Consumer-Centric Web Services Discovery and Subscription
Xuanzhe Liu, Li Zhou, Gang Huang, Hong Mei
School of Electronics Engineering and Computer Science, Peking University
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China, 100871
{liuxzh,zhouli04,huanggang}@sei.pku.edu.cn, meih@pku.edu.cn

Abstract
Nowadays, there are a number of similar web services over the internet or intranet. They provide consumers with more choices according to their personalized QoS requirements. However, in current web service discovery and subscription, it takes consumers too much time on manual selection and cannot easily benefit from the wide QoS spectrum brought by the proliferating services. In this paper, we propose a QoS-aware discovery and subscription approach to free consumers from time-consuming human computer interactions as well as help them negotiate QoS with multiple service providers. The core idea of this approach is to build up a “virtual service” grouping function similar services together (called service pool) and dispatching consumer requests to the proper service in terms of QoS requirements. This paper makes contributions for the aggregation and usage of similar web services in a “consumer-centric” manner. Such manner is on-demand, user-friendly and efficient. On-demand means the aggregate is driven by consumers instead of providers. User-friendly means consumers do not select services, handle different WSDL of similar services and switch between services at runtime any longer. Efficient means it integrates an efficient service search engine, reduces the incorrect services by some filter, discover the aggregate by a polynomial complex algorithm.

1. Introduction
Nowadays, using web services usually consists of three distinct phases: discovering a set of candidate services via service broker (e.g., UDDI), selecting the most proper one by investigating the details of services and their providers, and finally binding and invoking the target service. Such service discovery and subscription mode not only tires consumers but also prevents them from enjoying high quality of service (QoS) when the services proliferate. On one hand, there may be so many candidate services that have similar or same functionalities but vary from one another on QoS, e.g. performance, reliability, price and reputation. No matter how the consumers discover their desired services (by keyword searching or category browsing), they have to investigate the details of both services and their providers one by one so that they can select the most appropriate service (in fact, they select the most appropriate service provider). The more candidate services exist, the more time and energy consumers have to pay. On the other hand, a single provider usually serves with a limited QoS spectrum. The consumers have to pay enough attention to find the service provider who can satisfy their QoS requirements. If they cannot find a provider satisfying all their QoS requirements, they usually have to give up all the candidates or make some tradeoff. More seriously, if the subscribed service becomes unavailable, the consumers have to repeat the above process.

Our work aims to provide a more “consumer-centric” approach simplifying service discovery while satisfying QoS requirements. From our perspective, it is not necessary to list so many candidate providers to the consumers to browse, select and execute directly. These candidate providers can be grouped as a “virtual provider” that alleviates consumers from the time-consuming and tedious selection work: they do not have to view, select and bind to every provider manually; instead, there will be only one service provider serving different consumers. On the other hand, the consumers have the right to enjoy much wider QoS spectrum that a single provider cannot promise. The virtual provider clustering several providers may be capable of satisfying the QoS requirements as many as possible.

There are some technical challenges when applying the concept of “virtual provider” to improve user experience in service discovery and subscription [6]. Three most significant issues are:

1. How to identify the similar service providers and represent them to consumers as a single virtual provider?
2. How to select the most adequate services efficiently according to the QoS requirements?
3. How to process consumers’ requests by scheduling and dispatching them to the proper service providers at runtime?

In our previous work [6], [7] and [9], we have made some efforts on above issues by employing the concepts of service pool. Holding an assumption of clustering all the function-similar providers together as a pool, we allow consumers to directly submit their QoS requirements to the service pool while the QoS-aware selection and the request dispatch are almost transparent to consumers. A QoS-aware discovery mechanism has also been implemented to guarantee that the selected service has the higher availability while keeping the
lowest response time [7]. Briefly, the service pool addresses the third issue completely and touches the second issue partly. We did not deeply consider the underlying semantic of the web services when constructing the service pool, and the QoS-aware discovery algorithm covered only two attributes.

In this paper, we propose a total solution to the QoS aware service discovery and subscription based on the service pool while emphasizing on the first and second issues. After investigating the structure and the underlying semantic similarity of web services, we employ a similarity matching algorithm to cluster the function-similar services and generate a virtual WSDL so that the service pool can be accessed as a web service. Assuming consumers’ QoS requirements are compliant with Web Service Quality Model (WSQM) [10], we design algorithms to automatically finish the QoS negotiation between consumers and providers. Almost all details of such service discovery and subscription are transparent to the consumers, which is demonstrated by a web-based user interface.

