ABSTRACT

Cloud computing enables the automated creation of business applications from independently developed and deployed services. Mechanisms are thus needed to select service components that meet or exceed the functional and non-functional requirements of such applications. The primary objective of service selection in the Cloud could be viewed as the maximization of an application-specific utility function that matches the constraints of the service requester against the capabilities and offerings of the service provider(s). In this paper, we propose such an approach that computes the match between service requests and offerings, based on their functional and non-functional properties in an efficient manner (in terms of space and time complexity). The proposed technique incorporates behavior monitoring of potential matches to ensure enhanced application-specific utility. We compare our approach with similar existing approaches to show its applicability and performance.

Keywords: Cloud computing, Similarity, Matching, Composite services.

1. INTRODUCTION

Cloud computing facilitates scalable and cooperative sharing of resources (e.g., storage, processors, platforms, or application services) among different organizations. A “Cloud” can thus be defined as a platform that enables on-demand access to applications from anywhere in the world, without considering their implementation details. In other words, the consumers do not need to be concerned about the installation and configuration issues of the services they use. In recent years, we have seen an increase in such content access techniques, where any reference to the underlying hosting infrastructure is missing/hidden. This is mainly due to the increasing availability of the three ‘flavors’ of the Cloud computing paradigm:

- Infrastructure-as-a-Service (IaaS).
- Platform-as-a-Service (PaaS).
- Software-as-a-Service (SaaS).

IaaS is a provision model in which an organization outsources the equipment used to support operations, including storage, hardware, servers and networking components. The service provider owns the equipment and is responsible for housing, running and maintaining it. The client typically pays on a per-use basis. Similarly, PaaS delivers a computing platform as a service, often consuming Cloud infrastructure and sustaining Cloud applications. It facilitates deployment of applications without the cost and complexity of buying and managing the underlying hardware and software layers (e.g., the Facebook application). Lastly, SaaS delivers software/application as a service over the Internet, eliminating the need to install and run the application on the customer’s own computers, thereby simplifying maintenance and support. Cloud computing could thus be referred as a primary example of the service-oriented computing (SOC) paradigm, where the service providers publish their services on the Cloud and mostly charge customers on a pay per use model for accessing these services[4].
environment not only requires a language that must be able to abstract the services and the environment, but also needs the semantic descriptions of these services. Cloud services are usually defined hierarchically (mainly using Cloud computing ontology). The ontology-based representation of the Cloud computing environment enables the conceptualization of common attributes among Cloud resources and their semantic relationships. Since user requirements for Cloud services vary, service providers have to ensure that they can be flexible in their service delivery while keeping the users isolated from the underlying infrastructure. Moreover, service consumers usually require faster response times which may be achieved by distributing requests to multiple Clouds in various locations at the same time. This creates the need for finding similar services or applications on different Clouds that satisfy the different (and at times inter-related) consumer constraints. Cloud-based services are self-described, self-contained and platform-independent computational elements that can be published, discovered, and composed using standard protocols, to build applications across various platforms and organizations in a dynamic manner. With the increasing agreement on the functional aspects of these Web services (such as using WSDL [2] for service description, SOAP [18] for communication, and WS-BPEL [19] for composing Web services etc.), the research interest is shifting towards the non-functional aspects of Web services.

Developers can now add descriptions (using standards such as OWL-S [7]) to their Web services to define and advertise the non-functional aspects of services (including input, output, pre-condition, post-condition and functions), thereby facilitating automated discovery, invocation and inter-operation. However, the first step in this process is to 'resolve' the consumer request against prospective Web services, so that the most appropriate component could be selected. The expected availability of a large number of highly specialized component services means that it would be increasingly challenging to find the most suitable service(s) in a reasonable amount of time [15]. Moreover, some Web services may not be able to satisfy consumer requests individually, and hence need to be integrated with other Web services to provide the desired functionality. This adds to the complexity of an already challenging problem [3].

In this paper, we present a novel approach (defined MASC: Matching Alternative Services in the Cloud) for Web service selection in the Cloud. MASC is divided into three levels. In the first level, we filter the available Web services under a specific category based on their functional properties such as input, output and operations. In the second level, we further reduce the service search space based on non-functional properties, such as Quality of Service (QoS) parameters [1]. Once a reduced pool of similar Web services is obtained, we rank them based on their utility value (in the third level). Utility functions allow consumers and providers to ascribe a value to the usefulness of a system artifact. Since QoS is an important consideration for dynamic service selection in the context of Clouds [14], we use QoS parameters for utility evaluation. MASC filters Web services at each level so that more costly operations (e.g., reputation calculations) are applied on a reduced number of candidate services to shorten the time and space complexity of this search process. Moreover, since service selection is an on-demand process, we apply the MASC filters at run time. The main contribution of the new approach is rather than selecting Web services randomly, MASC provides a selection of services that have the highest number of operations in coherence with the consumer request, and have corresponding acceptable QoS parameter values. MASC combines semantic and syntactic matching with QoS requirements. In addition, MASC ranks the available candidate services to provide the user with a list of candidate services even if no exact match is found. Moreover, MASC improves the service selection process by reducing the time and search space complexity.

