HAL open science

Dynamic Strategies for Query Constructing and Rank Merging from Multiple Search Engines

Kobkaew Opasjumruskit, Birgitta König-Ries, Jesús Expósito

To cite this version:

Kobkaew Opasjumruskit, Birgitta König-Ries, Jesús Expósito. Dynamic Strategies for Query Constructing and Rank Merging from Multiple Search Engines. 4th European Conference on Service-Oriented and Cloud Computing (ESOCC), Sep 2015, Taormina, Italy. pp.141-155, 10.1007/978-3-319-24072-5_10 . hal-01757570

HAL Id: hal-01757570

https://inria.hal.science/hal-01757570v1

Submitted on 3 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Dynamic strategies for query constructing and rank merging from multiple search engines

Kobkaew Opasjumruskit, Birgitta König-Ries and Jesús Expósito

Heinz-Nixdorf Chair for Distributed Information Systems, Institute for Computer Science, Friedrich-Schiller-University Jena, Germany
<firstname.lastname@uni-jena.de>

Abstract. Heterogeneous search engines differ in the algorithms they use and the domains they cover, thus there is no single search engine that performs best in every circumstance. In order to obtain optimal search results, it often makes sense to use more than one search engine. However, appropriately merging results from different engines is challenging, i.e. combining results in such a way that they reflect the ranking of results the user would choose. In this paper, we propose an effective way to achieve this for web services search which can be extended to cloud services and be applied to big data. In contrast to "classical" search processed by conventional text-based search engines, a more elaborated search request is needed here. In addition to the result merging, we therefore present a method to create a structured request for this specific task. The evaluation of our proposed solution shows that it is satisfying in terms of both result quality and performance.

1 Introduction

Search engines have been around for a few decades and not only provide access to textual information, in other words digitalized data, but also to other types of information sources and to functionality offered online, e.g. smart objects, sensors and actuators around us. Moreover, the discoverable information can be presented either in plain text formats (like contents on web pages or text files) or in structured formats such as XML or RDF.

There is a plethora of search engines which differ in specialties and algorithms. Each search engine has its own benefits and drawbacks. Therefore, if a user wants to be sure to obtain the optimal result, he or she has to make multiple queries on several search engines and integrate the results manually. Additionally, the quality of the search results of a specific engine depends on keywords and their structures provided by the user.

As an example, consider the search for web services. We compared the quality of results from different search engines as depicted in Table 1. First, we prepared three different search queries (i.e. "camera price", "book price" and "city country hotel"). Secondly, we employed three different search engines specialized in web service searching (i.e., two variants of iSEM [10] and SeMa2 [18]) and one text-based search engine (i.e. ElasticSearch [1]). Thirdly, we measured the precision,

Table 1. Service matchers' performance depends on queries.

Query	camera price				book price				city country hotel			
	iSem (Cos, structured)	**iSem (Cos)**	SeMa2	ElasticSearch	**iSem (Cos, structured)**	iSem (Cos)	SeMa2	ElasticSearch	iSem (Cos, structured)	iSem (Cos)	**SeMa2**	ElasticSearch
Precision	0.50	**0.60**	0.40	0.40	**0.80**	0.80	0.40	0.90	0.30	0.30	**1.00**	1.00
Recall	0.71	**0.86**	0.57	0.57	**0.80**	0.80	0.40	0.64	1.00	1.00	**1.00**	0.59
F-measure	0.59	**0.71**	0.47	0.47	**0.80**	0.80	0.40	0.75	0.46	0.46	**1.00**	0.74
nDCG	0.91	**0.90**	0.82	0.73	**0.94**	0.87	0.74	0.83	1.00	1.00	**1.00**	0.62

recall, F-measure and nDCG (normalized Discounted Cumulative Gain) rates to compare the quality of search results of the search engines. Though "iSem text similarity (Cos, structured)" search engine performed best with the "camera price" query, the other search engines yielded better quality of results when we used other queries. Moreover, the overall quality of semantic search engines surpasses the text-based search engine. This advocates the need to use several search engines relying not only on a conventional text-based search.

There are several researches, e.g. [16] and [7], trying to take advantage of multiple search engines to improve the quality of search results. This way, one can always select the best result from all of them. One challenge in this approach is how to properly rank the results from search engines into one list. Assume that the search engines return match values between 0 and 1 for the individual services, where 1 means the services matches perfectly and 0 means the service does not match at all. In such a setting, for instance, results from a search engine A could b S1(0.9), S2(0.8), S3(0.75), and results from a search engine B could be S4(0.95), S1(0.8), S2(0.9); where Sn means an item of results, and a number in brackets is a similarity score between that item and the query. What would be the best way to integrate these results? Is S4 with 0.95 score from search engine B better than S1 with 0.9 score from search engine A?

