WEB SERVICE DISCOVERY FOR LARGE SCALE IOT DEPLOYMENTS

Yehia Elshater[1], Khalid Elgazzar[2], Patrick Martin[1]
[1]School of Computing, Queen's University, Canada
{elshater, martin}@cs.queensu.ca
[2]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
elgazzar@cs.cmu.edu

Abstract

IoT scenarios cannot tolerate long latency in finding relevant Web services to consume on the fly or dynamically integrate in IoT applications providing real time services. In this paper, we present a comprehensive Web service discovery approach for large scale IoT deployments. We leverage the information available in the Web service description document to cluster large service repositories into functionally similar groups to reduce the discovery latency and improve precision. Then, we use statistical indexing techniques to generate data structures for each cluster for fast and efficient matching. In this research, we propose and study the performance of four matching algorithms: semantic keyword matching using Normalized Google Distance (NGD), brute force search over Term Frequency-Inverse Document Frequency (TF-IDF) matrix, K-Dimensional (K-D) tree, and Locality Sensitive Hashing (LSH). Our thesis is that indexing-based discovery algorithms (i.e., K-D tree and LSH) provide a much faster response with comparable precision, while NGD and brute force search provide a slightly better accuracy, but at the cost of high latency. Our experimental results show that we can reduce the query latency by up to 5x fold, while achieving comparable precision with the state-of-the-art service discovery mechanisms.

Keywords: microservices, IoT, service discovery, Web service, feature extraction, TF-IDF, K-D tree, LSH.

1. INTRODUCTION

The Internet of Things (IoT) enables smarter interactions between people and billions of interconnected devices collecting real time sensor data. The real value of IoT, however, comes from IoT-based applications that leverage the IoT infrastructure and turn raw data into actions to provide value-added services to customers. To cope with the highly dynamic environments of IoT and meet both interoperability, loose coupling and real-time requirements, these application are typically developed using the microservices architecture [1], where services providing access to real time data can be dynamically discovered and integrated into an IoT application on the fly.

Several organizations [2] forecast that there will be more than 50 billion devices communicating with each other over the Internet by 2020, creating a much richer and sensor-dense environments than we see, can that even foresee today. We envisage that each of these sensors will expose their functionality as microservices in RESTful APIs for interoperability, ubiquitous data access and multimodality consumption (i.e, both users and applications). Hence, the arm-reach future will witness billions of these microservices that expose real time data access, posing significant challenges on service discovery mechanisms on how to efficiently discover services of interest in a near real-time fashion. This highlights the significance of efficient service discovery techniques that can locate relevant and quality services to meet on-demand requests.

RESTful services (aka. APIs) are capable of communicating with both applications and users through leveraging the built-in HTTP *Content-Type* header. XML or JSON-formatted responses can be sent to requests sent by applications, while human-readable responses in HTML format can be dispatched to the user's Web browser. This multimodal response feature is at the core essence of IoT principles, where both machines and humans can intercommunicate with services. However, this adds additional challenges on service discovery mechanisms from both scale and capability perspectives.

Microservices share common characteristics and are described the same way as standard Web services using WSDL 2.0. Since the release of the WSDL 2.0 in June 2007, RESTful service description became supported. However, microservices are inherently different from standard Web services from many dimensions including: extremely large number of offerings, dynamic binding, temporal and spatial dependency, on-demand provisioning, distributed nature, possible variation in performance due to hosting environments and persistent or transient network conditions. Hence, centralized service repository and indexing are infeasible. This makes discovery techniques developed for traditional Web service discovery inefficient to accommodate the dynamicity and scale for microservices and in general IoT scenarios.

Our vision is to break down these many challenges to facilitate efficient and robust service discovery. To this end, we assume that services could be published on geographically relevant repositories or geospatial indexing techniques [3] such as R-Trees [4] can partition Web services into spatially-bound categories of collocated services that are

maintained in geographically distributed service repositories at a metropolitan or even a finer granularity level. This enables search over the spatial properties of available and active Web services. However, this will still leave us with an enormous number of services that span a wide range of functionality spectrum, inhibiting the discovery process from meeting time and latency constraints. Search engines can become a bottleneck in the discovery process due to the significant amount of matching that they need to perform, especially if the service corpus is extremely large.Therefore, categorizing these services by functionality would limit the search space to the order of multiple thousands services per cluster. Then, statistical indexing techniques can facilitate rapid service discovery.

The problem of poor search recall for non-semantic Web services using search engines can be approached in three ways. The first approach is to perform a broad matching process and return a potentially large number of unranked services, most of which may not be of interest to the user. The second approach is to improve search engine retrieval with mechanisms to cluster services into similar functional groups while they crawl their description files [5]. This approach can effectively reduce the search space of Web services while improving the matching process. However, the response time of finding relevant services to a specific request depends on the size of the relevant cluster. The third approach is to build a statistical model that represents all the services and use indexing techniques to navigate the service corpus and match requests more efficiently [6]. This approach counts on building the statistical model and search index once offline and using them during runtime.

This paper proposes an efficient Web service discovery approach using statistical modeling and indexing techniques. First, we cluster large repositories into functionally similar groups to reduce the search space. Then, we generate a Term Frequency-Inverse Document Frequency (TF-IDF) model [7] that is used to build the TF-IDF matrix of each cluster and build a search index. When a discovery request is received, the system matches the request and returns a list of relevant services. We propose four matchmaking algorithms, namely NGD semantic matching, Brute Force search over TF-IDF matrix, K-D tree and LSH. To the best of our knowledge, this is the first research that uses K-D tree and LSH indexing techniques to improve Web service discovery.

The remainder of this paper is organized as follows. Section 2 highlights some of the related work. Section 3 provides an abstract view of the proposed discovery approach. Section 4 presents our clustering approach and briefly highlights relevant background. Section 5 explains in detail the proposed four matchmaking algorithms. Section 6 describes our experimental setup, performance evaluation model, and discusses experimental results. Lastly, Section 7 concludes the paper and outlines future research directions.

2. Related Work

Non-semantic Web services are described by Web Service Description Language (WSDL) documents [8], which provide details about the service functionality, data types, access protocols, and endpoints by which consumers can access the service functionality. Semantic Web services, on the other hand, use Web ontology languages (OWL-S) [9] or Web Service Modeling Ontology (WSMO) [10] as a description language. XML is used to construct the basic blocks of Web service communication through XML messaging protocols, such as SOAP [11] and REST [12]. Non-semantic Web services are more popular and supported by both the industry and development tools. The discovery process is quite different according to the Web service description method. Semantic Web services are discovered by high level matchmaking approaches [13], while non-semantic Web services discovery uses information retrieval techniques [14]. Our proposed *goDiscovery* approach targets the discovery of non-semantic Web services as they are more prevalent.

Non-semantic Web service discovery typically uses keyword matching to find relevant services to a user request. Discovery mechanisms either use clustering approaches to reduce the search space or indexing techniques to enable fast discovery over the whole corpus. Both approaches leverage Web mining to extract unique keywords and vocabulary that best represent the service functionality from Web service description files.

Woogle [15] is a Web-service search engine that supports simple keyword searches as well as similarity search for Web service operations, based on similar inputs and outputs. Basically, Woogle extracts input and output parameters of the Web service operations from the WSDL document and generates a bag of words (bow) for each service. Then, the authors use the TF-IDF to measure the similarity of two bags to find services with similar operations. The challenge with this approach is that there is no standard naming convention for the operation parameters, hence, parameter names are developer dependent and they may not be descriptive or even correct English words. Relying on such parameter names most likely leads to incorrect matching. Hao et al. [16] use the TF-IDF score to compute the similarity between Web service clusters. They propose two indexing structures of TF-IDF vectors to provide relevancy score and ranking of the output results. The authors show that the time cost of building these indices significantly increases when the number of Web services increases (e.g., it takes ~ 560 seconds to index 420 services). Moreover, the searching time is long if the user query contains many terms. The proposed search algorithm takes more time if the query contains multiple terms distributed over all the services (i.e., it acts like a brute force search). The TF-IDF model is also used to support search engines to remove the ambiguity of keywords in Web documents [17].