The rest of the paper is organized as follows. Section 2 provides a motivating scenario which is used to illustrate the MASC approach further in the paper. Section 3 describes our approach in detail. Section 4 presents the experiments, while Section 5 lists the related works. Finally, Section 6 concludes the paper and provides directions for future work.

2. MOTIVATION

In this section, we present an example scenario to motivate the problem and associated solution.

Figure 1. shows a typical service auction scenario in a Cloud computing environment called ‘Trading Scenario’ (TS), where a broker conducts an auction to match bids of multiple resources advertised by different providers. 1) The providers advertise their resources with the required price. 2) The consumers then submit their bids to show the degree of interest in the advertised resources. 3) The bids from the consumers are queued in the database by Job schedule services which will help in calculating the winning bid. 4) On the other hand, the resources are indexed and stored in the database by Catalogue services. 5) After that, the Trading broker service coordinates the matching of resources and bids, and trading between auction participants. 6) At the end of the auction the broker decides the winners and sends the reservation requests to the Reservation service. 7) Then the reservation service informs the resource providers and consumers about the final result (i.e.,
who won the bid). The payment processing takes place through the Accounting service.

Figure 1: A trading scenario (TS) in the Cloud

In this paper, we focus on five main services in TS: Job schedule services (Jss), Catalogue services (Cs), Trading broker service (Tbs), Reservation service (Rs) and Accounting service (As) (see Figure 1). Jss is the service that stores the bids coming from the consumers and sorts them by date, time and price. Cs stores the resources advertised by the providers and indexes them to reduce the complexity for the bid matching process. Tbs is the service that manages the auction and matches the bids with the required price. Rs is the service that reserves a specific resource to the winner of the auction and informs results to the participants of the auction process. Finally, As is the service that is responsible for processing payments: check if the consumer’s credit meets the credit score requirements, make a payment and send the results to the user. As consist of four operations: Get-Quote, Calculate-Price, Get-Time and Send-Receipt. All the previous services are outsourced through TS. TS has to deal with multiple issues when trying to outsource component services. For example, when TS is looking for an accounting service, the first step is to find all the Web services that provide this functionality (i.e., resolve both syntactic and semantic equivalence). Furthermore, when it finds functionally equivalent Web services for accounting service, they may have different non-functional (QoS) properties (such as service A may have a response time of 3ms and service B may take 7ms to respond to user requests). Hence TS needs to differentiate among the candidate services based on the value (utility) they add to the composition. Thirdly, if a component service fails during execution, TS needs to find a suitable replacement to meet the consumer requirements. The main motivation behind MASC is to solve these above mentioned issues while reducing the time and space complexity of this (service) search process. We believe that an efficient solution to the service selection problem is also paramount in reducing fault recovery time in Clouds, for cases where a faulty service needs to be replaced by a ‘similar’ one [1].

3. MASC ARCHITECTURE

In this section, we present the architectural details of our proposed approach (MASC: Matching Alternative Services in the Cloud). Generally, any software matching measurement consists of two components: syntactic and semantic. In syntactic matching, we look for the matching between data items where value of this matching usually lies in the range [0,1]. In semantic matching, we look for defined relationships among various terms and concepts (e.g. defined in ontology or extracted). In MASC, both syntactic and semantic matching filters are applied to find a set of Web services that match users’ requirements. We assume that each Web service is defined using a description language such as WSDL (Web Service Description Language) which describes the functional service properties and its interface. WSDL files are published in a service registry that allows providers to advertise general information about their Web services. This information is used by clients for discovering providers and Web services of interest. Numerous Web services providing similar functionality may thus be listed under the same category in a Cloud service registry.

In MASC, we search the registries and retrieve the Web services under a category and send them to the first level (FCF). Thus, MASC starts by calculating the syntactic matching for each attribute, and if a syntactic match is not found, candidate services are checked for semantic matching. The attributes types in FCF are: (1) syntactic attributes, which include the list of input and output parameters, the data types of the parameters, and the protocol to be used to invoke the Web service such as SOAP. (2) Semantic attributes, include the pre-conditions and effects of an operations execution.

