Service Level Agreement parameter matching in Cloud Computing

Tejas Chauhan

MEFGI, Rajkot, India
Email: tejas.chauhan@-marwadieducation.edu.in

Sanjay Chaudhary, Vikas Kumar and Minal Bhise

DA-IICT, Gandhinagar, India
Email: {sanjay_chaudhary, vikas_kumar, minal_bhise}@daiict.ac.in

Abstract

Cloud is a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). It provides an on-demand, pay-as-you-go computing resources and had become an alternative to traditional IT Infrastructure. As more and more consumers delegate their task to cloud providers, Service Level Agreement (SLA) between consumer and provider becomes an important aspect. Due to the dynamic nature of cloud the matching of SLA templates need to be dynamic and continuous monitoring of Quality of Service (QoS) is necessary to enforce SLAs. SLA template contains many parameters like cloud's resources (physical memory, main memory, processor speed etc.) and properties (availability, response time etc.). This work addresses the issue of matching SLA parameters to find suitable cloud provider for particular application.

Index Terms

Cloud Computing, Service Level Agreement, Cloud Capability Model, Application Requirement Model

1. Introduction

The relationship between the cloud provider and the cloud consumer must be described with a Service Level Agreement. Because cloud consumers trust cloud providers to deliver some of their infrastructure services, it is vital to define those services, how they are delivered and how they are used. A Service Level Agreement (SLA) is part of the contract between the service consumer and service provider and formally defines the level of service. SLA between consumer and provider becomes an important aspect.

An SLA contains several parameters like a set of properties and resources the provider will deliver, a specific definition of each resource, the responsibilities of the provider and the consumer, a resource quantity, an auditing mechanism to monitor the service etc. [1].

Although managing and monitoring the quality levels of services rely heavily on automated tools, at present the actual Cloud SLAs are typically plain-text documents, and sometimes an informative document published online. An example of a document with legal obligations is the Amazon S3 Service Level Agreement [2], Amazon EC2 Service Level Agreement [3].

Finding correspondences between multiple elements is required in many application scenarios and is often referred to as matching. This work aims to define the process of identifying compatible cloud provider for a given requirements by matching SLA parameters. This work will be contributed in 'Task B2: Compatibility calculation module' of Middleware for the cloud shown in Figure 1. This middleware is a part of the project "Cirrocumulus: A Semantic Framework for Application and Core Services Portability across Heterogeneous Clouds" [4] undergoing at Kno-e-sis Center at Wright State University.

SLA parameters of the cloud are defined in a cloud capability model and application's requirements are defined in an application requirement model. Cloud capability model and application requirement model are defined in Resource Description Framework (RDF) format [5]. RDF is a W3C standard for describing Web resources. RDF is written in XML and it is a W3C Recommendation. To identify the compatible cloud provider model/graph matching algorithm is used.

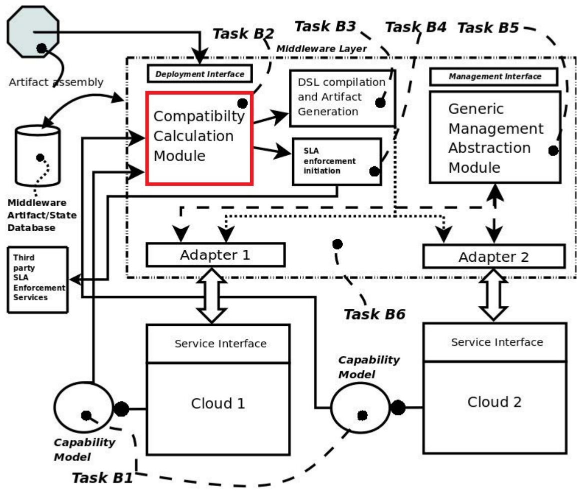

Figure 1. Functionality of the Middleware Layer in Cirrocumulus

2. Related Work

At present the actual Cloud SLAs are typically plain-text documents, and sometimes an informative document published online. An example of a document with legal obligations is the Amazon S3 Service Level Agreement [2], Amazon EC2 Service Level Agreement [3]. Consumer needs to manually match application requirements with each and every cloud provider to identify compatible cloud.

Proposed algorithm identifies the compatible cloud provider. It gives suggestion to a consumer in terms of number of matched instances.

3. Proposed Approach

Before going to the details of matching algorithm, let us briefly walk through an approach to match capability model and requirement model.