Nayak [18] proposes a method to improve the Web service discovery process using the Jaccard coefficient to calculate the similarity between Web services. He provides the user with related search terms based on other users' experiences with similar queries. Platzer et al. [6] propose a vector space search engine for Web services. Their approach produces a TF-IDF vector for the user query and computes the cosine distance for all Web services (i.e., using brute force methods). Although this approach produces good results, the brute force method is time-consuming and poses significant overhead on the query response time.

To the best of our knowledge, no previous research addresses the complexity of service discovery in the IoT

domain. We provide a comprehensive solution that better fit service discovery in IoT scenarios. Our approach includes service clustering to reduce the search space, and hence supports near real time discovery, fast extraction of search keywords from discovery requests. It also includes our lightweight *goDiscovery* approach [19] that leverages statistical models for fast and efficient service discovery. *goDiscovery* builds the TF-IDF model, K-D tree index, and Locality Sensitive Hashing (LSH) tables for the service corpus offline and uses them at runtime to match user queries. The maintenance of both the TF-IDF model, the K-D tree, and hashing tables (i.e., insertion of new services and deletion of stale services) is performed online and takes a similar amount of time as a request.

3. Overview of the Proposed Approach

We address the complexity of service discovery in IoT scenarios from different perspectives. We start with the assumption that services are published in geographically-bound repositories, where each repository serves a specific geographical region. Service providers can publish directly to their respective geographical registry (if publishing endpoint is known) or to a global service registry that maintains a global hierarchy based on services' locations. This registry forwards publishing requests (as well as discovery requests) to the appropriated repository. Consumers (both applications and users) may also send discovery requests directly to their local registry, if discovery endpoint is known. Figure 1 shows an abstract overview of the proposed approach.

In each repository, we run our clustering algorithm offline to categorize services into functionally-similar groups to reduce the search space and improve the discovery precision. Then, we generate efficient indexing data structures (K-D trees and/or LSH tables) for each group for runtime request matching. Each tree structure and LSH table has a universal unique ID. All the indexing data structures are placed in memory for better performance. At runtime, when a discovery request is received (step 1), the system extracts search keywords and generates a TF-IDF vector for the request. A classifier binds the request to the most relevant cluster to perform a search in its associated index (step 2). The retrieved list of Web services is sent back to the consumer to choose the service that best satisfies the desired functionality (step 3). In this article, we provide multiple matchmaking techniques for comparison purposes, each has pros and cons. We leave the choice of the matchmaking algorithm open to individual implementations based their own requirements and constraints.

4. Service Clustering

We cluster Web services into functionally similar groups based on their descriptions, so that, matching algorithms do not need to match the user's request against all the service offerings in the corpus, but rather with a particular set of services that share similar functionality. The clustering and classification of new services are performed offline to eliminate any overhead during service matching [5]. The service clustering renders large repositories of heterogeneous service offerings into multiple locally distributed domain-specific registries. Our clustering approach not only reduces the discovery latency by reducing the search space, but also improves the recall and precision via data mining and text analytic techniques.

RESTful services are described using WSDL 2.0. A WSDL description document contains all the details of a Web service including: the endpoint URL, supported communication mechanisms, operations, and message structure. Discovery mechanisms use these details to match service functionality with user requests and consumers use the information to interact with the service. We extract features from WSDL documents to cluster services based on functionally to bootstrap the performance of matchmaking algorithms.

4..1 WSDL 2.0 Document Structure

WSDL 2.0 is an XML language with the core namespace http://www.w3.org/ns/wsdl. The root element of a WSDL 2.0 document is the `<description>` element. There are four child description elements that encapsulate all the details of a REST Web service: `<types>`, `<interface>`, `<binding>`, and `<service>` as the following skeleton shows.

```
<wsdl:description
  xmlns:wsdl="http://www.w3.org/ns/wsdl">
  <wsdl:types/>
  <wsdl:interface/>
  <wsdl:binding/>
  <wsdl:service/>
</wsdl:description>
```

The `<types>` element is an XML type definition that describes the data containers used in message exchanges. The `<interface>` element defines the Web service operations (functions) that can be performed by the Web service. Each operation is associated with input and output messages with the order in which these are passed. The `<binding>` element specifies the communication protocol and data format for each operation. For REST Web services, that consumers communicate with the service using HTTP. The `<service>` element associates an endpoint address for the Web service with a specific interface and binding.

Liu and Wong [20] propose a service clustering approach that leverages Web crawlers to collect and cluster active online Web services. The authors apply text mining techniques to extract key features from the description documents such as service *content*, *context*, *host name*, and *name*. However, features such as *service context* (i.e., surrounding Web pages with the same domain name) and *service host name* offer little or no help in the clustering process as providers tend to advertise services on their own web site rather than UDDIs. Hence, surrounding pages have little or no relation to the semantic of the Web service. Our feature extraction process exclusively depends on information available in the description documents. Our set of features are experimentally proven to be more efficient and to provide better results [5]. In the following, we describe this set of features.

Feature 1: Content Words Extraction

Figure 2 shows the steps of the *Content Word Extraction* module to generate the content word vector for a Web

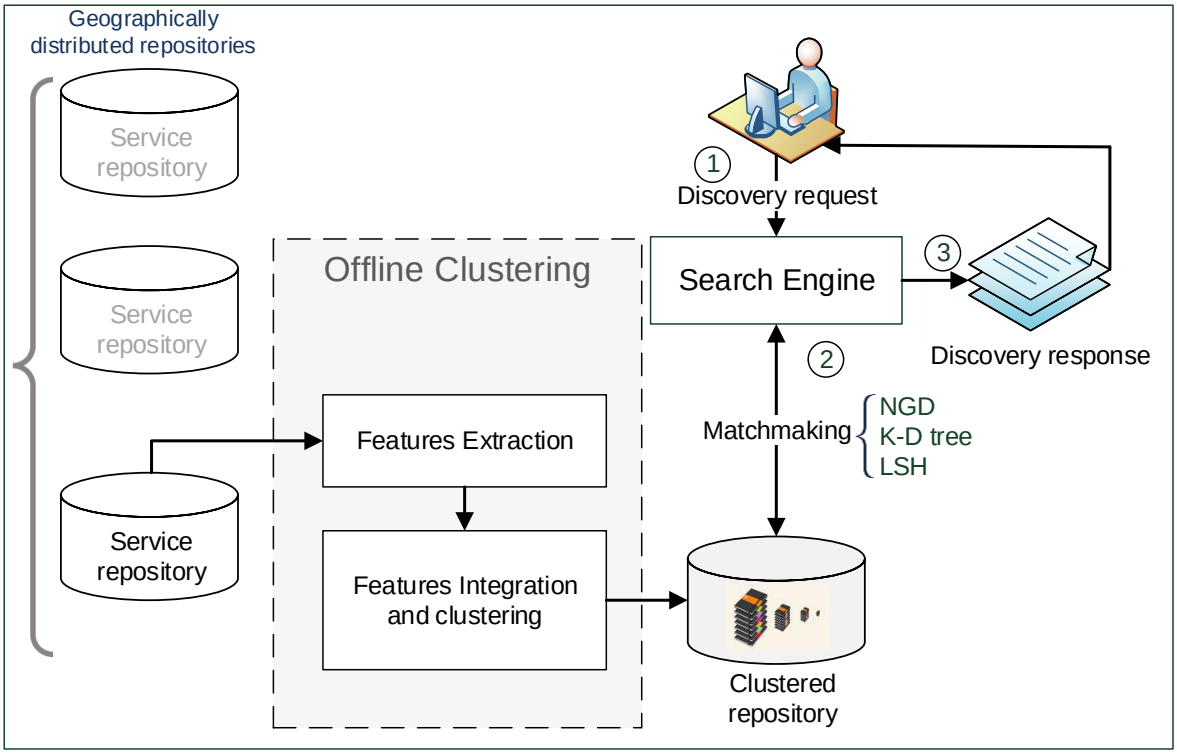

Figure 1: Schematic block diagram of the clustering approach.

service or a user query. The module reads the Web service description files to extract a vector of meaningful content words that best describe the service functionality. Steps to generate this vector are as follows:

1) **Parsing WSDL:** The contents of the document are parsed based on white spaces to produce a vector of tokens T_i.
2) **Tag removal:** The next step removes all tokens from T_i that are part of an XML tag so that only valid content words remain in the vector. Removing XML tags from the tokenized vector is straightforward since all XML tags used in WSDL documents are predefined.
3) **Word stemming:** In this step, all words in T_i are reduced to their base words using a Porter stemmer [21]. Tokens with a common stem will usually have the same meaning, for example, 'connect', 'connected', 'connecting', 'connection', and 'connections' all have the same stem 'connect'. Performing this reduction groups the occurrences of the word variations.
4) **Function word removal:** Function words (also known as *stop words*) tend to be independent of one another. Often, function words can be distinguished from content words using a Poisson distribution to model word occurrence in documents[20]. This step is intended to remove all function words from the service word vector. To decide whether a certain word w is a function word, we calculate the overestimation factor for all words in the word vector as follows:

$$\Lambda_w = \frac{estimatedDocumentFreq}{observedDocumentFreq} = \frac{\hat{n}_w}{n_w} \quad (1)$$

where n_w is the number of documents that contain the word w (we use the occurrence in Web documents), $\hat{n}_w$ and is the number of estimated documents that contain the word w. To estimate the document frequency for a word we need to have the document frequency for a single occurrence of that word in the Web corpus. While it is not feasible for a search engine to identify documents that contain a single occurrence of a particular search term, there are techniques, such as the K-mixture word distribution model [22], to estimate the word frequency using page count. We can then calculate the overestimation factor for all words in T_i as well as the average $avg[\Lambda]$ of all overestimation factors. An overestimation factor threshold (Λ^{thre}) is defined as follows [5].

$$\Lambda^{thre} = \begin{cases} avg[\Lambda] & \text{if } avg[\Lambda] > 1 \\ 1 & \text{otherwise} \end{cases} \quad (2)$$

Any word that has an overestimation factor above the Λ^{thre} is considered to be a content word. Otherwise the word is considered to be a function word and is removed from the vector T_i.

Feature 2: Operations

The `<interface>` element in WSDL 2.0 defines the operations a service can perform. In good development prac-

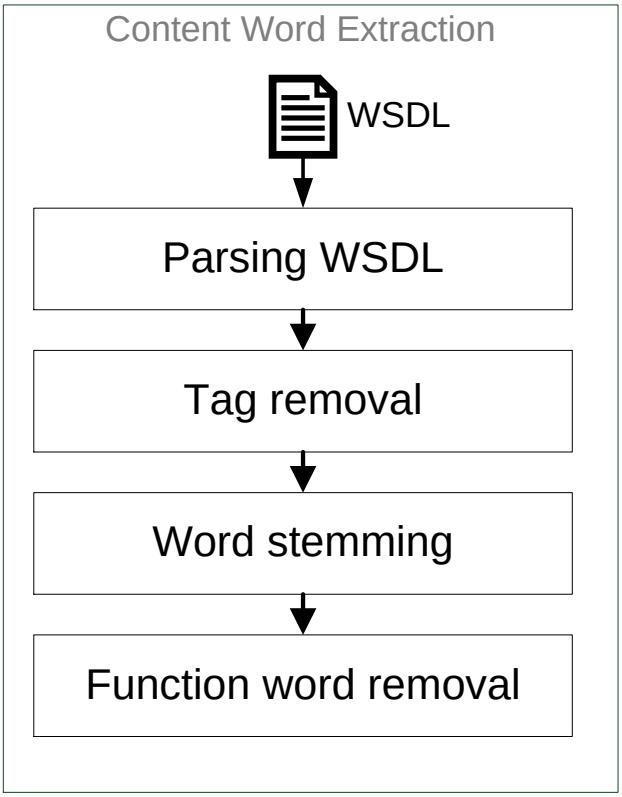

Figure 2: Steps of content word extraction.

tices, the name of the operation typically reflects the semantic meaning of what the operation does. The child `<documentation>` element also tells more about the functionality of the operation. The name of an operation and the keywords describing the functionality of the operation are valuable information that positively contribute to finding similarity between services. We use Eq. (3) to calculate the similarity score between two services based on exposed operations. Eq. (3) is also used for the following three features.

$$match(s_i, s_j) = \frac{M(s_i, s_j)}{avg(E_{s_i}, E_{s_j})} \quad (3)$$

where s_i and s_j are different Web services, $M(s_i, s_j)$ produces the number of matched operations, and E_{s_i} and E_{s_j} are the total numbers of defined operations in both s_i and s_j respectively.Eq. (3) is a generic formula that is used for following three features.

Feature 3: Messages

The `<interface>` element and its child operation element define the operations exposed by the service. For example a *getBookList* operation returns a list of books. Each operation has a *pattern* attribute that specifies the message exchange patterns. For instance, `pattern="http://www.w3.org/ns/wsdl/in-out"` means the operation receives an input message and returns an output message, order preserved. WSDL 2.0 also support several other message exchange patterns, for instance, *In-Only*, *In-Out* and *In-Optional-Out*. The patten type is a good candidate for inferring the functional similarity between two services. Message definitions are typically considered as an abstract definition of the message content. The message may contain multiple attributes (or elements) and the sequence of the elements is strict. In our approach, we match the messages' structure between Web services and use Equation (3) to calculate this match. In this case, $M(s_i, s_j)$ is the number of matched messages between s_i and s_j, and E_{s_i}, E_{s_j} are the total number of defined messages in s_i, and s_j, respectively.

Feature 4: Types (complexType)

WSDL 2.0 supports multiple type systems for describing the message content. The types are used to describe the message structure of a specific operation, so that consumers know what input messages to send and what output messages to expect. The following list shows a sample `<complexType>` definition of a book store Web service.

```
<complexType name="getBookListType">
 <sequence>
  <element name="author" type="string"/>
  <element name="title" type="string"/>
  <element name="publisher" type="string"/>
  <element name="subject" type="string"/>
  <element name="language" type="string"/>
 </sequence>
</complexType>
```

WSDL specifications use XML Schema Definition (XSD) as their canonical type system. Types can be as simple as a single element or as complex as an array of elements. Each element has a name attribute and a type attribute. While the name attribute is sometimes not a useful feature, the type attribute is a good candidate for describing the functionality of a service. Xin et al. [23] show that complex data types are the most informative element in WSDL documents. We therefore extract element types and determine the number of type matches between a pair of Web services using Eq. (3). In this case, $M(s_i, s_j)$ is the number of matched types between s_i, s_j, and E_{s_i} and E_{s_j} are the total numbers of defined types in s_i and s_j, respectively.

Feature 5: Service Name

We consider the service name used in the URI of the WSDL document. For example, the URI of the WeatherForecast Web service is http://www.webservicex.net/WeatherForecast.Asmx?WSDL, and so the name of the Web service is "Weather Forecast". This name could be totally different from the name used inside the WSDL document itself. In case of composite names, such as "WeatherForecast", we split the composite name into multiple names based on the assumption that a capital letter indicates the start of a new word. Using Normalized Google Distance (NGD) [24], we find the similarity between services names as follows:

$$sim(sname_i, sname_j) = 1 - NGD(sname_i, sname_j) \quad (4)$$

where $sname_i$ and $sname_j$ are the names of the Web services s_i and s_j respectively.

TABLE 1: Performance comparison over four manually identified clusters.

