Journal of Computer and Communications, 2014, 2, 17-31
Published Online August 2014 in SciRes. http://www.scirp.org/journal/jcc
http://dx.doi.org/10.4236/jcc.2014.210003

Scientific Research

Cloud Service Provisioning Based on Peer-to-Peer Network for Flexible Service Sharing and Discovery

Andrii Zhygmanovskyi, Norihiko Yoshida

Department of Computer Science, Saitama University, Saitama, Japan
Email: andrew@ss.ics.saitama-u.ac.jp, yoshida@ss.ics.saitama-u.ac.jp

Received 8 June 2014; revised 8 July 2014; accepted 6 August 2014

Abstract

In this paper, we present an approach to establish efficient and scalable service provisioning in the cloud environment using P2P-based infrastructure for storing, sharing and discovering services. Unlike most other P2P-based approaches, it allows flexible search queries, since all of them are executed against internal database presenting at each overlay node. Various issues concerning using this approach in the cloud environment, such as load-balancing, queuing, dealing with skewed data and dynamic attributes, are addressed in the paper. The infrastructure proposed in the paper can serve as a base for creating robust, scalable and reliable cloud systems, able to fulfill client's QoS requirements, and at the same time introduce more efficient utilization of resources to the cloud provider.

Keywords

Peer-to-Peer, Cloud Computing, Service Provisioning, Service Discovery, Service Sharing

1. Introduction

Service-oriented computing nowadays is a major approach for constructing and managing systems of different complexity, starting from one-service applications, such as URL shortening services or online documents converters, and up to real-time systems with millions of users. Originally present mostly in the form of Web Services with centralized discovery and management methods, at the moment many applications can be seen as services employing interoperability standards and frameworks, as well as featuring flexible billing models and public APIs. Since service-oriented computing improves the productivity of programming and administering applications in increasingly complex scenarios, there is a necessity for sophisticated communication and coor-

How to cite this paper: Zhygmanovskyi, A. and Yoshida, N. (2014) Cloud Service Provisioning Based on Peer-to-Peer Network for Flexible Service Sharing and Discovery. *Journal of Computer and Communications*, **2**, 17-31.
http://dx.doi.org/10.4236/jcc.2014.210003

dination protocols as well as scalable and flexible means for implementing all stages of lifetime of services, including their deployment, discovering and managing.

Being extensively used by enterprise, service-oriented computing is usually implemented by utilizing another driving force for the modern IT—cloud computing. Defined by NIST as "*a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [1], cloud computing 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 become 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 introduce 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. Having already established itself as invaluable technology, cloud computing, however, is constantly evolving and in the next few years the focus of cloud computing is expected to shift from building the infrastructure—today's main front of competition among the vendors—to the application domain, therefore increasing importance of delivering new ideas for efficient management of services deployed in the cloud.

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. This, in turn, comes from traditional architectural model designed for Web Services which defines three types of participants [2]: 1) *service providers*, which create and own services, and advertise them to potential users; 2) *service brokers*, which maintain a registry of advertised services and might introduce service providers to service requestors; 3) *service requestors*, which search the registries of service brokers for suitable service providers and then contact a service provider to use its services. Due to its nature, 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.

Despite the fact that P2P 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 P2P-based service discovery networks include Hypercube [3] and Speed-R. Traditionally, many proposed solutions are based on Semantic Web approach for describing services. For instance, in the approach shown in [5], services are described using DAML-S, while service publishing and discovery mechanism is based on JXTA technology. Similar approach is used in [5], but Gnutella P2P overlay is used instead of JXTA. Generally speaking, all P2P-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 [6], sliding window partition method in [7] or tree-based solutions such as MX-CIF quad tree in [8] to name a few.

In this paper, we propose a Web Service discovery architecture based on structured P2P 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

[9] and Intentional Name System, described in detail in [10]. 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 paper 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.

The rest of the paper is organized as follows: Section 2 presents the architecture of proposed service sharing and discovery network including service description, service storing and service discovery mechanisms. Section 3 presents the proposed way of applying this mechanism to the service provisioning in the cloud environment, along with outline of related main challenges. We describe experimental framework and present experimental results in Section 4. Conclusions and further research directions are given in the Section 5.