The first step is to define cloud model (RDF file) that contains cloud resources, properties, resource quantity etc. and requirement model (RDF File) that contains application's required resources and its quantity. Let's consider a Cloud model as 'ModelC' and Requirement model as 'ModelR'.

Second step is to convert these models to graph/-DOM [6]. Jena APIs are used here to convert these RDF models to a Graph structure (Jena Model [7]). An example Graph model is depicted in Figure 2.

Third step is to find Pairwise Connectivity Graph and using this Pairwise Connectivity Graph an Induced Propagation Graph is calculated. Pairwise Connectivity Graph is defined by: source and target nodes are merged if they have the same property edge.

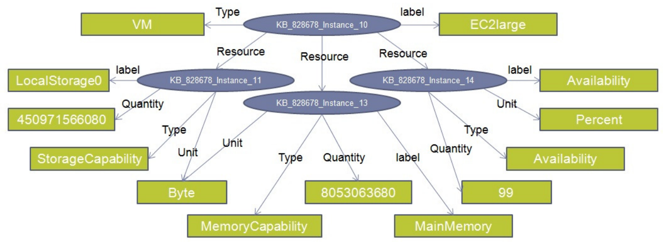

Figure 2. Graph Model of ModelC

Each node in the Pairwise Connectivity Graph is an element from $ModelC \times ModelR$. Let's call such node $map\,pairs$.

A Propagation Graph is an auxiliary data structure derived from two models and in Propagation Graph each edge is given a weight. These weights are used in the fixpoint computation in the algorithm.

Forth step is to find initial mapping between two models using RDF Schema. For initial mapping nodes of two models are compared as if they are of subclass, superclass or equivalence in the RDF Schema file. This initial mapping is used in similarity flooding algorithm. A portion of the initial mapping is shown in Table 1.

Table 1. A portion of initial mapping (5 of total 16 entries are shown)

Similarity	Node in ModelR	Node in ModelC
1.0	KB_828678_Instance_101	KB_828678_Instance_11
0.0	Availability	VM
0.0	StorageCapability	MemoryCapability
1.0	Byte	Byte
1.0	KB_828678_Instance_103	KB_828678_Instance_13

Subsequent step is to apply similarity flooding algorithm [8] to match two models. Similarity flooding algorithm is applied on two graph models which are combined together into Pairwise connectivity graph which intern converted to Similarity propagation graph.

Result of the Similarity flooding algorithm is then filtered out to only the instance node pairs that have similarity value greater than zero. As we can see in Table 2, this approach is able to produce a good mapping between cloud capability model ($ModelC$) and application requirement model ($ModelR$).

4. Algorithms

In this section, an algorithm is explained using a simple example presented in Figure 3. The upper part of the figure shows two models A and B that we want to match.

Table 2. Mapping after filtering the result

Similarity	Node in ModelR	Node in ModelC
1.0	KB_828678_Instance_100 [VM]	KB_828678_Instance_10 [VM]
0.47619045	KB_828678_Instance_103 [Availability]	KB_828678_Instance_13 [Availability]
0.44047618	KB_828678_Instance_101 [StorageCapability]	KB_828678_Instance_11 [StorageCapability]
0.44047618	KB_828678_Instance_102 [MemoryCapability]	KB_828678_Instance_12 [MemoryCapability]
0.08333333	KB_828678_Instance_101 [StorageCapability]	KB_828678_Instance_12 [MemoryCapability]
0.08333333	KB_828678_Instance_102 [MemoryCapability]	KB_828678_Instance_11 [StorageCapability]
0.04761905	KB_828678_Instance_101 [StorageCapability]	KB_828678_Instance_13 [Availability]
0.04761905	KB_828678_Instance_103 [Availability]	KB_828678_Instance_12 [MemoryCapability]
0.04761905	KB_828678_Instance_103 [Availability]	KB_828678_Instance_11 [StorageCapability]
0.04761905	KB_828678_Instance_102 [MemoryCapability]	KB_828678_Instance_13 [Availability]