Another challenge is how to convert a user's simple text input into a structured request, which is varied by different search engines. For example, the user is looking for a data in which containing "pricing" with in an attribute name "service", not in any other attribute. Search engine A may be able to search the data in XML format, while search engine B can search only for JSON format data. In order to use both search engines, this user needs to create two types of requests. This can take a great effort for the user to learn how to construct a request that conformed to different search engines. Therefore, we need to create a tool to help the user for these conversions.

Together with semantic technology, it is possible to improve the search process, where the search engine understands the meaning of keywords and can search not only "literally" but also "semantically".

As a tangible use case, we can apply the approaches of using multiple search engines and creating a structural request message to service discovery as proposed by [22]. Imagine that Brian needs to check whether a light in his study room is turned on or off. Since the light switch in this room is connected to the local network, he can check its status via a web service. But he does not know which one is belong to his study room. Fortunately, these devices are provided with standardized and machine-interpretable service descriptions which can be optionally extended by semantic annotations. Thus, instead of remembering all names of sensors in his house, he can input a simple keyword to the service discovery engine to look for the right sensor.

According to [3], there is currently no single global set of standards for the service descriptions, and in all likelihood never will be. This is also true for the service description language and service matching algorithm, since numerous of them have been developed and been widely in use for a while. They all have their own pros and cons; consequently, it is not deducible which one is the best for every circumstance. One optimal solution is to apply all prominent description languages and service matching algorithms together in one single discovery process as proposed in [22]. To realize such an approach, we need to compose a request which can be interpreted by service matching engines. Afterwards, a meaningful strategy to merge all outcomes is needed to create a single set of search results.

This paper is structured as follows: the technical background and related work will be reviewed in the next section. This also raises research questions which became requirements for our approach. Consequently, the solutions section will elaborate in technical detail how we fulfill the requirements. The evaluations section advocates the solutions we proposed, and finally, we conclude our work in this paper and plan for further development.

2 Literature Review

In this section, we provide an overview of existing techniques for result merging and discuss their respective advantages and drawbacks. We then focus on the specific type of search engine used for this evaluation, namely (semantic) web service search engines and the underlying service descriptions.

2.1 Result merging

[16] and [7] discussed different methods of merging multiple search engines' results. These techniques can be categorized into three types: score-based, rank-based and content-based.

The *score-based method* is the simplest technique. Assuming all search engines have comparable similarity scores, then all the results can be merged by linear combination methods discussed in [25], which accumulate each item's normalised score from all search engines and reorder them to a final ranked list. However,

this approach does not take into account that different search engines differ in their reliability. Thus, this does not work well in practice.

Content-based approaches like Search Result Records (SRRs), Top Document (TopD) and their successors are claimed to be the most effective [16]. However, they are rather heavy-weight since they need to download documents for analysis. Besides, the content-based algorithms are already used by search engines. Thus, there is no need to repeat the algorithm again in our approach.

The *Rank-based method*, which assigns each item a score corresponding to the rank in which it appears within each search result, is simple and versatile. It neglects the original scores from the search engines, on the other hand, assigns a new score to each item. This does not require document analysis, so it can save time and memory consumption. The simplest rank-based method is to consider the best rank from all search engines directly using voting systems as discussed in [23], like Borda's Positional method. Moreover, when taking the reliability of search engines into account, we can use an approach like weighted Borda-Fuse.

Borda count [23], a voting-based data fusion method, is also simple and effective in term of quality and time consumption. In Borda count, each search engine represents a voter. Each voter ranks a fixed set of candidates according to preference. The top rank will gain the highest score. Consecutive ranks will get fewer score. Finally, all scores of each candidate will be collected from all voters. The item which gets the highest sum of scores will be placed in the first rank and so on. In this work, Borda count is used for assigning scores to each item from search engines' results. Total scores will be arranged to provide the final result and also be used to calculate a weight value for each search engine. These technical details will be elaborated in the solution section.

2.2 Service description

The service description plays a key role in the service discovery engine since we assume that all devices are available as web services which must be supplemented with basic information in WSDL (Web Services Description Language) [2], a recommendation by the World Wide Web Consortium.

For an automatic service discovery, the WSDL description alone is inadequate unless this description is enhanced with a semantic annotation. This problem is addressed by SAWSDL (Semantic Annotations for WSDL) [6], which allows the extension of WSDL descriptions with semantic annotations from arbitrary ontologies. As one example for a more heavy-weight semantic description language, we also cover OWL-S (Web Ontology Language for Web Services) [17] in our evaluation.

Based on these service descriptions, there are several promising matchmakers as reviewed by [21] and [29]. In order to evaluate all the techniques based on different description formalisms, there are competitions, e.g. the Semantic Web Services Challenge [5] and the S3 Contest [9]. According to the results of these contests, we collected some auspicious approaches like SAWSDL-iMatcher [30], iSEM, OWLS-MX [12, 11, 13], etc., and used those in our evaluation.

3 Our Solution

To assist a user in finding appropriate devices in dynamic situations, an environment-adaptive service discovery engine has been developed. This service discovery engine appears to users as a search engine. Consider that a request from a non-tech-savvy user would be simple keywords, e.g. 'my location', 'heart rate', etc.