We feed the (reduced) output set from (FCF) into the second level (NCF) filter (see Figure 2.). NCF is a filtering mechanism based on QoS parameters which measures the quality of a Web service over the consumer-desired parameters. There are many parameters that can be used to measure a Web service’s quality such as response time, availability, reliability, cost, security and privacy. In addition, we can add any new specification to each one of these filters. After finding the providers that support the same service based on functional context (n-m providers), we filter them based on QoS requirements and are left with (n-m-k providers). In the third level, we rank the candidate Web services based on their utility. We divide the Web services into
two sets: HighRank set and LowRank set. The HighRank set includes the Web services that have QoS values higher or equal to consumer’s requested values with the constraint that the price does not exceed consumer’s maximum price. However, the LowRank set includes the Web services that have QoS values lower than the requested values. In case of empty HighRank set, the first Web service in the LowRank set is considered the best candidate. Note that the first two levels (i.e., FCF and NCF) are ‘context based filters’. A context is “any information that can be used to characterize the situation of an entity. An entity is a person, place, or subject that is considered relevant to interaction between a user and an application, including the user and the application themselves”. Due to space restrictions, we omit further details regarding context definition. The interested reader is referred to [13]. In the following, we provide details for the MASC filtering levels mentioned above.

![Figure 2: Matching levels for MASC](image)

### 3.1 Level I: Functional Context Filter (FCF)

As mentioned earlier, the WSDL files are published in a registry (external and/or local) and consist of (textual) descriptions of the Web service’s operations (such as input, output, conditional output, precondition and postcondition). While some service providers describe these functionalities in different ways (e.g. both input and precondition are described as input), so MASC includes preconditions with inputs and postconditions with outputs. In MASC, we extract this and other functionalities’ information from OWL-S which provides service profile, process-model, and service grounding (for more details see [7]). In an ideal scenario we would be able to find a service that perfectly matches to user requirements. However, in a Cloud-based environment with numerous combinations of service attributes (i.e., input, output, and operations) the chances of having such a perfect match may be slim. Thus, instead of trying to find a perfect match, we could find a Web service that fulfills the user’s requirements as much as possible (i.e., Web services may provide less functionalities or may have more functionalities than requested). In MASC, we first look for a perfectly matching service, and then we increase our search to incorporate services that provide more functionalities than requested. If we cannot find any suitable candidate in the first two searches, we expand our search to include services that provide less than desired functionalities. However, in such scenarios we would need to compose multiple services to provide the requested functionality. Thus, we may run into scenarios such as: Equivalent, Subsume, Not-equivalent and Plug-in services. For term definitions and details, the interested reader is referred to [13].

To classify any Web service under one of the matching scenarios in MASC, we identify the service operations according to the following. We consider three main types of operations: one-way, request-response, and confirmation. In one-way operations, the Web service receives a message without producing any output message (i.e., one-way communication). In request-response operation, the service receives an input message, processes it, and sends correlated output message to the sender. Confirmation operation sends an output message but does not expect to receive any more messages. As we can see Request-response operations have both input and output messages. However, one way operations only contain input messages and confirmation operations only produce output messages. Each message consists of one or more parameters called parts in a WSDL. A parameter has a name and a data type. The data type gives the range of values that maybe assigned to the parameter. The first step in finding functional equivalence among Web services is to extract this parts information from the WSDL file for the parameters and return values of operations provided by candidate Web services.

**Definition 3.1.1.** Each Web service is accessible via operations and each operation is identified by a tuple 
\(<\text{Description}_ij, \text{Mode}_ij, \text{Input}_ij, \text{Output}_ij, \text{Purpose}_ij, \text{Category}_ij, \text{Quality}_ij>\), where Description\(_ij\) is a textual summary about the features of the operation, Mode\(_ij\) is the type of operation (i.e., one way, response-request or confirmation), Input\(_ij\) is the input of the operation (if it exists), Output\(_ij\) is the output of the operation (if it exists), Purpose\(_ij\) is the business function offered by the operation, Category\(_ij\) describes the operation domain, Quality\(_ij\) provides the operation’s qualitative properties.

**Definition 3.1.2.** operation\(_ij\) is similar to operation\(_kl\) or operation\(_ij\) is subsumed by operation\(_kl\) if

1. \(\forall x \in \text{Input}_ij \exists x \in \text{Input}_kl\)
2. \(\forall y \in \text{Output}_ij \exists y \in \text{Output}_kl\)
3. \((\text{Category}_ij = \text{Category}_kl)\) or \((\text{Category}_ij \subseteq \text{Category}_kl)\).
4. \(\text{Mode}_ij = \text{Mode}_kl\) (see definition 3.1.1).
5. \((\text{Purpose}_ij = \text{Purpose}_kl)\) or \((\text{Purpose}_ij \subseteq \text{Purpose}_kl)\).
6. Text matching between Description\(_ij\) and Description\(_kl\) ≥ \(t\) (is a pre-determined threshold).
In FCF, we apply neighborhood calculation to find the similar Web services based on their functional properties. This filter works based on three predefined matrices: input matrix, output matrix and operation matrix. Figure 3 shows the steps of how FCF works. Upon arrival of a consumer request, a list of n services is retrieved from the external registry under the requested category. FCF’s Matrix Builder module then creates the three matrices: input, output and operation. Each matrix has \( m \times n \) dimensions where \( m \) is the number of retrieved Web services and \( n \) is the number of inputs, outputs or operations for each matrix respectively. For instance, an \( A_{6 \times 4} \) operations matrix is created as:

\[
A_{i,j} = \begin{cases} 
1 & \text{if Web service}_i \text{ has operation}_j \\
0 & \text{otherwise.}
\end{cases}
\]

When the Web service \( i \) does not provide the operation \( j \), the value of \( A_{i,j} \) is zero. Figure 5 shows the main operations for the accounting services and the operations for each service. For example, service2 provides the operations: Get-Quote, Calculate-Price, Get-Time then the operation vector for it is \( <1,1,1,0> \). While we are filling the matrix, the main concern is determining if the parameters of service \( i \) is the same as the parameters in the matrix. For instance, finding a flight using Web service \( i \) requires the input (airport name), but Web service \( j \) may requires the input (zip code) for the same operation. Hence it is important to find semantic similarity to address such scenarios. We use Figure 4 to illustrate matrix building based on the operations included in \( Ac \). The matrix contains six Web services and four operations. The matrix dimensions are \( A_{6 \times 4} \). The first step of MASC is determining the inputs, outputs and operations of the Web service which based on the consumer request.

If all the properties are available in the matrix then we just add the service name, and insert one under the property if the service provides it, else insert zero. However, if the property does not exist, we will edit and add the new property to the matrix. For example, one provider wants to publish service7 which includes the operations (Get-Quote, Calculate-Price, Get-Time and Get-Rate). The first three operations exist in the matrix but the last operation is a new one. In this case we add the new property (Get-Rate) and add one under the operations (Get-Quote, Calculate-Price and Get-Time).

<table>
<thead>
<tr>
<th>Service</th>
<th>Get-Quote</th>
<th>Calculate-Price</th>
<th>Get-Time</th>
<th>Send-Receipt</th>
<th>Get-Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service1</td>
<td>0.0635</td>
<td>0.1111</td>
<td>0</td>
<td>0.0635</td>
<td>0</td>
</tr>
<tr>
<td>Service2</td>
<td>0.0635</td>
<td>0.1111</td>
<td>0.0794</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Service3</td>
<td>0</td>
<td>0.1667</td>
<td>0.1190</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Service4</td>
<td>0.0476</td>
<td>0.0833</td>
<td>0.0595</td>
<td>0.0476</td>
<td>0</td>
</tr>
<tr>
<td>Service5</td>
<td>0.0953</td>
<td>0.1667</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Service6</td>
<td>0</td>
<td>0.1111</td>
<td>0.0794</td>
<td>0.0635</td>
<td>0</td>
</tr>
<tr>
<td>Service7</td>
<td>0</td>
<td>0.0833</td>
<td>0.0595</td>
<td>0.0476</td>
<td>0.0119</td>
</tr>
</tbody>
</table>
FCF inserts the requirements into a vector by getting the parameters for a specific category from the service repository. It then builds a priority matrix. The priority matrix is a matrix that gives weight to each property and will move the focus towards more important operations. Based on TF-IDF [10], we define the priority matrix over the original matrix $A_{m \times n}$ to compute the weight of each item as:

$$w_{i,j} = \log(A_{i,j}/|O_p|) \times (A_{i,j} \times |W_{si}|)/|O_p|$$

(1)

where $A_{i,j}$ is one if the operation $j$ exists in Web service $i$, otherwise $A_{i,j}$ is zero, $|O_p|$ is the number of times that $O_p$ has been used by all Web services, $|W_{si}|$ is the number of operations in the matrix and $|O_p|$ is the number of operations for Web service $i$. The result after applying Equation 1. to our example will be the priority matrix in Table 1. The operations that are provided most often by the Web services in the same category will have the highest weights and the operations that are provided less will have lower weights. After building the priority matrix, FCF converts each row of the matrix into binary vectors, for example if the consumer request contains the operations <Get-Quote, Calculate-Price, Get-Time> while the available service has <Get-Quote, Calculate-Price, Get-Time, Send-Receipt, Get-Rate> then the query vector of this Web service is <1, 1, 1, 0, 1>. FCF finds the match Web services based on vector similarity [6] as:

$$Matching(I, J) = \frac{\sum_{i=1}^{n} (v_i \times j_i)}{\sqrt{\sum_{i=1}^{n} v_i^2} \times \sqrt{\sum_{i=1}^{n} j_i^2}}$$