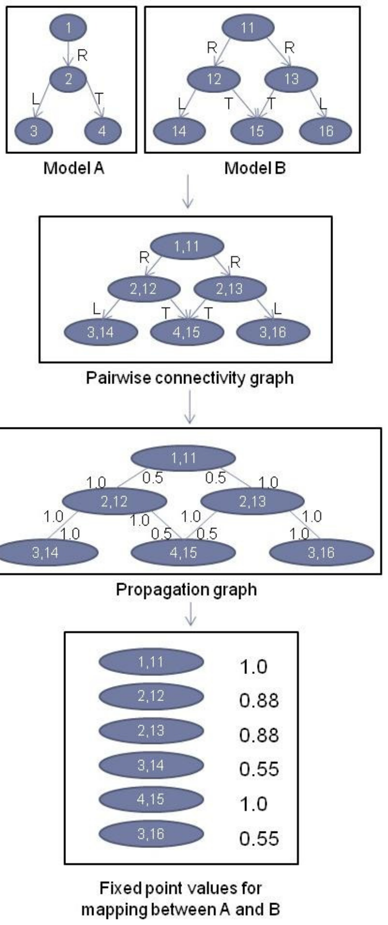

Figure 3. Example illustrating the Algorithm

4.1. SLAParameterMatching

SLAParameterMatching is the procedure that takes cloud models and requirement model as an input and gives an suggestion in terms of ResultList.

Algorithm 1 SLAParameterMatching

```
INPUT: SchemaFile / OWL Ontology, Requirement Model, Cloud Models, n (iteration count)
OUTPUT: ResultList
1: Load Schema file / OWL Ontology
2: Load Requirement Model
3: for all Cloud Model do
4:     Load Cloud Model
5:     Call calcPairwiseConnectivityGraph
6:     Call calcInducedPropagationGraph
7:     Call findInitialMapping
8:     for $i = 1$ to n do
9:         Call calcFixpointValue
10:        Call normalizeFixpointValue
11:    end for
12:    Call filterMatchingPairs
13:    Add to resultList
14: end for
```

4.2. calcPairwiseConnectivityGraph

Pairwise connectivity graph (PCG) is defined as follows:

$$((x, y), p, (x', y') \in PCG(A, B) \Leftrightarrow (x, p, x') \in A \, and \, (y, p, y') \in B$$

The connectivity graph for our example is shown in Figure 3.The intuition behind arcs that connect map pairs is the following. Consider for example map pairs (1, 11) and (2, 12). If 1 is similar to 11, then probably 2 is somewhat similar to 12. The evidence for this conclusion is provided by the R-edges that connect 1 to 2 in graph A and 11 to 12 in graph B. This evidence is captured in the connectivity graph as R-edge leading from (1, 11) to (2, 12). We call (1, 11) and (2, 12) neighbours.

4.3. calcInducedPropagationGraph

The induced propagation graph for A and B is shown next to the connectivity graph in Figure 3. For every edge in the connectivity graph, the propagation graph contains an additional edge going in the opposite direction. The weights are placed on the edges of the

Algorithm 2 calcPairwiseConnectivityGraph

```
INPUT: Cloud Model, Requirement Model
OUTPUT: PCGModel
 1: Create temporary model tmpModel
 2: List all statements from Cloud Model
 3: for all statements in Cloud Model do
 4:    Get cloudSubject, cloudPredicate, cloudObject
 5:    List all statements from Requirement Model
 6:    for all statements in Requirement Model do
 7:       Get reqSubject, reqPredicate, reqObject
 8:       if reqPredicate != cloudPredicate then
 9:          Continue
10:       end if
11:       Combine cloudSubject and reqSubject to subject
12:       Combine cloudObject and reqObject to object
13:       Add statement(subject, cloudPredicate, object) to tmpModel
14:    end for
15: end for
16: return tmpModel
```

propagation graph indicate how well the similarity of a given map pair propagates to its neighbors and back. These so-called propagation coefficients range from 0 to 1. The approach illustrated in Figure 3 is based on the intuition that each edge type makes an equal contribution of 1.0 to spreading of similarities from a given map pair.

For example, there is exactly one L-edge going out from (2, 12) in the connectivity graph. In such case we set the coefficient $w((2,12),(3,14))$ in the propagation graph to 1.0. The value 1.0 indicates that the similarity of 2 to 12 contributes fully to that of 3 and 14. Analogously, the propagation coefficient $w((3,14),(2,12))$ on the reverse edge is also set to 1.0, since there is exactly one incoming L-edge for (2,12). In contrast, two R-edges are leaving map pair (1,11) in the connectivity graph. Thus, the weight of 1.0 is distributed equally among $w((1.11),(2,12)) = 0.5$ and $w((1,11),(2,13)) = 0.5$.

4.4. findInitialMapping

Let $\sigma(x,y)$ be the similarity measure of nodes $x \in A$ and $y \in B$ defined as a total function over $A \times B$. We refer to σ as a mapping. Mapping σ^0 represents the initial similarity between nodes of $A\,and\,B$, which is obtained e.g., using subclass, superclass or equivalence property in the RDF Schema file.