The remainder of the paper is organized as follows. Section 2 shows the usage scenario of service pool. Section 3 presents the pool construction related issues, including the structure investigation of web services, the similarity clustering mechanisms and representation of the service pool. Section 4 proposes the QoS modeling by employing WSQM and discusses about the consumer preference driven selection algorithm. Section 5 presents a short view of the runtime issues. Section 6 gives the demonstration with the user interface and the experiment evaluation. Section 7 compares the related work. And the paper ends in Section 8 with conclusion and future work.

2. Usage Scenario

Briefly, there are four steps in QoS aware service discovery and subscription supported by service pool:

- The consumer logs on to our web-based UI and submits the functional requirements to find the services. We support two styles: search by keywords (like Google), and category browsing (like Yahoo! News). Either style may discover several candidate providers. Currently, the category browsing not only requires providers or brokers to build up the categories but also involves consumers in relative complex and tedious human-computer interactions.

- On the contrary, the keyword search not only puts much less burden on the providers, brokers and consumers but also may find much more candidate providers. In that sense, this paper focuses on the keyword search. As the keyword search may return too many results and some of them may not provide the desired functions, we provide some additional constraints in service similarity matching to improve the precision (discussed in Section 3).

- The candidate services found are now involved in a service pool. Since the pool can be accessed as a web service, a virtual WSDL will be generated to describe the functions of the pool according to some rules. The mapping rules between the service pool and actual services are generated as well.

- The QoS values of the services are also recorded, and the pool publishes its QoS spectrum to the consumer. The consumer can submit their QoS requirements within the spectrum. The pool processes the negotiation between different QoS requirements, and discovers the most adequate service.

- The consumer sends a request to the pool with the virtual WSDL, the pool routes the request to the actual provider and finally returns the results to the consumer. Note the pool WSDL provided by the service pool is just a description for the functionality that the consumer desires. The pool stores requests from every consumer. Once the consumer finally subscribes the actual provider, the execution engine will look up the discovered provider from step 3 first and route the request to it.

Obviously, by the four steps above, consumers can easily discover and subscribe the provider while attaining their desired QoS. We decompose the problems as follows. (1) To represent the different providers as a single service to the consumers, we need to cluster the similar providers by retrieving their underlying similarities in the topics of domain and function; (2) in order to reason about QoS properties in web services, a model is needed which captures the descriptions of these from a consumer perspective; (3) the pool should provide the capability of parsing the consumer QoS requirements, matching it to the most adequate provider and carrying out the execution task.

3. Building up Service Pool

3.1 Web Service Similarity

A service pool consists of a set of function-similar web services. Therefore, before building up a service pool, we should get these services first.

In the area of web search, there have been some existing works on the similarity retrieval [2]. These works mostly identify the semantic similarity based on the WSDL document structure, including analyzing service names, operations and inputs/outputs. In our work, we employ the similarity model from Woogle[2] to compute the similarity of service operations, inputs and outputs.

We note the precision by similarity match in Woogle is limited to 60%~70%. The reason is that Woogle mainly focuses on high similarity cohesion at the operation level while taking little consideration of the business category that a web service belongs to. On the other hand, it might be more accurate to get the similar services in a same domain (for example, “NASDAQService” and
“StockQutoe” are very likely in “Stock” domain. Therefore, filtering the similar services searched by woogle with domain category, the precision might be much higher. One of the important features in UDDI is the “domainkey” that identifies the business category with its semantic information. As UDDI category is a tree-like structure, the similarity of domain category can be measured by the semantic distance between two nodes of domainkey. Thus we apply the domain similarity retrieval to reduce the discovery clustering range to improve the precision.

3.2 Pool WSDL Generation

When the similar services are discovered, they are grouped in a service pool. All providers are not changed or modified physically. Providers are just registered as a member in the pool and organized in a “loosely coupled” way. To help consumers request such “proxy provider”, a “virtual” WSDL should be generated.