2. Service Sharing and Discovery Network

2.1. Overview

The idea of building service sharing and discovery network based on P2P 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 P2P-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 our earlier paper [11] 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 [10] 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 [12], the approach for storing and discovering service descriptions proposed in this paper 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.

The approach described above bears some similarities with distributed scalable content discovery system,

[1]We use the term "vertex" meaning "node of a graph" throughout this paper, in order to avoid confusing it with terms "overlay node" or "cloud node".

proposed in [13] in that both use attribute-value based content registration and discovery mechanism (influenced by [10]), and propose similar mechanism of storing multidimensional data in the P2P 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.

2.2. Service Description

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, 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 1**. Here we can see an example of a service from the domain, explained in detail in Section 4.4. 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 [14]: *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 [14]:

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

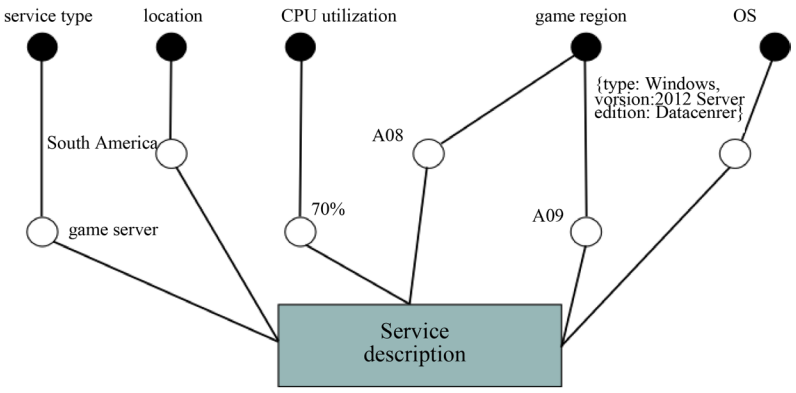

Figure 1. (*a*, *v*)-graph for service description.

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 3.4.

2.3. Service Description Storing

In the approach, proposed in this paper, Chord DHT peer-to-peer overlay is used to store service descriptions in a distributed manner. 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 2**. 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: $P.OS = \left(V_V \cup V_A \cup \{sd\}, E_{A,V} \cup E_{V,SD}\right)$ and $P.SS = \left(V_V \cup V_A \cup SD, E_{A,V} \cup E_{V,SD}\right)$, 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 $\left(v_A, v_V\right) | v_A \in V_A, v_V \in V_V$ and $E_{V,SD}$—set of edges $\left(v_V, sd^*\right) | v_V \in V_V, sd^* \in SD \vee sd^* = sd$. 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: 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**. First, we evaluate hash function against each attribute name in the service description. Having found the node,