Algorithm 3 calcInducedPropagationGraph

```
INPUT: PCGModel
OUTPUT: IPGModel
 1: Create temporary model tmpModel1
 2: List all statement from PCGModel
 3: for all statements in PCGModel do
 4:    Get pcgSubject, pcgPredicate, pcgObject
 5:    Add statement (pcgSubject, pcgPredicate, pcgObject) to tmpModel1
 6:    Add statement (pcgObject, pcgPredicate, pcgSubject) to tmpModel1
 7: end for
 8: Create temporary model tmpModel2
 9: List all statements from tmpModel1
10: for all statements in tmpModel1 do
11:    Add statement to tmpModel2
12:    Get subject and predicate from statement
13:    Create property 'predicate+weight' for tmpModel2
14:    Calculate weight for property
15:    Add statement (subject, 'predicate+weight', weight) to tmpModel2
16: end for
17: return tmpModel2
```

4.5. calcFixpointValue

The similarity flooding algorithm [8] is based on an iterative computation of $\sigma - values$. Let σ^i denote the mapping between $A\,and\,B$ after i^{th} iteration then mapping $\sigma^{(i+1)}$ is computed from mapping σ^i as follows:

$$\begin{aligned}\sigma^{(i+1)}(x,y) = {} & \sigma^i(x,y) \\ & + \sum_{(a_u,p,x)\in A,(b_u,p,y)\in B} \sigma^i(a_u,b_u)w((a_u,b_u),(x,y)) \\ & + \sum_{(x,p,a_v)\in A,(y,p,b_v)\in B} \sigma^i(a_v,b_v)w((a_v,b_v),(x,y))\end{aligned}$$

where,

$\sigma^{(i+1)}(x,y)$ is a mapping between x and y at $(i+1)^{th}$ iteration

$\sigma^i(x,y)$ is a mapping between x and y at i^{th} iteration

$\sigma^i(a_u,b_u)$ is a mapping between a_u and b_u at i^{th} iteration

$\sigma^i(a_v,b_v)$ is a mapping between a_v and b_v at i^{th} iteration

$w((a_u,b_u),(x,y))$ is a weight of the edge between (a_u,b_u) and (x,y)

Algorithm 4 findInitialMapping

```
INPUT: IPGModel
OUTPUT: FPVGModel
 1: Create temporary model tmpModel
 2: Create property SimilarityMeasure for tmpModel
 3: List all subjects from IPGModel
 4: for all subjects in IPGModel do
 5:   List all statements of subject in IPGModel
 6:   for all statements of subject in IPGModel do
 7:     Add this statement to tmpModel
 8:     Get predicate and object
 9:     if predicate is 'type' pradicate then
10:       Split subject and object to original two
11:       if both subjects are equal or both objects
          are equal then
12:         initialMeasure = 1.0
13:       else
14:         InitialMeasure = 0.0
15:       end if
16:     else
17:       initialMeasure = 0.0
18:     end if
19:   end for
20:   Add statement (subject, Similarity, initialMea-
      sure) to tmpModel
21: end for
22: return tmpModel
```

$w((a_v, b_v), (x, y))$ is a weight of the edge between (a_v, b_v) and (x, y)

4.6. normalizeFixpointValue

At the end of iteration, normalization is performed to maintain $\sigma - values$ between 0 and 1. For this reason the $\sigma - value$ of each node is divided by the larger $\sigma - value$ of that iteration.

4.7. filterMatchingPairs

This procedure filters the result to only $mappairs$ of instance nodes that has $\sigma - value$ greater than zero. It will simply discard the $mappairs$ that has $\sigma - value$ zero or does not contain an instance node.

5. Experiments and Results

5.1. Experimental setup

Proposed algorithm is implemented in JAVA. For analysis, 16 clouds and 10 different application requirements with SLA parameters as listed below were considered.