Cluster	Our approach		Liu's approach [20]	
	Precision%	Recall%	Precision%	Recall%
Currency exchange	90.0	94.7	84.2	88.9
E-mail verification	80.0	100	58.3	87.5
Weather forecast	94.1	100	70.0	87.5
Car information	90.0	90.0	60.0	90.0

Feature Integration

We use the Quality Threshold (QT) clustering algorithm [25] to cluster similar Web services based on the four similarity features presented above. We decided to use QT because it returns consistent results across multiple runs with the same input, and it can be used to cluster particular groups. The drawback of QT, however, is that it is more computationally intensive than other clustering algorithms [25]. We measure the similarity factor $\Theta(s_i, s_j)$ between Web services s_i and s_j as follows:

$$\begin{aligned}\Theta(s_i, s_j) =& 0.2S(T_i, T_j) + 0.2sim(sname_i, sname_j) + \\ & 0.2match(typ_i, typ_j) + 0.2match(msg_i, msg_j) + \\ & 0.2match(opr_i, opr_j) \end{aligned} \quad (5)$$

$\Theta(s_i, s_j)$ is equal to "1" if the two services are identical and $\Theta(s_i, s_j)$ is equal to "0" if they are completely different. We normalize $\Theta(s_i, s_j)$ by assigning weights of 0.20 to each of the four similarity features. We experimentally determined that these weights give reasonable results. In Equation (5) T_i and T_j are the content word vectors of services s_i, s_j respectively. $S(T_i, T_j)$ is the average similarity between the content word vectors T_i, and T_j and is calculated with

$$S(T_i, T_j) = \frac{\sum_{a \in T_i} \sum_{b \in T_j} sim(a, b)}{|T_i||T_j|} \quad (6)$$

where $sim(a, b)$ is the featureless similarity factor computed between words a and b using NGD based on the word co-existence in Web pages. $sim(a, b)$ is calculated using

$$sim(a, b) = 1 - NGD(a, b) \quad (7)$$

where a and b are content vector words that belong to T_i and T_j respectively.

Experimental results show that our clustering approach is more robust and accurate than the one proposed by Liu and Wongs [20]. Table 1 shows the performance comparison between both approaches in terms of the precision and recall relative to four manually identified groups in our test data set. We could not calculate the global precision and recall for both approaches as it is infeasible to manually verify all clusters in the dataset. We note that the low precision of our approach for the *"E-mail verification"* clusters in Table 1 is due to mutual correlation with some individual services from the *ip-to-country* Web services domain. We also note that all Web services that are supposed to belong to *"E-mail verification"* and *"Weather forecast"* groups are successfully placed in the clusters, as indicated by 100% recall value.

5. Matchmaking Algorithms

The next and most important step now is to build efficient data structures to perform matchmaking between discovery requests and available services. The matchmaking process could be done using similarity measures [24], that are able to capture the semantic relevancy between two terms, or statistical methods that calculate the absolute Euclidean distance [26]. In this paper, we provide four different methods: semantic keyword matching using NGD, K-D tree, brute force matching over TF-IDF matrix, and LSH. Our experiments highlight the precision and performance of each method. NGD can capture the semantic relevancy between two keywords, while the other algorithms cannot. As such, NGD provides the highest accuracy, but at the cost of latency. Therefore, we take NGD as the benchmark. In the following we provide the details of each algorithm and briefly highlight some related background to each method to help understand the concept.

5..1 NGD method

Normalized Google distance (NGD) [24]is a relative semantic distance between a pair of keywords based on the World Wide Web and a search engine that returns aggregate page counts. NGD is a featureless distance measure. For example,*"car"*, and *"vehicle"* would be very close to each other. We use NGD for our clustering approach to measure the functional similarity between two Web services. When a discovery request is received, we extract the keywords and perform all preprocessing as described earlier to generate the content word vector of the request. Then a brute force matchmaking is performed between the request vector and relevant cluster to retrieve relevant services. The NGD process is relatively time-consuming but produces high accuracy results.

5..2 K-D Tree method

The K-D tree discovery algorithm counts on generating the TF-IDF matrix for the service corpus and then build the equivalent K-D tree structure to remain in the memory for runtime discovery.

5..2.1 TF-IDF

Term frequency-inverse document frequency (TF-IDF) is a statistical measure that is used to estimate the importance of a word in a document or in a collection of documents [7]. The words with a high TF-IDF score are often the words that best describe the topic of the document. TF-IDF is commonly used in many information retrieval and text mining applications. Suppose we have a collection of N documents and f_{ij} is the frequency (i.e., number of occurrences) of a term (word) i in a document j. Then, the term frequency TF_{ij} can be defined as [27]:

$$TF_{ij} = \frac{f_{ij}}{max_k f_{kj}}, \quad (8)$$

where k is maximum number of occurrences of any word in the same document j. This maximum value is calculated after performing stop word removal and word stemming for all documents.

The inverse document frequency (IDF_i) describes whether a word i is a common or a rare term across all

the documents in the corpus. IDF_i is calculated using the following equation:

$$IDF_i = \log(\frac{N}{n_i}), \quad (9)$$

where N is the total number of documents in the corpus and n_i is the number of documents in which the word i appears.

The TF-IDF score for a word i in a document j is calculated by:

$$TF\text{-}IDF = TF_{i,j} * IDF_i \quad (10)$$

We use the TF-IDF measure to generate a TF-IDF vector for each service in the corpus. The length of this vector is the total number of unique terms in all WSDL documents and each item in the vector represents the TF-IDF score of the respective term in the document. Then, we generate the TF-IDF matrix of the corpus, where each row j is the TF-IDF vector of a service s_j. For instance, the size of the TF-IDF matrix of our test corpus is 3708×9672, where 3708 is the total number of services and 9672 is the number of unique terms in all WSDL files.

5..2.2 K-D Tree

The TF-IDF matrix is typically a highly dimensional and sparse matrix due to the large number of services and unique words across all services. This matrix must be efficiently indexed to improve the keyword searching process. Towards this purpose, we build the K-D tree for the corpus. K-D (K-Dimensional) tree is a binary tree in which every node is a K-dimensional point [26]. Every non-leaf node represents a splitting hyperplane that divides the space into two equivalent parts based on the hyperplane median value. Points to the left of this hyperplane represent the left sub-tree of that node and points to the right represent the right sub-tree. The hyperplane direction is selected such that every node in the tree is associated with one of the K dimensions, where the hyperplane is perpendicular to the axis of this dimension. Therefore, the data points are uniformly distributed around the hyperplane into two equal-size subsets. For instance, if the "y" axis is selected for a particular split, then all sub-tree points with a smaller "y" value than the split node will appear in the left sub-tree. Similarly, all points with a larger "y" value will appear in the right sub-tree. The hyperplane splitting is based on the median K dimension value of all points. This process ensures that the generated K-D tree is balanced and that the times to search the tree are uniform.

A K-D tree has a set of configuration parameters to tune the search performance. The $leaf\_size$ parameter is used to determine how many data points are stored in a leaf node. The search algorithm navigates the tree and switches to the *brute force* mode when it reaches a leaf node to exhaustively match all data points in this leaf node. The value of this parameter should be carefully set as it may directly impact the performance. Large values of the $leaf\_size$ makes building the K-D tree faster, but negatively impacts the search speed. On the other hand, setting the $leaf\_size$ with a small value may lead to missing nearest neighbour points located on other leaves.

5..2.3 K-D Tree Algorithm

Figure 3 shows an abstract view of our *goDiscovery* approach. Steps 1-4 are performed offline while steps 5-7 occur online. *goDiscovery* starts with the set of description files of available Web services (i.e., WSDL documents in this case). This set of description files makes up the service corpus. The *Content Word Extraction* component takes the service corpus and performs pre-processing to extract the content word vector of each service (i.e., the set of keywords that contribute the most to the meaning of the service functionality). Section 4.1 provides more details on this pre-processing step. Step 1 produces a $n \times m$ keyword matrix, where n is the number of services in the corpus and m is the total number of unique keywords in the whole corpus. Each row j represents the content word vector of service s_j. This matrix goes to the TF-IDF module (step 2) which generates the TF-IDF model and the TF-IDF matrix for the service corpus. The TF-IDF model is a *key-value* pair data structure that contains the total keywords in the service corpus and their associated occurrence number. This model is used in the following steps to generate the TF-IDF vector of each service, forming the TF-IDF matrix of the corpus. The value of each cell represents the TF-IDF score of each respective term. For instance, the value at row j and column i represents the TF-IDF score of the term i in the content word vector of Web service s_j. The TF-IDF matrix then goes to the K-D tree builder (step 4) to build the K-D tree that represents the corpus. This tree is used later to find their nearest neighbours (i.e., Web services) that are relevant to the user query during runtime (step 6). Both the TF-IDF model and the K-D tree can remain in the memory during runtime for efficient handling of user queries.