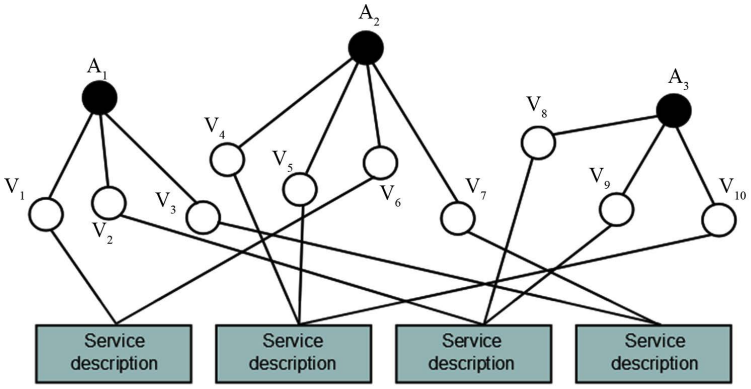

Figure 2. Merged (*a*, *v*)-graph.

```
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. Service description storing algorithm.

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.

2.4. Service Discovery

Among one of the most significant drawbacks of DHT-based P2P 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 paper is based on the way service descriptions are stored in the overlay network, that is, using merged (a,v)-graph.

Firstly, we assume that each search query in the system is submitted in the form, shown in **Figure 4** or can be represented as such. Here we let $op_i(A_i)$ be operators $=,\neq,\text{contains},>,<$ etc., defined on the sequence of attributes A_i as parameters, A_i—sequence of attributes $[a_1,a_2,\cdots,a_n]$ (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 P2P networks. You can see a concrete example of the query in **Figure 5**. 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 algorithm. The query issued by the node in the form shown in **Figure 3** can be represented as a graph $QU=(Q\cup LOP,E)$, where $Q=\{q_i=(a_i,op_i)\mid i=1,\cdots,n\}$—the set of query terms, $LOP=\{lop_i\mid i=1,\cdots,n\}$—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 3.3, 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 $R=\{r_i\mid i=1,\cdots,m\}$—initially empty set of results obtained from nodes after evaluating a query term, $LOP=\{lop_i\mid i=1,\cdots,n\}$—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 6**. 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 P2P overlay.

4) Each attribute expression in the attribute group is evaluated against the data in merged (a,v)-graph stored

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

Figure 4. Generic search query format.

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

Figure 5. Example of search query.

foreach $q \in QU.Q$
  $H := \mathbf{h}(q.a)$;
  $P := \mathbf{find}(H)$;
  $R := P.\mathbf{evaluate}(q)$;
  $RES.R := RES.R \cup \{R\}$ **for corresponding** $lop \in LOP$;
end foreach
return $r_1 \quad C[lop_1] \quad r_2 \quad C[lop_2] \ \ldots\ C[lop_{n-1}] r_n$
  where $r_i \in RES.R,\ lop_i \in RES.LOP$;

Figure 6. Search query routing and execution algorithm.

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.

3. Application to Cloud Computing

3.1. Overview

The process of deploying application services on clouds is known as *cloud provisioning* and consists of three primary steps [15]:

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.

In this section we outline a service provisioning approach which utilizes service sharing and discovery mechanism described in Section 2. 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.

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 [15].

• 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.

3.2. 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 [13] 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 [13].

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.

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 3.3). Next, the node must decide which of the remaining services is returned to the requestor based on the principle 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.

3.3. 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 7**), 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

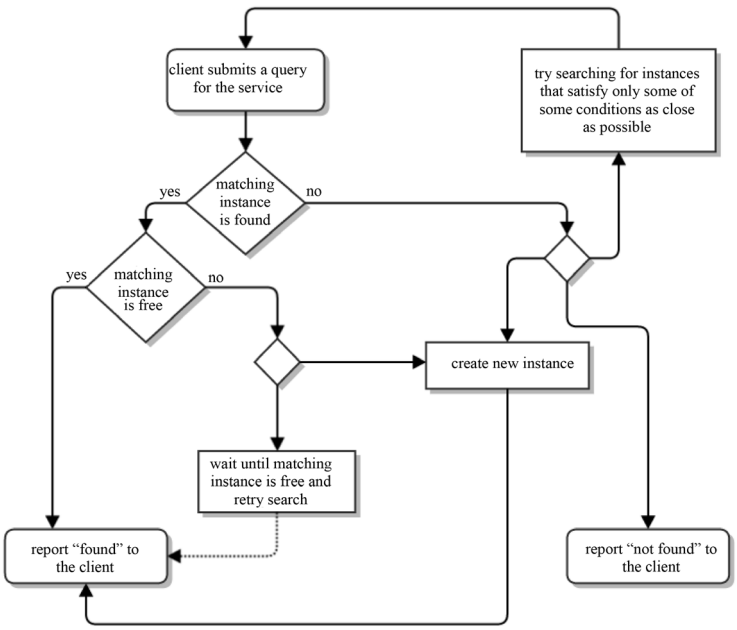

Figure 7. Service discovery in the cloud: client request flow.

to the action 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.

3.4. 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 3.3.