(2)

Now, let us suppose that the consumer requests a matching for Web service that includes the operations: <Get-Quote, Calculate-Price, Get-Time> i.e., the vector will be $V_1 = <1, 1, 1, 0, 1>$ and it will compared to all vectors $V_d$ where $d \in [1,m]$ in the matrix $A_{m \times n}$.

$$t = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}, \text{Matching}(I, J) = \begin{bmatrix} 0.6667 \\ 0.6002 \\ 0.8054 \\ 0.6667 \\ 0.3052 \end{bmatrix}$$

(3)

If the threshold $\varepsilon$ for selecting the Web services is (0.8) then the result from FCF is the set { Webservice2, Webservice3, Webservice4, Webservice5 }. In this case, we find that Webservice3 and Webservice5 have the same matching value (0.8403). Notice that Webservice2 does not provide the operation Get-Quote and Webservice5 does not provide the operation Get-Time. In such case, we return back to the operation priority matrix which shows the priority for the operation Get-Quote is 0.1190 and the priority for the operation Get-Time is 0.0953, so we prefer Webservice3 contains the higher priority operation.

### 3.2 Level II: Non-functional Context Filter (NCF)

NCF is divided into two steps: The first step checks for the service availability, thereby, eliminating the Web services that are unavailable. The second step checks Web service matching based on other QoS parameters (e.g., response time, throughput, reliability, etc). We use the ping utility for the former, which has been used for Web service performance measurements [8]. After determining the set of Web services that respond to the ping inquires, we start the second step where we use Context Policy Assistance (CPA) to test the similarity between the QoS parameters that are required by the consumer and the QoS parameters offered by the available Web services. CPA is created by the service provider and should be attached to the service. It facilitates interaction between providers and the service registry to store the context policies. For further details, the interested reader is referred to [13].

#### Definition 3.2.1. QoS vector description:

It is an extendable vector used to define QoS parameters of the provider and the consumer. It is expressed as: $QoS = <QoS_1, QoS_2, ..., QoS_n>$, $n \in \mathbb{R}$, where $QoS_n$ indicates the $n^{th}$ QoS attributes and $QoS_i$ where $i \in [1,n]$ is equal to {Availability, Cost, Response time, Error rate, Throughput, Reliability, Reputation, and Security}.

QoS parameter matching is done as:

1) Convert the parameters into QoS vector descriptions. Then, we have one vector for the consumer request $<ConQoS_1, ConQoS_2, ..., ConQoS_n>$ and multiple vectors for the provider offerings $<Pro_{1QoS_1}, Pro_{1QoS_2}, ..., Pro_{1QoS_n}> ... Pro_{kQoS_1}, Pro_{kQoS_2}, ..., Pro_{kQoS_n}>$. Here we assume that the providers provide trusted information for the QoS values [18]. There are three different cases in converting the QoS parameters: the consumer vector is greater than the provider vector, the consumer vector is less than the provider vector, or the consumer vector is equal to the provider vector. In the first case we add zeros at the end of the consumer vector and in the second case we add zeros at the end of provider vector.

2) Create a new vector called conform with length equal to the max length of consumer and provider vectors. For each element in the vector use the polices to compare the conditions, and if the condition is met then add one to the conform vector else put zero.

3) Calculate the conformity degree between the services for the consumer QoS, and the provider QoS as:
where \( i,j \) are Web services, \( z \) is the maximum length of parameters, i.e., \( z = \max (|QoS_i|,|QoS_j|) \), and \( \text{Weight}_k \) is the weight assigned to each QoS parameter. Consumers may have different expectations about the conformity degree of their services. For this purpose, they provide a conformity threshold \( \theta \) (\( 0 < \theta \leq 1 \)). In NFC, we find all Web services \( j \) where the conformity degree (\( QoS_i, QoS_j \)) is greater than \( \theta \), which are then passed to the ranking level. The conformity threshold is given by the consumer as a part of his profile, while the QoS weight is created automatically by the system based on the level of the consumer’s expertise.

### 3.3 Level III: Web service Ranking

In this level, MASC ranks the Web services based on the range compatibility of the QoS parameters. We use weighted sum filter function after converting the QoS parameters into a range vector in the format of the component vector description.

**Definition 3.3.1.** The component vector description is expressed as: \( < (QoS_1, QoS_{1\min}, QoS_{1\max}), \ldots, (QoS_n, QoS_{n\min}, QoS_{n\max}) >, n \in \mathbb{R} \), where \( QoS_{i\in[1;n]} \) is the best QoS value for parameter \( i \), \( QoS_{i\min} \) is the minimum acceptable value for parameter \( i \), and \( QoS_{i\max} \) is the maximum acceptable value for parameter \( i \), then \( QoS_{i\min} \leq QoS_i \leq QoS_{i\max} \).