Table 1 The naming rules for pool WSDL

<table>
<thead>
<tr>
<th>Elements</th>
<th>Naming Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceName</td>
<td>The service name of the web service that holds the highest cohesion value.</td>
</tr>
<tr>
<td>Profile</td>
<td>The domain and functionality description of the service providers in the pool.</td>
</tr>
<tr>
<td>Op</td>
<td>The operation name of the web service that holds the highest cohesion value.</td>
</tr>
<tr>
<td>Input</td>
<td>The input name consumer inputs</td>
</tr>
<tr>
<td>Output</td>
<td>The output name consumer inputs</td>
</tr>
</tbody>
</table>

Table 2 The example for generating a pool WSDL

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Output</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1 GlobalWeather</td>
<td>ZipCode</td>
<td>CityTemperature</td>
<td>GetTemperature</td>
</tr>
<tr>
<td>S2 WeatherFetcher</td>
<td>PostCode</td>
<td>Temperature, WindChill, Humidity</td>
<td>GetWeather</td>
</tr>
<tr>
<td>S3 WeatherForecaster</td>
<td>AreaCode</td>
<td>LocalWeatherByAreaCodeResult</td>
<td>ForecastWeather</td>
</tr>
</tbody>
</table>

The structure of the pool WSDL is represented as the vector of SP = <ServiceName, Profile, Op, Input, Output >. However, the service pool may have a lot of member services, each of whom has its own operations and input/output names, so it is not easy to identify a common name for them. Then we sort the services in the pool by computing their cohesion value to the input/output that the consumer uses to search. The cohesion value is computed by using the KMP substring match algorithm [11]. We choose the ServiceName and Op of the service which have the highest cohesion values. The reason is that such naming rules make consumers feel the “service” discovered (in fact it is the service pool) meets their input/output requirements at most. In other words, the consumers will find the personalized web services which are named by themselves instead of service providers. Table 1 shows the naming rules for the attributes and Table 2 illustrates three services in the “weather report” category. If the keywords of input and output are “zip” and “temperature”, the best match is S1. And the pool WSDL can be briefly described as SP = <GlobalWeather, WeatherReport, GetTemperature, Zipcode, Temperature >.

It should be noted that every provider in the service pool is marked with a unique ID. Once the consumer actually requests the service, the pool will dispatch the request to the actual provider according to the mapping relationship.

4. QoS-Aware Service Discovery

4.1 Web Service Quality Modeling

Generally, QoS may cover a lot of attributes hosted by different roles. Web Service Quality Modeling [10] specifies the QoS for different roles and their measurement respectively. In our approach, we adopt four key attributes that consumers are mostly cares about when they use web service (Note that other QoS models can be applied to our approach without fundamental modification). We briefly formalize them as follows:

- Price Cost: the execution price cost is the fee that a service consumer has to pay for invoking the operation op of a web service.
- Response Time: it means the time taken to send a request and to receive the response. The Response Time is measured at an actual Web service call and it can be calculated by applying the following formula. ResponseTime = ResponseCompletionTime − ConsumeRequestTime.
- Successability: it is calculated as the number of successful response messages over the number of request messages. To measure it in a standard way, in this paper, we present successability in the log likelihood, \( LER = -\log(1 - Successability) \).
- Reputation: The value of the reputation is defined as the average ranking given to the service by consumers. We can get the reputation by the formula, which means the reputation of a web service is the average rank value by all consumers.

\[
Reputation = \frac{1}{n} \sum_{i=1}^{n} R_i
\]

4.2 Analysis of the Quality Model for Service Pool

4.2.1 Analysis of Quality Constraints

In our work [9], we have designed a series of algorithms to discover the service providers in service pool in much higher successability while keeping the lower response time. A service pool with n member services is denoted as Pool = \( \{S_1, S_2, ..., S_n\} \), and the selection of providers in the pool is denoted as a sequence.
$\text{Sel} = (a_1, a_2, ..., a_n)$, where $a_i$ is sorted by the descending order of successability/response time. The elements of Sel correspond to the index of $S$. When a request comes, the service sequence selected by the pool is determined by Sel. We hold an assumption that the provider in the pool is selected one by one. It means if $S_1$ is failed, $S_2$ will be selected. The selection sequence should satisfy the requirements of the four qualities above.