Our first challenge is to construct a query with a specific format in order to be comparable with service descriptions. Secondly, when service descriptions contain synonyms instead of an exact term the user is looking for, we can apply semantic annotations to the query so search engines can discover them. Thirdly, to cope with different standards of description, we need a request converter. When the request is ready, then we deploy service matchers, which can be replaced by any matcher that complies with description standards.

Each service matcher can have their own algorithm in assigning a similarity score to each ranked result, thus we cannot compare similarity value between two ranks from different matchers directly. However, we need several matchers to boost the result quality. So a ranks merger, in other words, a result integrator must be able to calculate and formulate the final ranking with the best quality.

Therefore, our proposed service discovery engine is separated into four modules: a request constructor, a request converter, a service matcher and a result integrator as shown in Fig. 1.

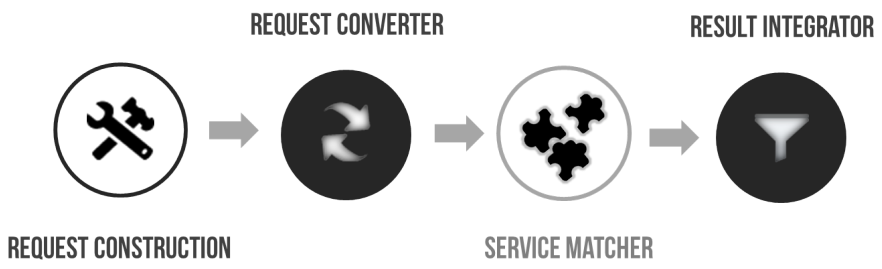

Fig. 1. A building block diagram for the service discovery.

3.1 Request constructor

This module is responsible for constructing a meaningful query message out of the user inputs in free-text format. By using a semantic search engine, we can retrieve synonyms or relevant terms to enhance the search result. Afterwards, these terms will be separated into input, output and operational descriptions to be mapped with the predefined description format, which is WSDL in this work.

The number of irrelevant services is typically larger than the number of relevant services, therefore the keyword and synonyms are used to filter out irrelevant descriptions. This will save more time in the service matching process.

Additionally, service descriptions are either in XML or RDF structures, thus making them accessible to an attribute or value of node level. For example, if a user wants a service that provides him a price of a specific model of a camera, he

will look for a service which contains "camera model" in input, "price" in output, and "find" in an operation description. It is more tangible to process this query by a structural search rather than a conventional text-based search, i.e. accessing the input, output, and operational attributes in each service description and comparing them with the user's query as exemplified in Fig. 2.

Furthermore, the user does not need to know how to construct these structural descriptions. The request constructor requires at least one simple keyword to create a structural request automatically.

3.2 Request converter

As several service description languages exist, in order to find all relevant services, the request needs to be converted into appropriate input formats for search engines supporting these different languages. Therefore, the request converter is responsible for translating the previously constructed description into other formats such as OWL-S or SAWSDL. By using a mapping schema, we can convert a request from a user into different formats and this can be extended for other prominent formats, e.g. JSON, tagged-based description when they are available.

Fig. 2 shows the common placeholders where we can insert mandatory information in order to create a description in various formats. Please note that this only works because current search engines do not take advantage of the full power of semantic descriptions; therefore, it is possible to automatically create appropriate queries with rather little effort. More powerful search engines would require more effort in query generation. Semantic annotations, which are optional here, can be filled in with information from user preference or user context if they are available.

Service request from a user	SAWSDL
Input: GPS location Output: Emergency Unit Behavior: Find ambulance service	<wsdl:description ...> ... <wsdl:input sawsdl:modelReference="http://.../protont.owl#Location" /> <wsdl:output sawsdl:modelReference="http://.../MedicalTransport.owl#Hospital"/> ... <wsdl:service name="FindAmbulanceService" >...</wsdl:service> </wsdl:description>
	OWL-S
	<rdf:RDF ...> <service:Service rdf:ID="FindAmbulanceService"/> <process:Input rdf:ID="Location"> <process:parameterType> http://.../protont.owl#Location</process:parameterType> </process:Input> <process:Output rdf:ID="Hospital"> <process:parameterType>http://.../ MedicalTransport.owl#Hospital </process:parameterType> </process:Output> ... </rdf:RDF>

Fig. 2. A user request and the resulting structured descriptions.

3.3 Service matcher

For this part we rely on existing work. We use SAWSDL and OWL-S service matchers for a proof of concept. The different approaches and description formalisms of each matcher assure that all suitable services will be discovered. Each matcher operates independently, thus the time consumption in this process can be reduced by using multithreading. Still, since we need results from all of the matchers, the runtime of the slowest matcher determines the overall runtime.