During runtime, the user submits a query to discover relevant Web service(s). The query goes into the *Content Word Extraction* module, which produces the query content word vector (step 5). Next, the TF-IDF model is used to generate the equivalent TF-IDF vector (step 6). Then, the *goDiscovery* approach traverses the K-D tree with this vector using Algorithm 3 (step 7) to retrieve relevant service(s).

Algorithm 1: build-index($wsdl\_dataset$)

```
for wsdl in wsdl_dataset do
    // bag of words extraction from WSDL files
    wsdl_bow = extract-words(wsdl)
    stop-word-removal(wsdl_bow)
    stem(wsdl_bow)
    wsdl_bow_set.add(wsdl_bow)
end for
// generating the TF-IDF model from the bow
tf_idf = generate-tfidf(wsdl_bow_set)
// building K-D tree using the TF-IDF vectors
kd_tree = build-tree(tf_idf.matrix)
```

The K-D tree discovery algorithm consists of three main components. The first component, Algorithm (1), generates the TF-IDF vectors from Web service description files and builds the K-D tree index for the service corpus. The second component, Algorithm (2), processes the user query and generates its equivalent TF-IDF vector using the TF-IDF model generated by Algorithm (1). The third component,

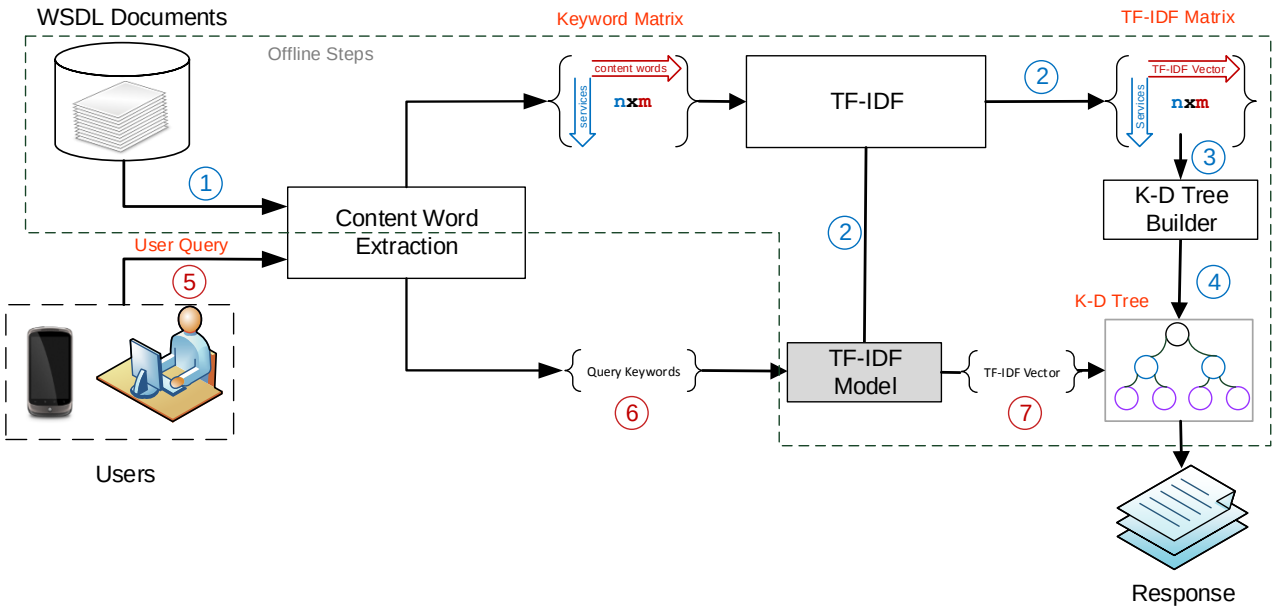

Figure 3: An abstract view of of the K-D tree based discovery approach.

Algorithm 2: submit-query($query$, tf_idf, kd_tree)

stop-word-removal($query$)
stem($query$)
/* transforming the query string to its equivalent TF-IDF vector */
$query\_tfidf\_vector$ = $tfidf.model$.transform($query$)
/* calling the find_nn() algorithm to navigate the K-D tree starting from the root ($depth = 0$) */
$result$ = find_nn(0,kd_tree,$query\_tfidf\_vec$,k)
return $result$

Algorithm 3: find_nn($depth$,kd_tree,$query\_tfidf\_vec$,k)

if kd_tree is leaf node **then**
  /* retrieving all TF-IDF vectors of the nearest neighbour services that are stored in the leaf node */
  $nn\_services$ = $getNNServices(kd\_tree)$
  // sorting candidate services based on distance
  $sort(nn\_services, euclidean\_distance)$
  // returning the results
  $result$ = $nn\_services.sublist(k)$
  return $result$
end if
$splitting\_value$ = $kd\_tree.root.value$
$query\_median\_value$ = $query\_tfidf\_vec[depth]$
/* splitting the tree based on the query median value of the current dimension and the root splitting value */
if $query\_median\_value \leq splitting\_value$ **then**
  $sub\_tree = root.left$
else
  $sub\_tree = root.right$
end if
// recursive calling to find_nn() on the sub-tree
return $find\_nn(depth + 1, sub\_tree, query\_tfidf\_vec, k)$

Algorithm (3), navigates the K-D tree to discover matching services to the user query at runtime.

Algorithm (3) finds the k nearest neighbour services. It uses the values in the query TF-IDF vector to traverse the K-D tree. Each node in the tree splits on a particular hyper-plane dimension. Then, recursively the algorithm compares the value for that dimension in the query vector and takes the appropriate subtree. Once reaching a leaf node, the algorithm retrieves the nearest neighbour services that are stored in this leaf node.

Then, the find the nearest neighbour algorithm sorts the retrieved vectors in ascending order based on the Euclidean distance. The Euclidean distance d between two points p and q in the Euclidean space is calculated by:

$$\sqrt{\sum_{i=1}^{d}(p_i - q_i)^2} \tag{11}$$

After sorting, it returns the first k nearest neighbour services.

The complexity of the K-D tree search is $O(log_2(n))$, where n is the total number services in the corpus. The sorting complexity is a constant and can be safely ignored as the number of TF-IDF vectors that are stored in the leaf node is a constant.

Searching the K-D tree starts from the root node down to the bottom and checks at successive each level whether the query vector falls to the left or to the right of the current node. Hence, to find all the exact matches, the search may end up testing almost all the nodes in the dataset and the

complexity reaches $O(N)$ [28]. In addition, building the K-D tree requires calculating all the median statistics over all dimensions which is computationally intensive and expensive if the working dataset is highly dimensional. Therefore, in such cases we propose using LSH.

5..3 Locality Sensitive Hashing

Hash tables are good for high dimensional data. An ideal hash table provides $O(1)$ for a lookup operation using $O(N)$ memory space. However, inserting data points into a hash table requires a well-designed hash function to allocate close values into the same bucket. Despite hash tables are typically used for exact matching, Datar et al. [29] propose *Locality Sensitive Hashing (LSH)*, which an approach that can efficiently find approximate matches. The key idea of LSH is to hash input data points using multiple hashing functions such that for each function, the probability of collision is much higher for close points than those far apart. Thus, we can determine the nearest neighbours of a given query by retrieving collocated data points sharing the same bucket with the query.

The basis of LSH is to perform random projections for the query point to generate the corresponding hashing value. For instance, $h(q) = q.x$ where q is the query vector in a high dimensional space and x is a random vector such that each value of x is selected randomly from a Gaussian distribution (e.g., $N(0, 1)$). The hashing value is the result of scalar projection of the query vector over the random vector.

A hashing family F is called sensitive for any two points p and q in a high dimensional space R^d if:

- $||p - q|| \leq R$ then $P[h(q) = h(p)] \geq P_1$ and
- $||p - q|| \geq c * R$ then $P[h(q) = h(p)] \leq P_2$
- $P_1 >> P_2$

where c is a constant and R is the radius distance between q and its nearest neighbour points as Figure 4 illustrates.