We are going to extend our approach in [7], which only considered the successability and response time. In fact, we find that consumers often present their QoS requirements by the constraints of strict maximum/minimum value (e.g. “response time must be less than 3 seconds, otherwise the service should fail”) and the expected value (e.g. “the average response time should be less than 3 seconds, if not, 3.5 seconds is also accepted”). In this paper, the strict maximum/minimum constraints are called the hard constraints and the expected constraints are called the soft constraints. The violation of hard constrains is difficult because some of them cannot be controlled by human (such as response time is related to the execution duration and network delay). In our QoS modeling, the successability is a statistic value, so its hard constraint is not applicable. The hard constraint of reputation cannot be guaranteed either, for reputation is an average value measured by multi users.

In our approach, we define a set of reputation levels of service pool and eliminate services under the corresponding reputation level. When the hard constraint of response time constraint is more than the timeout $T$, the service is failed. Then, denoted as $H(RT)$, obviously $H(RT) \leq T$. If users have no constraints of these qualities, we will use the default values. The constraint specification and default values are defined below:

Table 3 The constraint specification and default values

<table>
<thead>
<tr>
<th>Quality</th>
<th>Constraint</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successability</td>
<td>Soft only</td>
<td>0 for both Suc and LER</td>
</tr>
<tr>
<td>Response Time</td>
<td>Both hard and soft</td>
<td>The timeout constraint from the setting in middleware</td>
</tr>
<tr>
<td>Price Cost</td>
<td>Both hard and soft</td>
<td>The maximum cost value in the service pool</td>
</tr>
<tr>
<td>Reputation</td>
<td>soft constraint and reputation levels</td>
<td>The minimum reputation value of services in the pool of the corresponding reputation level</td>
</tr>
</tbody>
</table>

4.2.2 Formalized and Approximation Model

From the above discussion, the measurement of the QoS requirements is very complex. To simplify them, we need some preprocess before the selection.

We denote successability, response time, price cost and reputation with “Suc”, “RT”, “P” and “R” respectively. $S(Q)$ and $H(Q)$ denote the soft and hard constraint values of quality $Q$ respectively. $Q_i$ denotes the value of the service $Si$. The soft constraints are considered as the mathematical expectation of corresponding qualities. Therefore, $S(Q(Sel))$ and $H(Q(Sel))$ denote the mathematical expectation and maximum/minimum value of Q of the selection sequence $Sel = (a_1, a_2, ..., a_n)$ respectively. Then we define the measurement of the qualities by the following formulas:

$$H(RT(Sel)) = \sum_{i=1}^{n} RT_i, S(LER(Sel)) = \sum_{i=1}^{n} LER_i,$$

$$H(P(Sel)) = \max_{i=1}^{n} [P_i];$$

$$S(RT(Sel)) = \sum_{i=1}^{n} \left( \prod_{j=1}^{i} E R_{j} \right) RT_i,$$

$$S(Q(Sel)) = \sum_{i=1}^{n} \left( \prod_{j=1}^{i} E R_{j} \right) Suc_{i}, Q_i + \prod_{j=1}^{i} E R_{j}, Q_i,$$

$$Q \in \{ P, R \}, \ E R = 1 - Suc, \ \sum_{i=1}^{n} E R_i = 0.$$

We also hold two approximations here. First, we consider the response time is an integer value of a certain unit such as 0.01 second. Therefore the RT values can be discrete. Second, in practice, we find that the successability of most services are over 85%. Therefore, for the soft constraints of the RT, Price, Rep, the contributions of services from the third to the last one in the selection sequence Sel are less than 4%, which can be neglected (the proof and some experiment results can be seen in [6]). Then approximations are listed as follows:

$$S(RT(Sel)) \approx S(RT_{app}(a_1, a_2)) = RT_{app},$$

$$S(Q(Sel)) \approx S(Q_{app}(a_1, a_2)) = S_{app}Q_{app} + ER_{app}Q_{app}(Q \in \{ P, R \}).$$

where $RT_{app}$, $Q_{app}$ denote the approximate value for $RT$ and $Q$.