3.4 Result integrator

This work proposes a technique on how to retrieve the best quality result from multiple service matchers dynamically. First of all, results from search engines will be assigned scores based on their ranks by using Borda count as described in Section 2.1. With these scores, we can sort all search results into a list which will be compared with the original results from the search engines. Thereupon, the distance between the search results and the combined result will be calculated. The higher the distance value is, the less reliable that search engine will be. This reliability, i.e. weight, will be multiplied to the score computed by Borda count. Then, the merged list will be regenerated. This process will be iterated until there is no change in weight value between two consecutive iterations.

Consider that each matcher will return a set of results per query, which can be presented as an array, i.e. $[r_1, r_2, r_3, ...]$. Each element represents the unique ID of a service such as URI, and the similarity value of each result compared to the request. In our calculation, we apply several queries to measure average qualities. Therefore, each array from each query becomes a row of a matrix (R^k). Given that n = number of ranks, m = number of queries and k = number of matchers/search engines, the pseudo code depicted in Fig. 3 explains the result integrator's computation:

Result matrices (Rk) will be merged into a combined result matrix, $R^C = \begin{pmatrix} r_{q1,1} & \cdots & r_{q1,n} \\ \vdots & \ddots & \vdots \\ r_{qm,1} & \cdots & r_{qm,n} \end{pmatrix}$, by the getCombinedResult function (line 13).

Since not every matcher provides similarity scores, we estimate these scores from the result ranks. An element in the upper rank will get a higher score and vice versa. The formula for calculating the score, as shown in function getScore (line 32), is according to the normalized Discounted Cumulative Gain (nDCG). In addition, each matcher gets a reliability score, which is used as a multiplier to normalize the score with the other matchers. If there is no assigned weight, the integrator treats each matcher equally.

In function *getWeight* (line 38), a result matrix from each matcher, R^k, is compared with the combined result matrix R^C using the Euclidean distance method.

For testing the iteration condition, an *isWeightStable* function (line 50) compares weights between two consecutive rounds. When the difference converges to 0, indicating the stability of weight, the iteration will cease and the result from this round is supposed to yield the best quality.

```
FUNCTION resultIntegrator (ResultMatrices[k]):
  SET weight[k] to [1/k, ... , 1/k]
  SET CombinedMatrix to getCombinedResult(ResultMatrices, weight)
  SET newWeight[k] to getWeight(CombinedMatrix, ResultMatrices)
  WHILE ( !isWeightStable(weight, newWeight) ) DO
    SET weight to newWeight
    SET CombinedMatrix to getCombinedResult(ResultMatrices,weight)
    SET newWeight to getWeight(CombinedMatrix, ResultMatrices)
  END WHILE
  RETURN CombinedMatrix
END.

FUNCTION getCombinedResult(ResultMatrices, weight[]):
  DEFINE combinedResult[m][n]
  FOR each i in queries
    DEFINE rowResult[]
    FOR each j in ranks
      FOR each l in matchers
        SET URI to ResultMatrices[i][j][l].URI
        SET score to getScore(j, weight[l])
        IF rowResult has URI THEN
          add score to rowResult[URI].score
        ELSE add URI and score to the rowResult as a new element
      END FOR
    END FOR
    SET sortedRowResult[n] to n elements of the rowResult with the
         highest score sort
    SET combinedResult[m] to sortedRowResult
  END FOR
 RETURN combinedResult
END.

FUNCTION getScore(a, b):
  IF a = 1 THEN RETURN 1
  ELSE RETURN b/log2(a)
  END IF
END:

FUNCTION getWeight(combinedResult, ResultMatrices[]):
  DEFINE distance[k]
  DEFINE weight[k]
  FOR each l in matcher
    FOR each i in query
      increase distance[l] with Euclidean distance value
           between the combined matrix and the result matrices
         END FOR
    SET weight[l] to 1-(distance[l]/m)
  END FOR
  RETURN weight/sum(weight)
END.

FUNCTION isWeightStable(weight, newWeight[]):
 SET i to index of maximum element in newWeight[]
  IF (weight[i] - newWeight[i])/weight[i] < 0.05 THEN
    RETURN true
  ELSE RETURN false
  END IF
END.
```

Fig. 3. The Result Integrator Algorithm

To exemplify the process flow, assume that we use three matchers (k1, k2 and k3) and two queries on a set of services (S1 to Sn) and consider the top five ranks of the result. Then, we will have 3 matrices with dimension [2 x 5]. The three matchers might return: $R^{k1} = \left(\begin{smallmatrix} S2 & S4 & S3 & S7 & S1 \\ S6 & S9 & S1 & S2 & S5 \end{smallmatrix}\right)$, $R^{k2} = \left(\begin{smallmatrix} S2 & S4 & S3 & S7 & S1 \\ S6 & S9 & S1 & S2 & S5 \end{smallmatrix}\right)$, $R^{k3} = \left(\begin{smallmatrix} S2 & S3 & S4 & S8 & S6 \\ S9 & S6 & S1 & S7 & S3 \end{smallmatrix}\right)$. Next, the following steps will be executed:

1. Accumulating scores from each rank in each matcher (each element in R^k), e.g. service "S4" is ranked in the second place by matcher k1 and k2, while k3 ranks the service in the 3rd place, the total score for service "S4" is $\frac{1}{\log_2 2} + \frac{1}{\log_2 2} + \frac{1}{\log_2 3}$.
2. After finishing scoring all services for each query, the scores are sorted descending and the top 5 ranks are added to a row of the combined result matrix, i.e. $R^c = \begin{pmatrix} S2 & S4 & S3 & S7 & S1 \\ (3) & (2.63) & (2.26) & (1) & (0.86) \\ S6 & S9 & S1 & S2 & S5 \\ (3) & (3) & (1.89) & (1) & (0.86) \end{pmatrix}$.
3. The result matrices from the three matchers are compared with the combined result matrix to calculate the distances and the weights. The (average) Euclidean distances of the matchers are 0, 0, and 0.235. These figures will be used to calculate normalized weights, which become 0.34, 0.34 and 0.32.
4. The weight value of each matcher is multiplied to the original result matrix and the combined result matrix, R^C, is calculated again. From the previous step, the third matcher performs worse than the other matchers, thus the result from this matcher has fewer score than the rest.
5. The step 1-4 are iterated until the quality of returned results changes insignificantly, then the best combined result is completed.

From our previous experiment, a poor performance matcher weakened the overall quality of the final result. Thus, in Step 1 of each iteration, the matcher with the lowest weight below a threshold, $mean - 0.2(standardvariation)$ of all matchers, is removed. The number 0.2 in the formula is adjustable; however, we chose this value based on results of our previous evaluation. When this number is too high, there is more possibility that no matcher will be removed, but if this number is too low, it is more likely that an average performance matcher will be eliminated.

4 Evaluation

To evaluate the idea proposed in the previous section, we adopted service descriptions from the S3 Contest [9]. As it provides pre-defined solution sets, we can measure the quality of the result from our work. The binary quality measurement is calculated by precision, recall and F-measure rates. Meanwhile, to measure the quality of rank, the nDCG is used.

4.1 Test collections

The S3 Contest provides a sample set of service descriptions equally in SAWSDL and OWL-S formalisms. The total number of descriptions (for each formalism)

is 1080. The contest also provides 42 service requests together with the ideal solution of the matching task. There are 2 types of solutions; a binary relevance and a grading relevance. The former states only "relevant" or "irrelevant". While a graded relevance provides more detail of relevancy level, i.e., highly relevant, relevant, partially relevant, and irrelevant.

4.2 Service Matchers

According to the test collections, we also adopted service matchers from S3 Contest for the evaluation. The two categories of service matchers we are using are OWL-S-based and SAWSDL-based which handle different description formalisms.

The OWL-S matchers used in this evaluation are: EMMA [4], iSeM, SeMa2, SPARQLent [26], OWLS-iMatcher [8], OWLS-SLR Lite [20], OWLS-MX1 [12], OWLS-MX2 [11], OWLS-MX3 [13].

The SAWSDL matchers used in this evaluation are: iSeM-SAWSDL [10], SAWSDL-MX1 [14], SAWSDL-MX2 [15], URBE [24], LOG4SWS.KOM [27], COV4SWS.KOM [28], SAWSDL-iMatcher.

To simplify the evaluation process, which is getting more complicated towards the end of the building block diagram, some matchers that have unsatisfactory performances are neglected.

4.3 Evaluation results

The objective is to test whether the proposed technique can improve the quality of the service discovery process. The request construction and the request converter must be integrated to the whole process so that we can measure the quality of the result. It is unnecessary to evaluate the service matcher part since this was successfully done by efforts like [5] and [9]. Nevertheless, outcomes from the service matcher module are merged in a result integrator module.

Therefore, we focused in evaluating the result integrator in terms of result quality compared to an individual result of a single service matcher. Eventually, the overall time performance was measured and discussed in order to find a gap of improvement.

Result Integrator: By applying all descriptions from the test collections together with service matchers mentioned previously, the quality of a result integrator was compared with individual service matchers. Fig. 4 depicts a comparison in precision, recall, F-measure and nDCG between each SAWSDL matchers, a text-based search engine (Elasticsearch) and the outcome of the service integrator in each round. The weights of all matchers were calculated and illustrated in Fig. 6 (a).

In this experiment, the criteria to select a round that yield the best result was met in the fourth round. However, when we had a solution to compare with, the actual best result was from the third round. Since we cannot have the