5..3.1 LSH Discovery Algorithm

The first step is to construct the hash tables through scalar projection of the TF-IDF vectors as shown in Algorithm 4. With the hash size (the size of the hash key), number of hash tables, and the total number of dimensions, we construct a random vector for each table such that each value in this random vector is selected from a Gaussian distribution with mean 0 and variance 1 (Algorithm 4). Secondly, for each table we calculate the dot product between this random vector and each data point p (i.e., TF-IDF vector) to get the corresponding bucket. Building the LSH index requires $O(nL)$, where n is the number of documents in the corpus and L is the number of the hash tables (Algorithm 5). The LSH indexing is entirely performed offline. We note that the LSH design supports assigning and storing new data points to existing buckets without re-constructing the hash tables. When a query is received, the system generates its equivalent TF-IDF vector and iterates over the hash tables to calculate the scalar projection of the query data point. This step returns a list of candidate nearest neighbours to the query point (i.e., those belong to query bucket). Then, based on a distance function (e.g., Hamming distance), the system returns the top k nearest neighbour points (Algorithm 6).

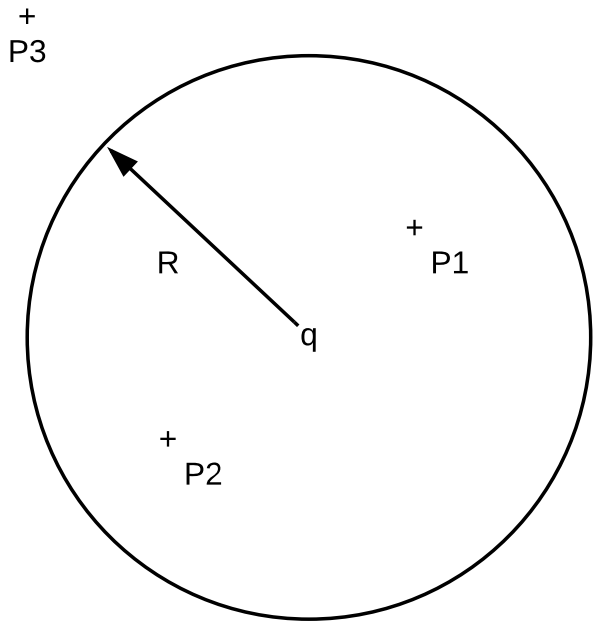

Figure 4: Query point and its nearest neighbours in R^2.

Alternatively, we can select the candidate points within a predefined R distance.

Algorithm 4: LSH.init($hashSize, L, D$)

```
for l from 0 to L do
    // Create L hash tables and L random vectors
    hashTables.add(new HashTable())
    v = new RandomVector(hashSize , D)
    RandomVector.init(v)
    randomVectors.add(v)
end for
RandomVector.init(v)
// Define a random generator variable
random = new Random()
vectorValues = new double [hashSize][D]
// Init. a random plane from a Gaussian
distribution
for row from 0 to hashSize do
    for column from 0 to D do
        values[row][column] = random.nextGaussian()
    end for
end for
v.values = values
```

6. Performance Evaluation

In this section, we study the performance of the proposed Web service discovery approach. We run a number of experiments to measure a set of performance attributes and compare the results with the state-of-the-art statistical-based Web service discovery mechanisms [6].

Algorithm 5: LSH.indexPoint($point$)

```
for htable in hashTables do
  for rv in randomVectors do
    // Perform dot product and return the binary
    representation of the hash value
    bin = hash(rv,point)
    htable.append(bin,point)
  end for
end for
```

Algorithm 6: LSH.query($query$,k)

```
candidates = new List()
for tableIndex from 0 to hashTables.size() do
  bin = hash(randomVectors[tableIndex],query)
  currentHashTable = hashTables.get(tableIndex)
  candidates.add(currentHashTable.get(bin))
end for// Sorting the candidates based on the
distance function
sortedCandidates = sort(candidates)
// Return top k candidates
return sortedCandidates.take(k)
```

6..1 Experimental Setup

We experiment over the dataset available from the Web service benchmark listed in [30], which contains 3716 WSDL files of different Web service categories. We run our experiments on a the subset of Web services whose WSDL files have a valid SOAP message structure, so that we can extract information about their operations, input/output messages and functionality descriptions. This subset includes 3708 WSDL files in total after ruling out invalid services. We perform manual classification of these WSDL files to establish ground truth for the service retrieval. We identified four categories: "Car Information" , "Currency Exchange", "Email Validation" and "Weather Forecast" to restrict the scope of the evaluation. We generate a vector of words from this information that represents the textual information of each WSDL document. Then, we perform stop-word removal and stemming using a Porter stemmer [21] on the generated vector to keep only the unique keywords that distinguish a service's functionality. This results in a matrix of content keywords (see Figure 3), where each row is the content word vector d_j of a service s_j in the corpus, where $1 \leq i \leq 3708$. The length of d_j is ideally the size of the total corpus vocabulary (for our dataset, this size is 9672). Then, we generate the equivalent TF-IDF matrix from the keyword matrix using scikit-learn [31], which is an open source Python library for data mining and analysis. The size of the generated TF-IDF matrix is (3708×9672). This matrix is used to build the K-D tree and generate the LSH index of our corpus. We ported an open source project called *LSHash* [32] to implement our own version of LSH. We iterate over varying hash sizes ranging from 2 to 10 with different numbers of hash tables ranging from 1 to 50 to determine the appropriate experimental settings. We observe that a hash size 4 with 20 hash tables provides a better F-Score and reasonable response time of LSH.

When a discovery request is received, the system generates the TF-IDF vector of the request and performs matchmaking. We evaluate four matchmaking algorithms: semantic keyword matching using NGD (refereed as *NGD*), brute force search over the TI-IDF matrix [6] (short as *"Brute Force"*), *K-D Tree*, and *LSH*. We evaluate the performance of the four algorithms in terms of accuracy, response time, and scalability.

6..2 Accuracy Analysis

To study the accuracy, we use a set of common performance metrics: *Precision*, *Recall*, *F-score*, and *Top-k Precision*. To formally define these parameters, we assume that Rel is the set of actually relevant Web services in the corpus for a specific discovery request (i.e., all true positive), Ret_t is the set of Web services retrieved by the search engine, Ret_r (true positive) is the subset of relevant services in the retrieved set, where $Ret_r \subset Ret_t$, and Ret_k is the subset of relevant services in the top k retrieved services.

Precision (P): Precision is a measure of how many retrieved services are relevant and is defined as the following:

$$P = \frac{|Ret_r|}{|Ret_t|} \tag{12}$$

Recall (R) Recall is a measure of how many relevant services are retrieved and is defined as the following:

$$R = \frac{|Ret_r|}{|Rel|} \tag{13}$$

F-score: It is a comprehensive measure of the overall system accuracy. The F-score can be viewed as a weighted average of the *Precision* P and *Recall* R. The F-score is a percentage value defined as follows:

$$\text{F-score} = 2 * \frac{P * R}{P + R} \tag{14}$$

The F-score is a harmony mean between precision and recall and its value represents a unified view of system performance. Therefore, the accuracy of the system can be represented with a single measure instead of multiple. In addition, F-Score helps to determine the best threshold of how many services should be retrieved per a specific query. We use F-Score to make our system adaptive and choose the appropriate threshold based on the maximum F-score value. Also, we use F-Score to evaluate different approaches.

Top-k Precision (P_k): is a measure of how many relevant services are in the top k retrieved services and is defined as:

$$P_k = \frac{|Ret_k|}{k} \tag{15}$$

We run four queries that target the four identified Web service categories. Figure 5 shows the F-Score of the four matchmaking algorithms for each query. According to recall, we can conclude that K-D tree retrieves at least 96% of relevant Web services in different categories, while LSH achieves $\sim 81.3\%$ precision. However, we observe that the "Email Validation" category has a relatively low precision ($\sim 81\%$). We attribute this to the diverse usage of the term *"valid email"*. Some services may contain variations

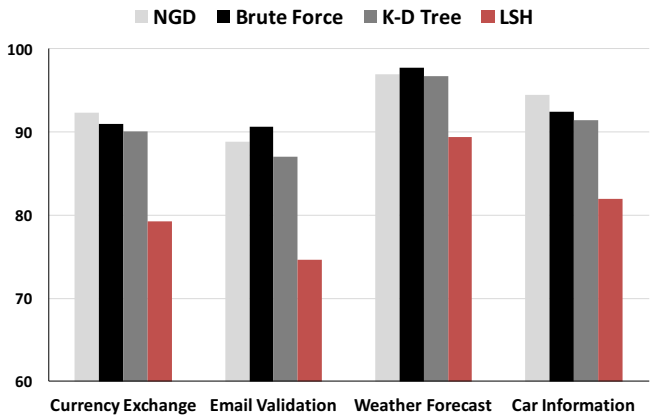

Figure 5: F-Score of the four identified categories.