We assume that the services with reputations under the expected level have already been eliminated from the pool. With above definitions, we can formalize the QoS-Aware Pooling Problem:

**Definition 1:** Given a service pool $Pool = \{ S_1, S_2, ..., S_n \}$ and constraints $S(Q)$, $H(Q)$, where $n$ is the number of services in the pool and $Q \in \{ LER, RT, P, R \}$, $Q' \in \{ RT, P \}$. The **QoS-aware pooling problem** is transformed to the selection of a sequence of indices $Sel = (a_1, a_2, ..., a_n)$ from the pool satisfying the following conditions:

$$S(Q_{app}(a_1, a_2)) \leq S(Q)(if \ Q \in \{RT, P\});$$

$$S(R_{app}(a_1, a_2)) \leq S(R);$$

$$S(LER(Sel)) = \sum_{i=1}^{n} LER_i \geq S(LER),$$

$$\forall i, a_i \in Sel, P(a_i) \leq H(P),$$

$$H(RT(Sel)) = \sum_{i=1}^{n} H(T_{app}) \leq H(RT).$$

4.3 QoS-Aware Discovery Algorithm

4.3.1 The Index-S(RT)-LER Matrix

We assume that Pool has already been sorted, i.e. $P_i \leq P_{i+1} (1 \leq i \leq n)$. Holding such assumption, the constraint $H(P)$ can be transformed to the constraint of the index. Formally, the constraint
∀i, a_i ∈ Sel, P(a_i) ≤ H(P)

∀i, a_i ∈ Sel, a_i ≤ k = Transform(H(P))

By binary search, such transformation can be done within the complexity log(n).

It can be proved that the discovery of a selection sequence with maximum LER with H(RT) constraint is a knapsack problem, which is NP-hard [11]. However, there have been some polynomial solutions for integer-value knapsack problems by constructing an integer table. Similarly, with the discrete assumption of RT, we also design a polynomial algorithm for the QoS-aware pooling problem. The main idea is to construct the Index-H(RT)-LER matrix (“IHLM” for short). For indices pair <i, j>, Pool′ = \{S′_1, S′_2, ..., S′_t\} is the set of services by eliminating S_i, S_j from Pool, and sorted by the ascending order of P. Formally,

**Definition 2**

Set f(m): S'_m = S_[f(m)], where

\[ f(m) = \begin{cases} m & (m < m \in i, j), \\ m + 1 & (m \in i, j) \leq m < m \in \max \{i, j\}, \\ m + 2 & (m \geq \max \{i, j\}) \end{cases} \]

IHLM(i,j) is a matrix of index pairs:

\[
\begin{bmatrix}
\lambda(0, 0) & \lambda(0, 0.01) & \cdots & \lambda(0, 1.00T_i) \\
\lambda(1, 0) & \lambda(1, 0.01) & \cdots & \lambda(1, 1.00T_i) \\
\vdots & \vdots & \ddots & \vdots \\
\lambda(n - 2, 0) & \lambda(n - 2, 0.01) & \cdots & \lambda(n - 2, 1.00T_i) \\
\lambda(n - 1, 2) & \lambda(n - 1, 0.01) & \cdots & \lambda(n - 2, 1.00T_i) \\
\end{bmatrix}
\]

where \( T_i = T - RT_{ij} \).

\[ \lambda(k, t) = \min \{M(j, k), \lambda(k, t), \lambda(k - 1, t), \lambda(k - 1, t - RT_{ij}), \lambda(k - 1, t - RT_{ij}) + LER_{f(k)}(j)\} \]

Then, the matrix (6) can be constructed by algorithm **IHLMConstruction** as follows:

```
Algorithm IHLMConstruction(i,j)
Define \( \lambda(k, t) = \min \{M(j, k), \lambda(k, t), \lambda(k - 1, t), \lambda(k - 1, t - RT_{ij}), \lambda(k - 1, t - RT_{ij}) + LER_{f(k)}(j)\} \)

FOR i=1 TO T_y STEP 0.01
  \( \lambda(0, t) = 0, 0 \)
FOR k=1 TO n
  FOR i=1 TO T_y STEP 0.01
    IF (t < RT_{f(k)})
      \( \lambda(k, t) = \min \{M(j, k), \lambda(k, t), \lambda(k - 1, t), 0 \} \)
    ELSE IF \( M(j, k) \geq M(j, k - 1, t - RT_{f(k)}) + LER_{f(k)}(j) \)
      \( \lambda(k, t) = \min \{M(j, k - 1, t), 0 \} \)
    ELSE \( \lambda(k, t) = \min \{M(j, k - 1, t - RT_{f(k)}), LER_{f(k)}(j), 1 \} \)
  RETURN \( \{ \lambda(k, t) \} \) (as a matrix)
```

The above algorithm also proves the existence of the IHLM for every pair (i,j). Here, \( \lambda(k, t) \) records the maximum LER value among all selections with the first k services in Pool’ and time constraint t. It also indicates whether the k-th service can be included in one of the selections.