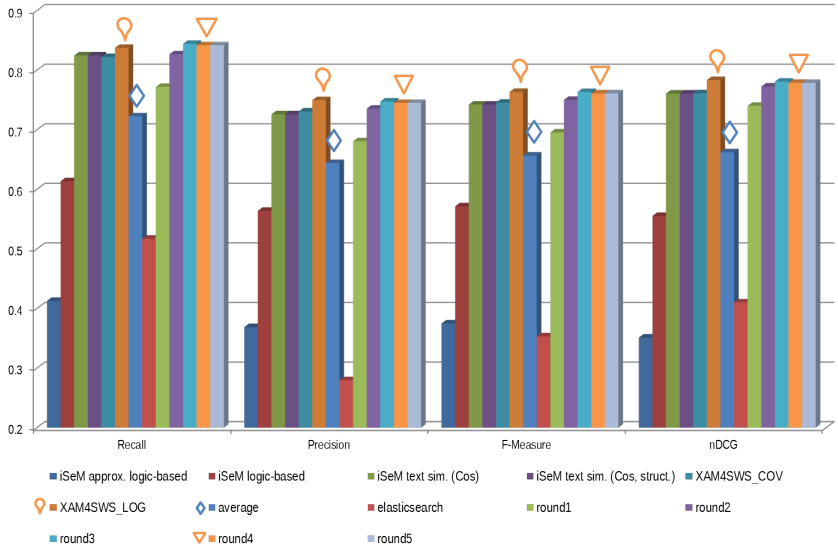

Fig. 4. Result integrator's quality in each iteration compare to each SAWSDL matcher.

Table 2. Evaluation result from the result integrator module

	Precision	Recall	F-measure	nDCG
Best SAWSDL Matcher	0.8377	0.7500	0.7638	0.7837
Average SAWSDL Matcher	0.7228	0.6444	0.6569	0.6624
This work on SAWSDL	**0.8421**	**0.7452**	**0.7613**	**0.7789**
Best OWL-S Matcher	0.9262	0.8167	0.8361	0.8539
Average OWL-S Matcher	0.6560	0.5762	0.5903	0.6101
This work on OWL-S	**0.9238**	**0.8143**	**0.8337**	**0.8673**

ground truth in the practical usage, plus, the difference between the third and fourth round were negligible, we can conclude that this technique produces a satisfactory result.

The same evaluation was applied to OWL-S matchers, see Fig. 5. From the weight measurement in Fig. 6 (b), the iteration stopped at round 7, which provided the best result.

From Table 2, our approach with SAWSDL matchers improved the recall rate compared to the best performing matcher by 0.5% while precision, F-measure and nDCG rates dropped by less than 0.7%. With OWL-S matchers, our approach boosted the nDCG by 1.57% whereas the other measurement declined by less than 0.3%. Nevertheless, comparing to the average performance of all matchers, our proposed solution outperformed them notably.

Overall performance: Up to now, we discussed how each component of a service discovery engine performs. Now we exhibit the overall run-time performance when everything is put together.

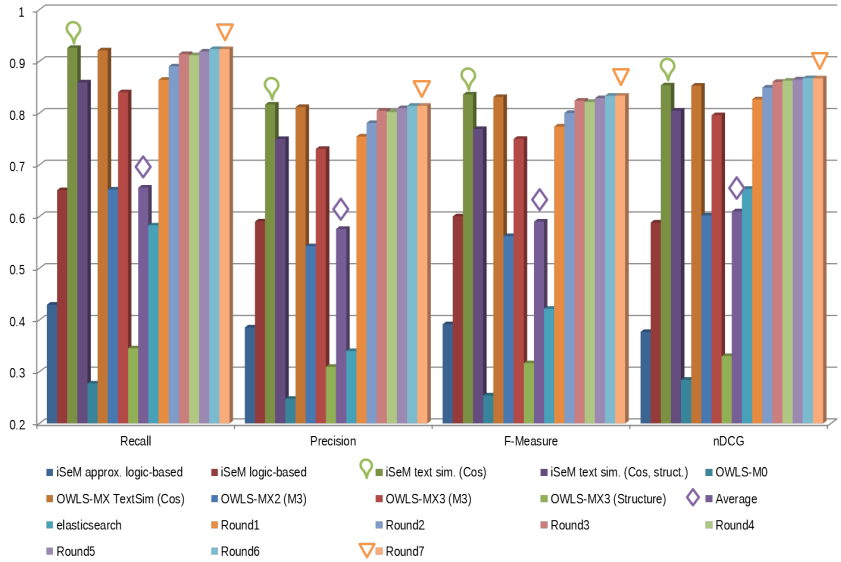

Fig. 5. Result integrator's quality in each iteration compare to each OWL-S matcher.

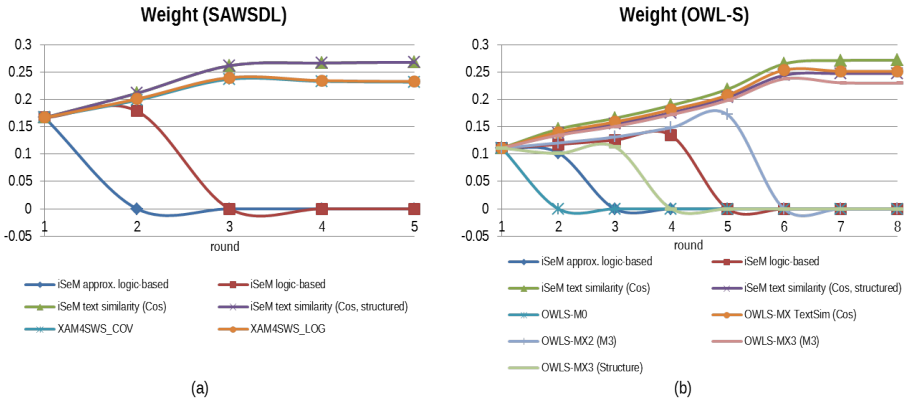

Fig. 6. Weight value of (a) SAWSDL and (b) OWL-S matchers calculated from each iteration.