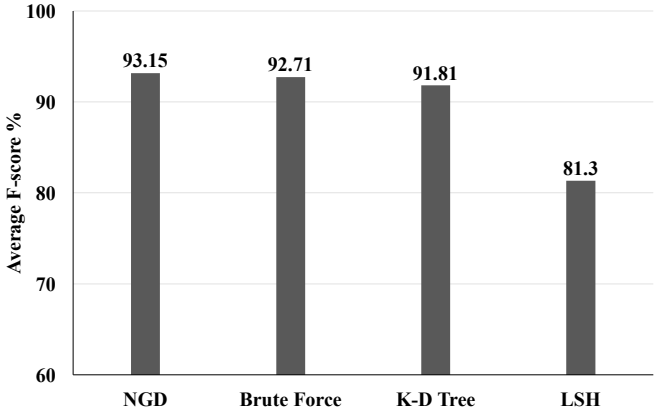

Figure 6: F-Score comparison between the four matchmaking algorithms-average of 4 queries.

of the phrase "valid email" in the documentation section, however, the Web service itself does not offer email validation service. For example, the *United-States-Account* WSDL file contains *"This should be a valid email address"* in the documentation of one of its input message parameters. A manual investigation of this service reveals that the goal of the service is to provide methods that allow users to open and manage accounts. The service requires a valid email for the user to register. Such services may mislead our system as it is considered a good candidate to retrieve. This is a limitation of the TF-IDF as it depends on a bag of words representation which does not consider the weight of each word in its context (i.e., the position of the word in the document). Current TF-IDF implementations consider that all words carry the same weight in the document. In general, both *NGD* and *Brute Force* search perform better than *K-D* and *LSH* in terms of precision but at the cost of response time.

Figure 6 shows the average F-score for the four algorithms under investigation. From these results, we find that the *NGD* algorithm outperforms the other three approaches in terms of accuracy. This is because it adopts a semantic-based search algorithm, while the other approaches depend on the absolute vocabularies that exist in the corpus. Also, we observe that the *Brute Force* approach is slightly better than both *K-D Tree* and *LSH* as it exhaustively calculates the distances between the TF-IDF vector of the query and all TF-IDF vectors in the corpus, so it ensures retrieving the exact nearest neighbours. K-D tree and LSH may miss some candidates if they are located in a different leaf node. However, we observe that the differences are negligible and the four approaches all perform similarly in terms of the overall accuracy. On another note, LSH takes more space and more time to build than K-D Tree. LSH provides lower accuracy but with very low latency.

6..3 Response Time Analysis

We also conducted experiments to measure the end-to-end query response time using the four algorithms. In these experiments, we use an Intel-based machine with a single core of 2.20 GHz - Core i5 and 6 GB of memory. Figure 7 presents the average query response time of the four algorithms over

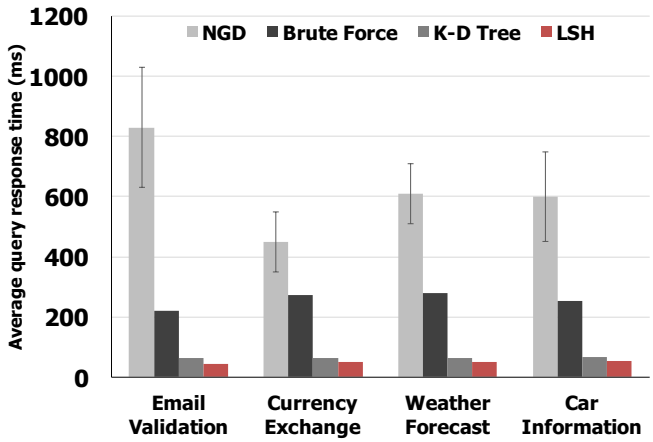

Figure 7: Average response time.

four different queries. We observe that LSH significantly outperforms the other three algorithms and presents a dramatic improvement to the overall query response time with K-D tree coming in second place. LSH improves the query response time by 91.5% on average compared to NGD and 80.6% on average compared to Brute Force. LSH improves the average response time compared to K-D tree by 22.2%. While K-D Tree brings a total reduction in response time of 92% compared to NGD and 77% compared to the Brute Force.

In addition to the runtime performance study, we carried out experiments to evaluate the offline overhead of building the TF-IDF model, K-D tree, and LSH tables with a varying size of the input dataset. It is worth noting that this is a one-time overhead. Figure 8 shows the average processing times (in msec) of generating the TF-IDF model and K-D tree with dataset on a dataset ranging from 25% (927 WSDLs) to 400% (14832 WSDLs). These results are based on the average of 20 runs.

6..4 Scalability Analysis

We assume that the ability to accommodate more simultaneous users is a good projection of the system scalability. We

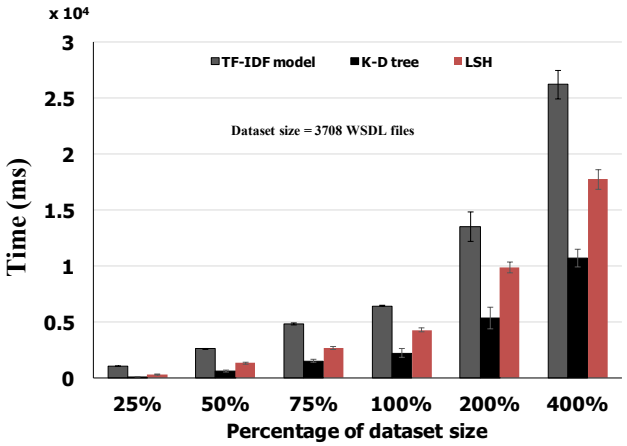

Figure 8: Offline average processing time.

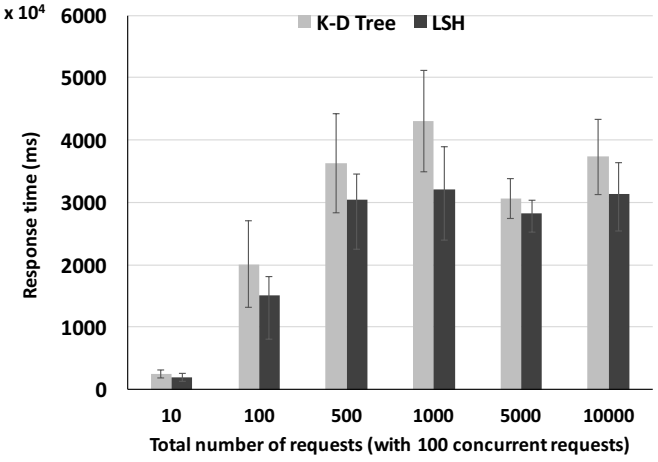

Figure 9: Concurrent discovery analysis.

conducted a load test to study the behavior of the system under a specific expected load. We perform this test on K-D tree and LSH only since they outperform other mechanisms, while returning comparable precision. We configure our experiment with various settings to capture a variety of concurrent discovery scenarios.

We use Flask [33] to expose the discovery algorithms as RESTful Web services. Both discovery services receive a discovery query and search the corpus for relevant services. We use Apache Benchmark [34] to apply load testing. We set the reference capacity of the host server to 100 concurrent requests. We use different settings for load testing as follows:

- Total 10 requests with 10 concurrent requests.
- Total 100 requests with 100 concurrent requests.
- Total 500 requests with 100 concurrent requests.
- Total 1000 requests with 100 concurrent requests.
- Total 5000 requests with 100 concurrent requests.
- Total 10000 requests with 100 concurrent requests.