**4.3.2 Selection by Pair Searching**

The IHLM can deal with three constraints (H(RT), H(P) and S(LER)). In our approximate model, the rest three constraints (S(RT), S(P) and S(R)) are sensitive to only the first two services. To consider all six constraints, the pool searches all pairs filtered by index constraint (S(P)). For S(RT), S(P), and S(R), by the function Check(i,j), pool’ = \{S′_1, S′_2, ..., S′_t\} is the set of services by eliminating S_i, S_j from Pool, and sorted by the ascending order of P. Formally,

```
Algorithm SelectBySuc(i,j,H(RT),k)
\[ a[0]=i; a[1]=j; \]
Tag=2; T=S(RT);
IF \( (T - RT_i - RT_j < 0) \) RETURN NULL
ELSE FOR k = g(k) TO 0 STEP -1
  IF (MER(K,T)=0) BREAK
  ELSE IF (Bool(K,T)=1)
    a[Tag]=f(K); Tag=Tag+1; T = T - RT_{f(k)};
RETURN a[1]
```

Then algorithm **Discovery** is to select services according to the consumers’ request.

**4.3.3 Complexity Analysis**

For each pair of (i,j), the complexity of algorithm IHLMConstruction (i,j) is O(nT), where T is the value of
timeout. As the complexity of IHLMConstruction \((i,j)\) may reach \(1000nT\). It leads to the \(1000n3T\) complexity for algorithm Update. However, this level of complexity is acceptable. In our mechanism, the pool server updates QoS attributes from time to time. The overhead of updating is not very acceptable. On the other hand, by analysis of the algorithm Discovery, the time complexity for a single request is \(O(n^2)\). It is acceptable as the number of services in the pool is usually less than 100. The \(O(n^2)\) complexity may be better than algorithms based on integer programming when the service size increases. Even in the worst cases, the integer programming method may reach the exponential complexity.

Obviously, the pool is actually a “proxy” delegating consumer requests to providers. When the number of providers and consumers grows, the pool would inevitably become a bottleneck and makes the performance degrade. We have taken consideration of alleviating these troubles. To solve the discovery overhead, we identify two different pool types: persistent and personalized. The persistent pool serves for those who have relatively steady requirements for a specific task. The personalized pool is constructed just-in-time for the personalization work and survives only in a session lifecycle. For the persistent pool, the pool server can store the discovered results of steady providers in a cache to alleviate the discovery workload. To reduce the routing overhead, the pool also stores the stubs in a cache. We use AJAX (Asynchronous Javascript and XML) in the client-side web browser to asynchronously download corresponding stubs just-in-time when the providers are discovered. In this way, the consumers can invoke the service in an end-to-end pattern instead of routing from the pool server.

5. **Service Subscription Runtime**

Once the discovery finishes, the service pool will dispatch the request to the target service provider and returns the result. In our work [7], we employ a server that manages the pool in aspects of service clustering, discovery and execution. The server records the services registered in the pool, including the service URL, service names, operations and inputs/outputs. At runtime, the pool server looks up the target service, dynamically compiles the stub for the service by reflection, delegates the request and finally returns the invocation result to the consumer client.

6. **Demonstration and Experiments**

We have implemented the service pool server including the service publish interface and service selection engine. The detailed design can be found in [6] and [7]. We present the user interface for service discovery and subscription, as shown in Figure 1. Such UI is so friendly that: (1) the users can do both basic...
keyword search and input/output search, and the discovered results are presented as a single provider; (2) the whole QoS spectrum of current pool is adjusted just-in-time according to the users’ preference.

Figure 2 shows the precision comparison of the service pool, woogle, and keyword search. The X-axis means the top-k results returned, Y-axis shows the precision. We setup a UDDI repository with more than 450 web services from some authoritative UDDI [12][13][14]. We evaluate the precision by searching three services: WeatherReport, StockQuery, and BookTickets. Then we compute the average precision of the top five, top ten and top thirty results returned respectively. From the curve, the keyword search holds the lowest precision, while the service pool and woogle demonstrate that considering more criteria of search will increase the precision. Although the experiment shows the precision of all three approaches decreases when the number of returned results increases, our approach is better than the other two. The reason is that by category filtering, more function-similar services are likely to be grouped and returned.