First, we set three different factors; type of query, number of service matchers and number of processing thread. There are two types of query for the service discovery engine; a simple keyword that has no structure and a user's request containing input, output and operational keywords. The second type is called a multiple query since we need to query three times for semantic meanings. The more service matchers are used, the more time are consumed respectively. Thus, we deployed the multithreaded processing to the semantic search (in the query

construction) and the service matcher modules, which consumed 95% of total time in service discovery.

According to Fig. 7, the time consumption for each semantic query were accumulated when using a single thread processing. The multithreading can dramatically reduce the query construction time though the multiple query generally takes more time than the simple query. This is due to waiting for the slowest query to finish.

On average, the number of queries has a small effect on the service matching's runtime. Multithreading can reduce the time usage significantly though not by 4 folds as expected due to waiting for all matchers to complete their tasks.

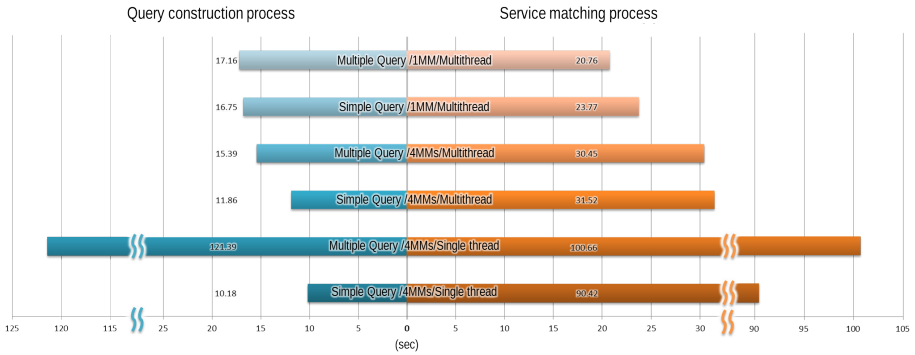

Fig. 7. Timing evaluation of a semantic search and service matcher modules.

5 Conclusion and future plan

The evaluation in this paper proved that our proposed technique can dynamically retrieve the best set of results from multiple search engines. The outcome of the request constructor and converter is a service description that is compatible with prominent service matchers in OWL-S and SAWSDL standards. The bottleneck of the total time performance occurred in the semantic search process is caused by an HTTP request to a remote server.

The result integrator produces a single set of results from multiple search engines with conceivable ranks. Additionally, from automatically calculated weights, we can rate the reliability of each search engine. This value can be recorded and used for statistical analysis further on. The time performance of this module is negligible compared to the semantic search and service matcher modules.

Our ultimate goal is to implement a user-friendly tool to efficiently utilize the benefits of the Internet of Things (IOT) [19] and to support big data. In the near future, we aim to integrate this work as the service discovery engine in MERCURY [22], a platform for user-centric integration, and management of heterogeneous devices and services via a web-based interface. Since MERCURY offers a great interface, it is practical to evaluate a usability test there. Moreover, we can extend the use case from local services to cloud services. We also consider supporting JSON and XML descriptions in the future development as well.

6 Acknowledgement

This research is accomplished under the framework of the Mercury project and was supported by IBM Germany Research & Development GmbH. We would like to specially thank you to Prof. Martin Welsch, a Chief Technology Advisor at the IBM Germany and Dr. Andreas Nauerz, a Lead Architect at IBM Laboratories, Germany for all supports and advices.

