing Research Repository*, abs/1212.5956, 2012.

[41] D. Mills, U. Delaware, and J. Martin. Network time protocol version 4: Protocol and algorithms specification, 6 2010. RFC 5905.

[42] John Eidson. Standard for a precision clock synchronization protocol for networked measurement and control systems, 6 2002. IEEE 1588-2002.