Each vector is accompanied by a decision model, i.e., ranges of all the QoS parameters as well as their respective priorities also known as the weights. The ranking will be based on the matching degree (i.e., how near the QoS parameter that is provided by the provider is to the QoS parameters required by the consumer). Note that we can use the provider QoS values in one of two ways: (i) QoS values as advertised by the service provider, or (ii) QoS values obtained using behavior monitoring through the community (i.e., provider reputation).

The best ranked Web service will be the Web service that is closest to the best value and the worst ranked will be the Web service that is the farthest from the best value. However, if a Web service provides a larger value than the best value but has a lower cost associated to it, then the system will give this Web service a higher rank.

**Definition 3.3.2.** All Web services \( WSi \in \omega \), where \( \omega \) is the set of all Web services which are similar (functional and nonfunctional) to the requested service:

\[
\text{WebServiceRankSet} = \begin{cases} 
\text{HighRank} & \text{if } WSi \geq \mu \land \text{cost} \leq \varphi; \\
\text{LowRank} & \text{if } WSi < \mu.
\end{cases}
\]

where \( \mu \) is the best value of QoS that is provided by the consumer from the vector \(< (QoS_j, QoS_{j\max}), QoS_{j\max} >\), and \( \varphi \) is the acceptable cost by the consumer.

### 4. PERFORMANCE ANALYSIS

In this section, we define an analytical model to study the performance of the proposed technique (MASC). We focus on computing the total time and searching space complexity for checking the matching degree of the target Web services through our three levels. We compare our technique with three similar existing works through this analytical model.

Table 2 defines the parameters and symbols used hereafter. We assume that Web services are divided into categories (e.g., under the external registry categories).

To simplify the analysis, we assume that the times to retrieve a description from a service registry and parse that description are fixed values. Based on the categories, we compute the average matching time \( T_{\text{match}} \) which is the time it takes to find the similar services.
Table 2: Definition of Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_j)</td>
<td>The value of the (j)th component of consumer’s vector.</td>
</tr>
<tr>
<td>(B_{ij})</td>
<td>The value of the (j)th component of the (i)th Provider’s vector.</td>
</tr>
<tr>
<td>(A_{j\text{(min)}})</td>
<td>The minimum allowed value of the (j)th component of consumer’s vector as provided by the consumer.</td>
</tr>
<tr>
<td>(A_{j\text{(max)}})</td>
<td>The maximum allowed value of the (j)th component of consumer’s vector as provided by the consumer.</td>
</tr>
<tr>
<td>(B_{ij\text{(min)}})</td>
<td>The minimum allowed value of the (j)th component of the (i)th Provider’s vector as provided by the provider.</td>
</tr>
<tr>
<td>(B_{ij\text{(max)}})</td>
<td>The maximum allowed value of the (j)th component of the (i)th Provider’s vector as provided by the provider.</td>
</tr>
<tr>
<td>(WA_j)</td>
<td>The weight of the (j)th component of consumer’s vector as provided by the consumer.</td>
</tr>
<tr>
<td>(WB_{ij})</td>
<td>The weight of the (j)th component of the (i)th Provider’s vector as provided by the provider.</td>
</tr>
<tr>
<td>(f_j)</td>
<td>Fitness of the solution for participant (j).</td>
</tr>
<tr>
<td>(F_s)</td>
<td>Fitness of the solution for all participants.</td>
</tr>
<tr>
<td>(N_{\text{com}})</td>
<td>Number of categories.</td>
</tr>
<tr>
<td>(N_p)</td>
<td>Number of policies per service.</td>
</tr>
<tr>
<td>(N_{cs})</td>
<td>Number of context specifications per policy.</td>
</tr>
<tr>
<td>(N_{\text{member}})</td>
<td>Number of members per category.</td>
</tr>
<tr>
<td>(N_r)</td>
<td>Number of rules.</td>
</tr>
<tr>
<td>(N_s)</td>
<td>Number of services in a particular category.</td>
</tr>
<tr>
<td>(T_{\text{poll}})</td>
<td>Time to fetch all policies of a service.</td>
</tr>
<tr>
<td>(T_{\text{poll}1})</td>
<td>The time to parse a service description and the network transmission delay.</td>
</tr>
<tr>
<td>(T_{\text{poll}2})</td>
<td>The time spent by each category to process its sub request.</td>
</tr>
<tr>
<td>(T_{\text{pone}})</td>
<td>Time to fetch one policy of a service.</td>
</tr>
<tr>
<td>(T_{\text{es}})</td>
<td>Time to get a context specification.</td>
</tr>
<tr>
<td>(T_{\text{XML}})</td>
<td>Time to parse a service description.</td>
</tr>
<tr>
<td>(T_{\text{Net}})</td>
<td>Network transmission delay.</td>
</tr>
<tr>
<td>(T_{\text{rep}})</td>
<td>Time spent to assess a reply from a category.</td>
</tr>
</tbody>
</table>