We exercised each scenario 10 times and report the average readings. Figure 9 shows the average response time under varying load profiles. We observe that both LSH and K-D tree scale well with increasing simultaneous discovery requests. However, LSH returns results with 18% faster response time on average than K-D tree. We note that load balancing techniques on multiple running machines serving service discovery can further improve the system scalability.

7. Conclusion

We present a comprehensive Web service discovery approach the better fit the requirements of large scale IoT deployments. Our main focus is to provide near real time service discovery so that IoT applications meet their real time constraints. To achieve this goal, we leverage statistical measures and indexing techniques to support fast and efficient service discovery. Our approach extracts unique keywords from service description documents, generates a TF-IDF model, and builds a K-D tree and LSH tables for the service corpus offline and uses them to match user requests at runtime. The system receives the user query and generates its equivalent TF-IDF vector. We study the use of four different matchmaking algorithms within the common framework. Our performance evaluation shows that LSH provides the lowest response time with a slightly lower accuracy. K-D tree is the second best matching algorithm in terms of response time with a better accuracy than LSH. NGD outperforms all algorithms in terms of precision due to its semantic matching capability, but at a high latency cost. Brute force searching over TF-IDF matrix trades off between precision and response time. In future, we plan to test our proposed system on multiple machines and study horizontal scalability. We also plan to compare between service discovery in cloud and cloutlet settings, where discovery requests are resolved on the network edge close to users/applications submitting the requests .

References

[1] Microservices. [Online]. http://martinfowler.com/articles/microservices.html. [Accessed: November 30, 2015].

[2] Microservices. [Online]. http://www.gartner.com/newsroom/id/2905717. [Accessed: November 30, 2015].

[3] W. Wang, S. De, G. Cassar, and K. Moessner, "An experimental study on geospatial indexing for sensor service discovery," *Expert Systems with Applications*, vol. 42, no. 7, pp. 3528–3538, 2015.

[4] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, Jun. 1984.

[5] K. Elgazzar, A. E. Hassan, and P. Martin, "Clustering WSDL Documents to Bootstrap the Discovery of Web Services," in *Proceedings of the 2010 IEEE International Conference on Web Services*, 2010, pp. 147–154.

[6] C. Platzer and S. Dustdar, "A vector space search engine for web services," in *Proceedings of the Third IEEE European Conference on Web Services*, 2005, pp. 1–9.

[7] G. Salton and M. J. McGill, "Introduction to modern information retrieval," 1983.

[8] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," June 26 2007, http://www.w3.org/TR/wsdl20. [Accessed: Feb. 10, 2012].
[9] D. Martin, M. Burstein, D. Mcdermott, S. Mcilraith, M. Paolucci, K. Sycara, D. L. Mcguinness, E. Sirin, and N. Srinivasan, "Bringing semantics to web services with owl-s," in *World Wide Web*, vol. 10, 2007, pp. 243–277.
[10] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fense, "Web service modeling ontology," *Applied Ontology*, vol. 1, pp. 77–106, November 2005.
[11] SOAP Messaging Framework (Second Edition). [Online]. http://www.w3.org/TR/soap12/. [Accessed: Feb 1, 2015].
[12] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, USA, 2000.
[13] M. Klusch, B. Fries, and K. Sycara, "Automated semantic web service discovery with owls-mx," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, 2006, pp. 915–922.
[14] J. Garofalakis, Y. Panagis, E. Sakkopoulos, and A. Tsakalidis, "Contemporary web service discovery mechanisms," *J. Web Eng.*, vol. 5, no. 3, pp. 265–290, 2006.
[15] N. Kokash, "A comparison of web service interface similarity measures," *Frontiers in Artificial Intelligence and Applications*, vol. 142, p. 220, 2006.
[16] Y. Hao, Y. Zhang, and J. Cao, "Web services discovery and rank: An information retrieval approach," *Future Generation Computer Systems*, vol. 26, no. 8, pp. 1053 – 1062, 2010.
[17] P. N. Mendes, M. Jakob, A. García-Silva, and C. Bizer, "Dbpedia spotlight: Shedding light on the web of documents," in *Proceedings of the 7th International Conference on Semantic Systems*, New York, NY, USA, 2011, pp. 1–8.
[18] R. Nayak, "Data mining in web services discovery and monitoring," in *International Journal of Web Services Research*, vol. 5, 2008, pp. 63–81.
[19] Y. Elshater, K. Elgazzar, and P. Martin, "goDiscovery: Web Service Discovery Made Efficient," in *Web Services (ICWS), 2015 IEEE International Conference on*. IEEE, 2015, pp. 711–716.
[20] W. Liu and W. Wong, "Web service clustering using text mining techniques," *International Journal of Agent-Oriented Software Engineering*, vol. 3, no. 1, pp. 6–26, 2009.
[21] M. F. Porter, "An algorithm for suffix stripping," *Program: electronic library and information systems*, vol. 14, no. 3, pp. 130–137, 1980.
[22] S. M. Katz, "Distribution of content words and phrases in text and language modelling," *Natural Language Engineering*, vol. 2, no. 01, pp. 15–59, 1996.
[23] X. D. Alon, X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang, "Similarity search for web services," in *Proceedings of VLDB'04*, 2004, pp. 372–383.
[24] R. L. Cilibrasi and P. M. B. Vitanyi, "The google similarity distance," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 3, pp. 370–383, 2007.
[25] S. Y. Laurie J. Heyer, Semyon Kruglyak, "Exploring expression data: Identification and analysis of coexpressed genes," *Genome Research*, vol. 9, no. 11, pp. 1106–1115, 1999.
[26] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
[27] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2011.
[28] M. Slaney and M. Casey, "Locality-sensitive hashing for finding nearest neighbors [lecture notes]," *Signal Processing Magazine, IEEE*, vol. 25, no. 2, pp. 128–131, 2008.
[29] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*. ACM, 2004, pp. 253–262.
[30] Y. Zhang, Z. Zheng, and M. R. Lyu, "WSExpress: A qos-aware search engine for web services," in *Proceedings of the IEEE International Conference on Web Services (ICWS)*. IEEE, 2010, pp. 91–98.
[31] "scikit-learn - machine learning in python," http://scikit-learn.org/.
[32] LSHash. [Online]. https://pypi.python.org/pypi/lshash/0.0.4dev. [Accessed: October 1, 2015].
[33] Flask. [Online]. http://flask.pocoo.org/. [Accessed: October 1, 2015].
[34] Apache Benchmark. [Online]. https://httpd.apache.org/docs/2.2/programs/ab.html. [Accessed: November 1, 2015].

AUTHORS

Yehia Elshater is a PhD student at Database Systems Lab, School of Computing - Queen's University. He obtained his Masters from faculty of computers and information - Cairo University in Service Computing. Prior to beginning the PhD program, he was a teaching assistant at Faculty of Computers and Information - Cairo University. He also was a research associate at Qatar Computing Research Institute (QCRI) - Distributed Systems Lab. His current research is focused on big data analytics optimization.

Dr. Khalid Elgazzar is a postdoctoral research fellow at Carnegie Mellon School of Computer Science and an adjunct Assistant Professor in the School of Computing at Queens University, Canada. Dr. Elgazzar received his PhD from Queen's University in 2013. He also received the 2014 Queen's School of Computing Distinguished Research Award. His research interests span the areas of distributed systems, mobile cloud and ubiquitous computing, context-aware cyber-physical systems, and elastic networking paradigms. Dr. Elgazzar has received several recognitions and best paper awards at top international venues. He is a member of the Google IoT Expedition team, leading the creation of an open stack for IoT.

Dr. Patrick Martin is a Professor in the School of Computing at Queens University and the Director of the Database Systems Laboratory. He is a Faculty Fellow and Visiting Scientist with IBMs Centre for Advanced Studies, a Scotiabank Scholar, a member of the Southern Ontario Smart Computing Innovation Platform (SOSCIP) Scientific Committee and a member of the Advisory Panel on Analytics for the Ontario Brain Institute. Dr. Martin has supervised over 90 graduate students at Queens and is the author of over 100 peer-reviewed journal and conference papers. His research interests include big data analytics, database system performance, cloud computing and autonomic computing systems.