Algorithm 5 calcFixpointValue

```
INPUT: FPVGModel
OUTPUT: resulting FPVGModel
 1: Create temporary model tmpModel
 2: List all subjects from FPVGModel
 3: for all subjects of FPVGModel do
 4:   List all statements of subject in FPVGModel
 5:   for all statements of subject in FPVGModel do
 6:     Get predicate and object
 7:     if predicate is SimilarityMeasure then
 8:       SMi = object
 9:       Continue
10:     end if
11:     if predicate is predicateWeight then
12:       Add statement to tmpModel
13:       Continue
14:     end if
15:     Add statement to tmpModel
16:     Get statement1 having SimilarityMeasure as
        predicate and object as subject from FPVG-
        Model
17:     Get statement2 having weight as predicate
        and object as subject from FPVGModel
18:     IN += object of statement1 * object of state-
        ment2
19:   end for
20:   Add statement (subject, SimilarityMeasure, SMi
      + IN) to tmpModel
21: end for
22: return tmpModel
```

- Virtual Machine (VM)
- Storage Capability
- Memory Capability
- Ethernet port
- Availability
- Processor Speed
- Response Time
- Server Reboot Time
- Service Credit

Table 3 shows the instance count of Cloud SLA parameters that are declared in cloud capability model. Table 4 shows the instance count of Application SLA parameters that are declared in application requirement model.

5.2. Results

Similarity measure of two instances defines how much similar those two instances are. Its value range between 0 and 1. Value 0 indecate mismatch of the

Algorithm 6 normalizeFixpointValue

```
INPUT: FPVGModel
OUTPUT: resulting FPVGModel
1: Get large fixpoint value from FPVGModel to largeSM
2: Create temporary model tmpModel
3: List all statements of FPVGModel
4: for all statements of FPVGModel do
5:   Get subject, predicate, object
6:   if predicate is SimilarityMeasure then
7:     SM = object / largeSM
8:     Add statement (subject, predicate, SM) to tmpModel
9:   else
10:    Add statement (subject, predicate, object) to tmpModel
11:  end if
12: end for
13: return tmpModel
```

Algorithm 7 filterMatchingPairs

```
INPUT: FPVGModel
OUTPUT: filteredGraph
1: Create temporary model tmpModel
2: List all subjects of FPVGModel
3: for all subjects of FPVGModel do
4:   if subject is an instance then
5:     Get all statements of subject from FPVGModel
6:     Add all statements to tmpModel
7:   end if
8: end for
9: return tmpModel
```

instances whereas value greater than 0 gives measure of similarity between two instances. Table 5 shows the number of instances matched having similarity measure value greater than 0.1.

A cloud having more number of matched instances will be suggested for perticular application requirement. Table 5 shows the suggested cloud for perticular applications.

In case of multiple clouds having same number of instances matched, the least similarity measure value of each cloud were considered and a cloud having largest similarity measure value amongst them becomes the suggested cloud. If we consider Requirement1, largest number of instance match count is 6 which appear at AT&T and IBM. Table 6 and Table 7 shows the similarity measure values of instances with AT&T cloud and IBM cloud respectively.

Table 3. Instance count of Cloud SLA Parameters

Cloud Capability Model	VM	StorageCapability	MemoryCapability	Ethernet	Availability	ProcessorSpeed	ResponceTime	ServerReboot	ServiceCredit
3Tera	1	1	1		1	1			
AmazonEC2	1	2	1	1	1	1			
AmazonS3	1	2	1	1	1	1			2
AT&T	1	2	2	1	1	1			
AzureStorage	1	2	1	1	1	1	1		2
CloudScale	1	1	1		1	1	1		
CloudSwitch	1	1	1		1	1	1	1	
CloudWorks	1	1	1	2	1	1	1	1	
enStratus	1	1	1	1	1	1			
GoGrid	1	1	1	1	1	1	1		2
IBM	1	2	2	1	1	1	1	1	
OrangeSpace	1	1	1		1	1	1		
OxygenCloud	1	1	1		1	1		1	
Rackspace	1	2	1		1	1	1		1
ReliaCloud	1	1	1	1	1	1	1		
VMware	1	1	1	2	1	1		1	3

Table 4. Instance count of Application SLA Parameters

Application Requirement Model	VM	StorageCapability	MemoryCapability	Ethernet	Availability	ProcessorSpeed	ResponceTime	ServerReboot	ServiceCredit
Requirement1	1	1	1	1					
Requirement2	1	1	1	1	1				
Requirement3	1	2	1	1	1	1			
Requirement4	1	1	1	1	1	1	1		
Requirement5	1	1	1	1	1	1	1	1	
Requirement6	1	1	1	1	1	1	1	1	1
Requirement7	1	1	1	1	1	1	1	1	1
Requirement8	1	2	1	1	1	1	1		1
Requirement9	1	2	1	1	1	1	1		
Requirement10	1	2	2	2	1	1	1	1	1

A least similarity measure value with AT&T cloud is 0.2711 and with IBM is 0.2664 which indicates that AT&T cloud has the larger similarity measure value for 6th matching instance. Hence AT&T cloud will be the suggested cloud for Requirement1.