\[
T_{\text{match}} = \left( \frac{T_{\text{MINmatch}} + T_{\text{MAXmatch}}}{2} \right)
\]  

where \(T_{\text{MINmatch}}\) is the best case matching time and \(T_{\text{MAXmatch}}\) is the worst case matching time. \(T_{\text{match}}\) includes a polling time \((T_{\text{poll}})\) and a decision time \((T_{\text{dec}})\), where, \(T_{\text{poll}}\) is a combination of \(T_{\text{poll}1}\) and \(T_{\text{poll}2}\). \(T_{\text{poll}1}\) includes the time it takes to fetch the policies, parse a service description and the network transmission delay. In the best case, the Web service would have only a single policy \((T_{\text{MINpoll}1})\) and in the worst case it may have \(N_{\text{com}}\) policies \((T_{\text{MAXpoll}1})\). Hence, \(T_{\text{poll}2}\) includes the time spent in each category to process its sub request.

\[
T_{\text{MINpoll1}} = T_{\text{pone}} + T_{\text{XML}} + T_{\text{Net}}
\]

\[
T_{\text{MAXpoll1}} = T_{\text{pone}} + T_{\text{es}} \ast \left( T_{\text{XML}} + T_{\text{Net}} \right)
\]

We multiply \(T_{\text{es}}\) by two because we need to compare each policy twice: once for the provider and once for the consumer. At a minimum, each policy would be compared to a single policy on both sides. Similarly, we multiply \(T_{\text{XML}}\) by four because we need to parse the description of XML four times (we need to parse XML files twice for the consumer and twice for the provider: once for determining the properties and once for determining the policies).

\[
T_{\text{MINpoll2}} = 2T_{\text{pone}} + 2T_{\text{es}} + 4T_{\text{XML}}
\]

\[
T_{\text{MAXpoll2}} = T_{\text{pone}} + N_{cs} \ast (T_{\text{XML}} + T_{\text{Net}})
\]

In calculating \(T_{\text{MINpoll2}}\) we multiply \(T_{\text{pone}}\) by two because we retrieve the policy for the source and the category member. The decision time \((T_{\text{dec}})\) includes the network delay and the time spent to assess a reply from the Web services under the same category. In the best case,

\[
T_{\text{MINdec}} = T_{\text{Net}} + T_{\text{rep}}
\]

\[
T_{\text{MAXdec}} = N_{\text{com}} \ast (T_{\text{Net}} + T_{\text{rep}})
\]

Based on the previous equations, \(T_{\text{match}}\) is then,

\[
T_{\text{match}} = \left( \frac{T_{\text{MINmatch}} + T_{\text{MAXmatch}} + N_{\text{com}} \ast (T_{\text{Net}} + T_{\text{rep}}) + T_{\text{MINdec}} + T_{\text{MAXdec}}}{2} \right)
\]

Figure 5 shows a comparison between the Brute-force method (exhaustive search), CME [13], CCB [16] and MASC for service matching time based on the number of services required by the consumer. Note that Brute-force has the highest matching time especially when we have a large number of services. CME performs better than Brute-force method, but still takes more time than CCB. However, our method provides the lowest matching time, even if the number of services is large because for each filter we reduced the number of compared services. Figure 6 shows the relationship between the number of services and the search space (number of service calls) for each matching method. We can see that MASC perform better in comparison with other approaches because CME calls each service three times for comparison, CCB calls each service twice, while MASC calls just the candidate services twice.
5. RELATED WORK