References

1. Elasticsearch. https://www.elastic.co/products/elasticsearch, accessed: 2015-10-07
2. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language. http://www.w3.org/TR/wsdl, accessed: 2015-10-07
3. Fleisch, E.: What is the internet of things? an economic perspective. Auto-ID Labs White Paper WP-BIZAPP-053, Zürich (2010), http://www.im.ethz.ch/education/HS10/AUTOIDLABS-WP-BIZAPP-53.pdf
4. García, J.M., Ruiz, D., Ruiz-Cortés, A.: Improving semant. web services discovery using sparql-based repository filtering. Web Semant.: Science, Services & Agents on the World Wide Web 17, 12–24 (2012)
5. Harth, A..M.D.: Semantic web challenge. http://challenge.semanticweb.org, accessed: 2015-10-07
6. Hobold, G., Siqueira, F.: Discovery of semant. web services compositions based on sawsdl annotations. In: IEEE 19th Int'l Conf. on Web Services. pp. 280–287 (2012)
7. Jadidoleslamy, H.: Search result merging & ranking strategies in meta-search engines: A survey. Int'l Journal of Comp. Sci. Issues (2012)
8. Kiefer, C., Bernstein, A.: The creation & evaluation of isparql strategies for matchmaking. In: The Semant. Web: Research & Applications, LNCS, vol. 5021, pp. 463–477. Springer (2008)
9. Klusch, M.: Semantic service selection (s3) contest. http://www-ags.dfki.uni-sb.de/ klusch/s3/index.html, accessed: 2015-07-10
10. Klusch, M., Kapahnke, P.: isem: Approx. reasoning for adaptive hybrid selection of sem. services. In: IEEE Int'l Conf. on Sem. Computing. pp. 184–191 (2010)
11. Klusch, M., Kapahnke, P., Fries, B.: Hybrid sem. web service retrieval: A case study with owls-mx. In: IEEE Int'l Conf. on Sem. Computing. pp. 323–330 (2008)
12. Klusch, M., Fries, B., Khalid, M.: Owls-mx: Hybrid owl-s service matchmaking. In: Proc. of 1st Int'l AAAI Fall Symp. on Agents & the Semant. Web (2005)
13. Klusch, M., Kapahnke, P.: Adaptive signature-based semantic selection of services with OWLS-MX3. Multiagent and Grid Systems 8(1), 69–82 (2012)
14. Klusch, M., Kapahnke, P., Zinnikus, I.: Hybrid adaptive web service selection with sawsdl-mx & wsdl-analyzer. In: Proc. of the 6th European Semant. Web Conf.: Research & Applications. pp. 550–564. Springer, Berlin, Heidelberg (2009)
15. Klusch, M., Kapahnke, P., Zinnikus, I.: Sawsdl-mx2: A machine-learning approach for integrating semant. web service matchmaking variants. In: 7th IEEE Int'l Conf. on Web Services (ICWS), L.A., USA. pp. 335–342. IEEE Press (2009)
16. Lu, Y., Meng, W., Shu, L., Yu, C., Liu, K.L.: Evaluation of result merging strategies for metasearch engines. In: Web Information Systems Eng, LNCS, vol. 3806, pp. 53–66. Springer (2005)

17. Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., Sycara, K.: Bringing semantics to web services: The owl-s approach. In: Semantic Web Services & Web Process Composition, LNCS, vol. 3387, pp. 26–42. Springer (2005)
18. Masuch, N., Hirsch, B., Burkhardt, M., Heßler, A., Albayrak, S.: Sema2: A hybrid semantic service matching approach. In: Semantic Web Services, pp. 35–47. Springer (2012)
19. Mattern, F., Floerkemeier, C.: From the Internet of Computers to the Internet of Things, LNCS, vol. 6462. Springer (2010)
20. Meditskos, G., Bassiliades, N.: Structural & role-oriented web service discovery with taxonomies in owl-s. IEEE Trans. on Knowledge & Data Eng 22(2), 278–290 (2010)
21. Ngan, L.D., Kirchberg, M., Kanagasabai, R.: Review of semantic web service discovery methods. In: 6th World Congress on Services. pp. 176–177 (2010)
22. Opasjumruskit, K., Expósito, J., König-Ries, B., Nauerz, A., Welsch, M.: Service discovery with personal awareness in smart env. In: Creating Personal, Social, & Urban Awareness through Pervasive Computing. pp. 86–107. IGI Global (2014)
23. Pacuit, E.: Voting methods. In: The Stanford Encyclopedia of Philosophy (2012), http://plato.stanford.edu/archives/win2012/entries/voting-methods/
24. Plebani, P., Pernici, B.: Urbe: Web service retrieval based on similarity evaluation. IEEE Trans. on Knowledge and Data Eng 21(11), 1629–1642 (2009)
25. Renda, M.E., Straccia, U.: Web metasearch: Rank vs. score based rank aggregation methods. In: Proc. of the ACM Symp. on Applied Computing. pp. 841–846 (2003)
26. Sbodio, M.: Sparqlent: A sparql based intelligent agent performing service matchmaking. In: Semantic Web Services, pp. 83–105. Springer Berlin Heidelberg (2012)
27. Schulte, S., Lampe, U., Eckert, J., Steinmetz, R.: Log4sws.kom: Self-adapting sem. web service discovery for sawsdl. In: 6th World Congress on Services. pp. 511–518 (2010)
28. Schulte, S., Lampe, U., Klusch, M., Steinmetz, R.: Cov4sws.kom: Information quality-aware matchmaking for sem. services. In: The Semantic Web: Research and Applications, LNCS, vol. 7295, pp. 499–513. Springer (2012)
29. Talantikite, H.N., Aissani, D., Boudjlida, N.: Sem. annotations for web services discovery & composition. Computer Standards & Interfaces 31(6), 1108 – 1117 (2009)
30. Wei, D., Wang, T., Wang, J., Bernstein, A.: Sawsdl-imatcher: A customizable & effective semant. web service matchmaker. Web Semant. 9(4), 402–417 (2011)