To evaluate the efficiency of this algorithm, we also design a BackStracking selection algorithm and make a comparison. The BackStracking algorithm neglects the selection sequence after the third provider in the pool. We simulate the selection time overhead with pool by increasing the services in the pool (called pool size). Once the pool size is smaller than 10, the difference between the two is so tiny as to be neglected, however, once the size is bigger than 15, the IHLM based discovery shows its efficiency than the Backstracking one. From our experiment, the main cost of our algorithm is the construction of IHLM matrix, and the matrix will not be reconstructed until the QoS values are updated. But the BackStracking will try to cover the possible selections every time. Especially, when the pool size is bigger than 20, the BackStacking algorithm cannot work for the time overhead is over 20,000 ms and even causes memory leak. On the other hand, although the selection time of our algorithm increases with the pool size, it does not cause unacceptable overhead. In fact, the size of a pool is usually smaller than 25 from our investigation, so we consider it can well work in most cases.

7. Related Work

The more services available on the internet, the more efforts the consumers would pay for identifying the most adequate ones. To reduce the manual operation, it is a very natural idea to group the similar services together. In fact, in the project WS-CatalogNet [3], Benatallah et al. has proposed an approach called self-organizing catalog service communities, to enable a potentially large number of catalogs to act as one catalog to serve customer’s queries. The catalog service communities are catalogs catering for similar customer needs grouped together. They discussed about how the communities are built, linked for interaction, and constantly monitored and adapted over time. Especially they summarized a set of P2P query patterns to enable the member selection in and out of communities. Our work motivation is very similar, where the service pool can be viewed as communities. Although, there are two main differences: (1) we pay more attention to the built issues by retrieving the underlying similarity and (2) we try to investigate the optimal QoS tradeoff algorithm from the consumer’s perspective on the service subscription. In our future work, we plan to apply the query patterns in SELF-SERV for more efficient service discovery.

Similar to service pool, Hu and Huai proposes service club in the CROWN project [5]. CROWN aims to empower in-depth integration of resources and cooperation, and the service discovery is the kernel in the project. To optimize the discovery performance, service club is efficient by restricting the search range on a smaller level to faster discovery. Compared to service club, the goal of our approach is not only on discovery performance, but also on discovering the most adequate provider by evaluating the consumer’s preference. In fact, as the identical services are physically distributed over internet and provide similar functionalities, they can also be viewed as a service grid.

Our work aims to provide a more “consumer-centric” approach to improve current service discovery and subscription. In fact, W.T. Tsai, et al considered current
SOA is producer-centric, in which the service providers publish services that they produce and let the consumers to search available services to compose their applications [1]. They proposed the GSE (Global Software Enterprise), to support the application builders to publish their application requirements for the service providers to follow when producing or customizing services to support the application. Compared to their work, our approach focus on the “service aggregation”, and pays much attention to simplify the consumers’ task. It makes sense that these two works can cooperate, for example, the requirements templates on GSE can be reused in the search phase.

8. Conclusion and Future Work

With the ever increasing number of service providers on internet, the consumer may have trouble in the tedious and time-consuming search for services. Aggregating function-similar web services is considered as a promising way to alleviate the consumers and provide flexible quality of service. This paper provides a novel approach aggregating services from the perspective of consumers instead of providers.

The most important contribution of this paper is to make the aggregation and usage of similar web services on-demand, user-friendly and efficient. On-demand means the aggregate is driven by consumers instead of providers. User-friendly means consumers do not select services, handle different WSDL of similar services and switch between services at runtime any longer. Efficient means it integrates an efficient service search engine, reduces the incorrect services by some filter, discover the aggregate by a polynomial complex algorithm.

The future work includes: refining the clustering algorithm by retrieving more web services to get more precise similarity and enhance the discovery algorithm in the dynamic environment.

9. Acknowledgements

This work has been supported by the National Grand Fundamental Research 973 Program of China under Grant No.2005CB321805, the National Natural Science Foundation of China under Grant No. 90412011, 90612011, 60403030.

10. References