4. Evaluation and Experimental Results

In this section we present experiment environment and methodology used to evaluate an approach, presented in Section 2. At the moment, the implemented solution is still not fully adapted to run on the cloud instances, but designed to show soundness of the (a,v)-graph based approach to adequately address the problem of partitioning of multidimensional service description space over the set of nodes and to make their services discoverable.

4.1. Simulator Implementation

The proof-of-concept implementation of the service description and sharing framework described in the Section 2 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 [12] 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 Java-

Script/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 (http://msdn.microsoft.com/en-us/library/dd456779(v=vs.110).aspx), 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 **Table 1** and **Table 2**.

4.2. 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 3** shows all parameters and notations used in the simulation.

4.3. 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 $\lambda = 50\ \text{reg/s}$. Maximum (a, v)-pair registrations per node is chosen to be $T_r = 100$ and threshold $R_r = 30\ \text{reg/s}$.

Table 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 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

Table 3. Parameters and notations used in the simulation.

Symbol	Meaning
Network	
NC	Number of nodes in the overlay network
NA	Number of possible attributes in the system
NV	Number of possible values for each attribute in the system
NAV	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 variance of $E[a]$
$Var(a)$	
Thresholds	
Rr	Maximum registration rate
Rq	Maximum query rate
Tr	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 8 shows how registration success rate changes depending on how many services were fed to the system. According to Section 4.2, 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 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 9**, even for the unrealistic uniform query dataset case, query success rate is 100% only for the small amount of queries and it steadily drops down to the 50% as the query load increases. When

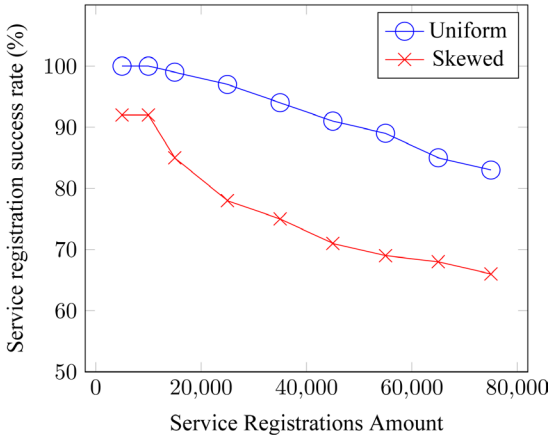

Figure 8. Service registration success rate comparison.

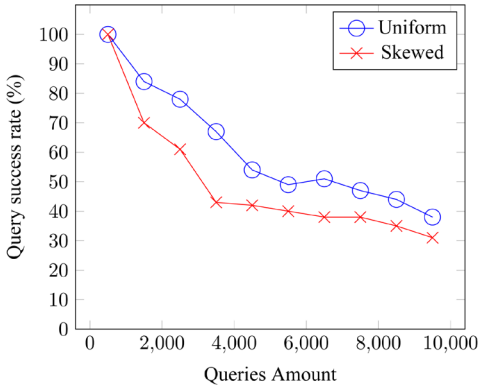

Figure 9. Query success rate comparison.

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 3.3 (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 experiments shown in [13], LBM (Load Balancing Matrix) approach introduced in Section 3.2 has desired features in this regard, therefore we consider its usage as a load-balancing mechanism for our approach to be appropriate as a part of our future research.

4.4. Example Domain

To further demonstrate the system operation, we chose a domain of dedicated gaming solution based on Google Cloud Platform (https://cloud.google.com/developers/articles/dedicated-server-gaming-solution), 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 micropayments 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. Also, on the current stage of the experimentation, we included some representative dynamic attributes (such as current hard disk space or network latency) as a part of static service description. During later stages of research, they will be properly simulated as truly dynamic ones according to the approaches given in Section 3.4. The possible example of service descriptions in given domain is shown in **Figure 10**.