In this section, we provide a brief overview of some of the related literature. Several methods have been proposed to deal with the service matching problem. For instance, the technique in [20] uses two stage assessments. In the first stage all service belonging to a specific category are gathered. The second stage consists of finding similarity among these services based on input, output, conditions and effects. Context-based matching (CBM) has been proposed in [13], where the matching process is performed via peer-to-peer interactions between a context-based matching engine, CPAs and community services. A service consumer sends a matching request to context-based matching engine which sends a sub request to the communities and compares the consumer requirement with each community members. Then the context-based matching engine finds the intersection between the matching set from each community. The communities have been created based on the policies inside the Web services. The problem in this technique is that the number of comparisons will be high if the same Web service exists in all communities (i.e., the Web service includes all policies that the consumer has requested). Circular context-based (CCB) [16] is another technique that compares context extracted from each Web service based on its WSDL description to with other Web services’ textual description context. The second stage consists of finding the context overlap among the Web service through parsing WSDL file. The technique presented in [9] focuses on syntactic comparison among attributes of Web services. Since we use contextual information of Web services, we position our work with existing context-oriented Web service frameworks. Several context-aware approaches have recently been proposed to enhance Web service discovery and composition mechanisms. For instance, context aware service discovery technique for mobile environments is proposed in [11]. It defines the context of a Web service as a set of attributes included in the service description. Examples of context attributes include user location and network bandwidth. The discovery engine first lookups for Web services based on traditional criteria. Then, it reduces the qualified services to be returned to clients through context attribute evaluation. This approach uses contextual information for service discovery not for service composition. Additionally, it focuses on client-related contextual information. It does not seem to consider provider-related context which is important for Web service composition. Finally, the definition of context is limited to some attributes added to service descriptions. We adopt a more generic definition of Web service context through an ontology-based categorization of contextual information. Contextualization is also proposed at the Web service deployment, composition and conciliation or matching levels in [12]. The description of contexts is assumed to occur along three categories: profile, process model, and grounding. The profile describes the arguments and capabilities of a context. The process model suggests how context collects raw data from sensors and detects changes that need to be submitted to the Web service. Finally, the grounding defines the bindings (protocol, input/output messages, etc.) that make context accessible to a Web service. The authors did not however mention how relevant contexts are elicited in a service matchmaking process.

We identify several improvements over current matching techniques. First, existing techniques rely mainly on static information while our technique is based on dynamic information (i.e., the best matching list changes based on the service’s performance). Second, existing frameworks use a limited number of matching attributes such as only inputs and outputs. In our approach, we defined all possible attributes by identifying them as either required or optional attributes, and we cluster these into three levels: functional, non-functional and ranking. We start by filtering n Web services through the functional context filter (FCF) to have n-m web services. Then we filter these n-m Web services through the functional context filter (FCF) to have n-m-k Web services. Lastly, we filter these web services
using a utility-based ranking to have n-m-k similar Web services. Third, our technique reduces the comparison cost (time and space) by filtering out services at each level to reduce the number of comparisons needed for further evaluation.

6. CONCLUSION

MASC extends the scope of Web service selection by providing a semantic Web service matching and ranking approach for the Cloud. Rather than selecting Web services randomly, we provide a selection of services that have the highest number of operations in coherence with the consumer request, and have corresponding acceptable QoS parameter values. Our method combines semantic and syntactic matching with QoS requirements. In addition, our method ranks the available candidate services to provide the user with a list of candidate services even if no exact match is found. Experiment results show that our proposed technique improves the service selection process by reducing the time and search space complexity. In the future, we plan to investigate techniques for enabling automated planning and replacement of faulty Web services with similar ones that have high utility. In addition, we will conduct more analysis about the performance of the matching algorithms in real cloud computing platforms.

REFERENCES


ABOUT THE AUTHORS

Amal Alhosban is an Assistant Professor of computer science at New Mexico Highlands University. She received the Ph.D. degree in computer science from Wayne State University in July 2013. Her research interests include service computing, reliable distributed systems, semantic Web, data mining, big data, software engineering and wireless networks. She has published several research papers on these topics in top journals and conferences, and she is an active member of ACM and IEEE.

Khayyam Hashmi is a Ph.D. candidate in the Department of Computer Science at Wayne State University, where he is a member of the Services COmputing REsearch (SCORE) Laboratory. He received his M.S in computer science degree for National University of Computer and Emerging Sciences, Pakistan. His research interests include service computing, distributed systems, semantic web and software process engineering.

Zaki Malik is an Assistant Professor of computer science at Wayne State University, where he heads the Services COmputing REsearch (SCORE) Laboratory. His research interests lie broadly in service computing, reliable distributed systems, Web databases, and semantic integration systems. The focus of the research is on applying service-oriented techniques in these areas to build computer systems that are deployable in practice. He has published several papers in top journals and conferences, and authored a book on Trust Management in Service-oriented Environments. He also serves in various capacities on the committees and boards of database and service-oriented computing conferences and journals. He received the PhD degree in Computer Science from Virginia Tech, and is a member of the IEEE and ACM.

Brahim Medjahed is Associate Professor at the University of Michigan - Dearborn's Department of Computer and Information Science. He received the Ph.D. degree in computer science from Virginia Tech in May 2004. He received the 2004 "Outstanding Graduate Research Award" at Virginia Tech's Department of Computer Science. He also received the Computer Journal Wilkes Award for 2008 (Best paper award). His research interests include service-oriented computing, Web services, data integration, and semantic Web. He authored a book published by Springer on semantic Web service composition. He has published more than 60 papers in international journals and conferences.

(Received January 2012, revised January 2012, accepted November 2012)