6. Conclusion

A simple structural algorithm is proposed here which is based on fixpoint computation and useful for matching SLA parameters to identify compatible cloud provider. Also advantage of semantics is considered here by using RDF. This algorithm helps user to

Table 5. # of instances matched

# of instances matched																
	3Tera	AmazonEC2	AmazonS3	AT&T	AzureStorage	CloudScale	CloudSwitch	CloudWorks	enStratus	GoGrid	IBM	OrangeSpace	OxygenCloud	Rackspace	ReliaCloud	VMware	Suggested Cloud
Requirement1	3	5	5	6	5	3	3	5	4	4	6	3	3	4	4	5	AT&T
Requirement2	4	6	6	7	6	4	4	6	5	5	7	4	4	5	5	6	AT&T
Requirement3	6	9	9	10	9	6	6	8	7	7	10	6	6	8	7	8	AT&T
Requirement4	5	7	7	8	8	6	6	8	6	7	9	6	5	7	7	7	IBM
Requirement5	5	7	7	8	8	6	7	9	6	7	10	8	6	7	7	8	IBM
Requirement6	5	7	9	8	10	6	7	9	6	9	10	4	6	8	7	11	VMware
Requirement7	5	7	9	8	10	6	7	9	6	9	10	6	6	8	7	11	VMware
Requirement8	6	9	11	10	12	7	7	9	7	10	11	7	6	10	8	11	AzureStorage
Requirement9	6	9	9	10	10	7	7	9	7	8	11	7	6	9	8	8	IBM
Requirement10	7	11	13	13	14	8	9	13	9	12	15	8	8	11	10	12	IBM

Table 6. Similarity measure values between Requirement1 and AT&T cloud instances

Similarity	Node in Requirement1	Node in AT&T
1.0000	KB_828678_Instance_200 [VM]	KB_828678_Instance_40 [VM]
0.3544	KB_828678_Instance_203 [Ethernet]	KB_828678_Instance_45 [Ethernet]
0.2711	KB_828678_Instance_202 [MemoryCapability]	KB_828678_Instance_44 [MemoryCapability]
0.2711	KB_828678_Instance_202 [MemoryCapability]	KB_828678_Instance_43 [MemoryCapability]
0.2711	KB_828678_Instance_201 [StorageCapability]	KB_828678_Instance_42 [StorageCapability]
0.2711	KB_828678_Instance_201 [StorageCapability]	KB_828678_Instance_41 [StorageCapability]

Table 7. Similarity measure values between Requirement1 and IBM cloud instances

Similarity	Node in Requirement1	Node in IBM
1.0000	KB_828678_Instance_200 [VM]	KB_828678_Instance_110 [VM]
0.3497	KB_828678_Instance_203 [Ethernet]	KB_828678_Instance_115 [Ethernet]
0.2664	KB_828678_Instance_202 [MemoryCapability]	KB_828678_Instance_114 [MemoryCapability]
0.2664	KB_828678_Instance_202 [MemoryCapability]	KB_828678_Instance_113 [MemoryCapability]
0.2664	KB_828678_Instance_201 [StorageCapability]	KB_828678_Instance_112 [StorageCapability]
0.2664	KB_828678_Instance_201 [StorageCapability]	KB_828678_Instance_111 [StorageCapability]

define compatible cloud provider for an application by matching parameters of application requirements and cloud SLAs. It gives suggestion to a consumer in terms of number of matched parameters. A cloud having larger number of matched parameters, is a compatible cloud for an application. This algorithm can be used as independent tool or it can also be used as an plugin code to any middleware tool.

References

[1] Cloud Computing Use Cases Discussion Group, *Cloud Computing Use Cases A white paper*, vol. 4, July 2010.

[2] “Amazon simple storage service (s3) service level agreement,” 01 October 2007.

[3] “Amazon elastic compute cloud (ec2) service level agreement,” 23 October 2008.

[4] “Cirrocumulus: A semantic framework for application and core services portability across heterogeneous clouds,” project at Kno-e-sis Center at Wright State University, 2010.

[5] “Rdf primer,” 10 February 2004.

[6] Philippe Hegaret, Ray Whitmer, and Lauren Wood, “Document object model (dom),” January 2005, W3C Recommendation.

[7] “Jena - a semantic web framework for java,” 2011.

[8] S. Melnik, H. Garcia-Molina, and E. Rahm, “Similarity flooding: A versatile graph matching algorithm and its application to schema matching,” in *18th International Conference on Data Engineering*, 2002, pp. 117–128.