```
<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 10. Node services data example.

5. Conclusions and Future Work

In this paper, we presented a preliminary study on the problem of building scalable and robust service provisioning platform in the cloud environment. To this end, we proposed an application of P2P technology to solve the problem of efficient services sharing and discovering in a decentralized way. The main advantages of proposed approach are 1) utilizing of well-known DHT, which guarantees the correctness and soundness of underlying P2P overlay, 2) partitioning of multidimensional service description space over the set of nodes that naturally allows execution of range queries, and 3) modular framework architecture which allows using wide range of overlays and service description storage mechanisms. Also, we proposed organizing services that run on cloud node instances into DHT-based overlay, and utilize this approach to achieve fully decentralized scalable and self-managing service discovery and load-balancing. We argue that introducing this approach to the traditional cloud systems architecture will lead to lowering management costs and maintaining higher levels of availability to support system SLA even in cases of severe failure.

As a part of future work, we intend to enhance existing implementation of the system, so that it could be deployed on cloud instances and managed through well defined API or/and graphic interface. The issues that have yet to be addressed in future research are stated in detail in Section 3. Another important challenge in building reliable peer-to-peer overlay over the cloud infrastructure is to explore more efficient ways to ensure optimal load-balancing and scheduling. To address these issues we will investigate existing strategies proposed by leading cloud providers and adapt them to our approach.

References

[1] Mell, P. and Grance, T. (2011) The NIST Definition of Cloud Computing. NIST Special Publication.

[2] Singh, M.P. and Hunhs, M.N. (2005) Service-Oriented Computing: Semantics, Processes, Agents. Wiley, Chichester.

[3] Schlosser, M., Sintek, M., Decker, S. and Nejdl, W. (2002) HyperCuP—Hypercubes, Ontologies and Efficient Search on P2P Networks. *Agents and Peer-to-Peer Computing*, **2530**, 112-124. http://dx.doi.org/10.1007/3-540-45074-2_11

[4] Ramljak, D. and Matijašević, M. (2005) SWSD: A P2P-Based System for Service Discovery from a Mobile Terminal. *Knowledge-Based Intelligent Information and Engineering Systems*, **3683**, 655-661.

[5] Paolucci, M., Sycara, K., Nishimura, T. and Srinivasan, N. (2003) Using DAML-S for P2P Discovery. *Proceedings of International Conference on Web Services* (*ICWS*03), Las Vegas, 23-26 June 2003.

[6] Schmidt, C. and Parashar, M. (2003) A Peer-to-Peer Approach to Web Service Discovery. *World Wide Web*, **7**, 211-229. http://dx.doi.org/10.1023/B:WWWJ.0000017210.55153.3d

[7] Lee, G., Peng, S.L., Chen, Y.Ch. and Huang, J.S. (2012) An Efficient Search Mechanism for Supporting Partial Filename Queries in Structured Peer-to-Peer Overlay. *Peer-to-Peer Networking and Applications*, **5**, 340-349.

[8] Tanin, E., Harwood, A. and Samet, H. (2007) Using a Distributed Quadtree Index in Peer-to-Peer Networks. *The VLDB Journal*, **16**, 165-178.

[9] Li, Y., Zou, F., Wu, Zh. and Ma, F. (2004) PWSD: A Scalable Web Service Discovery Architecture Based on Peer-to-Peer Overlay Network. *Advanced Web Technologies and Applications*, **3007**, 291-300.

[10] Adjie-Winoto, W., Schwartz, E., Balakrishnan, H. and Lilley, J. (1999) The Design and Implementation of an Intentional Naming System. *Proceedings of the* 17*th ACM Symposium on Operating Systems Principles*, *Kiawah Island Resort*, 12-15 December 1999, 186-201.

[11] Zhygmanovskyi, A. and Yoshida, N. (2013) Peer-to-Peer Network for Flexible Service Sharing and Discovery. *Multiagent System Technologies*, **8076**, 152-165.

[12] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F. and Balakrishnan, H. (2001) Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *Proceedings of ACM SIGCOMM*'01, San Diego, 27-31 August 2001, 149-160.

[13] Gao, J. and Steenkiste, P. (2006) Design and Evaluation of a Distributed Scalable Content Discovery System. *IEEE Journal on Selected Areas in Communications*, **22**, 54-66.

[14] Samet, H. (2006) Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann Publishers, USA.

[15] Ranjan, R. and Zhao, L. (2013) Peer-to-Peer Service Provisioning in Cloud Computing Environments. *The Journal of Supercomputing*, **65**, 154-184.