Web Service Indexing for Efficient Retrieval and Composition

Marco Aiello, Christian Platzer, Florian Rosenberg, Huy Tran, Martin Vasko, Schahram Dustdar
Distributed Systems Group, TU Vienna – Argentinierstraße 8/184-1, A-1040 Vienna, Austria
{christian,florian,htran,vasko,aiellom,dustdar}@infosys.tuwien.ac.at

Abstract

The success and widespread use of Web service technology is bound to the availability of tools for rapidly discovering services and for composing their operations. We present a system for service discovery and composition based on syntactic and semantic matching of the published service interfaces. The system relies on accurate preprocessing of the available services, that is, on the indexing time of the services' descriptions. The composition task is performed by index lookup and an efficient heuristic for graph traversing. The presented system is the VitaLab entry for the Syntactic and Semantic WS-Challenge 2006.

1. Introduction

One of the most intriguing visions brought by the Web service technologies is that of dynamic discovery from a large set of services and invoking their operations to achieve complex tasks. To achieve this vision, there are two fundamental blocks that need efficient and precise tools: service discovery and service composition. The two components are in strict relation to one another. Typically, to satisfy a query of some kind, one needs to first discover the services that satisfies the request and then perform the sequence of operations that will arrive to the goal of the requester.

As usual in the context of querying large dynamic data sets, there is a delicate balance between how much preprocessing of the data is performed and how much is executed once the actual query is known. Preprocessing implies the generation of indexes for the data. On the positive side of this approach, once the indexes are available, it is rapid to retrieve the objects to which the index refer and in turn to respond to the query. On the negative side, the indexing process is computationally expensive, it requires additional space to store the indexes and, if the data changes often, the indexes need constant update. The opposite approach is to do everything at run-time, which has the advantage of avoiding indexing and its related costs, furthermore, it deals with always up-to-date information, but, as the amount of data to search increases, it becomes inefficient.

Considering the vision of an Internet populated by millions of (loosely coupled) services, we consider a solution to the service discovery and composition problem which heavily relies on rich indexes. The indexes are easily identifiable in the operation names published in the WSDL service descriptions. In addition, considering the possibility of having different names for the operations which are related by semantic relations, such as 'is-a', 'instance of', 'part of' and so on, we consider a vector space based on the semantic representation of the operations. The latter approach has proved to be effective in the context of NLP, e.g., [2].

Next we present the VitaLab system which performs the indexing of a large collection of WSDL service description following a semantic description of operations (given as a tree of 'is-a' relations) and, given a request for a service, composes the available services to satisfy the request. The system is the VitaLab entry for the WS-Challenge 2006 which follows the one held in 2005 [1].

2. The VitaLab System

The system for searching and composing Web services we propose is made of two main elements: one responsible for indexing the available service descriptions, and one responsible for retrieving and possibly composing services to satisfy queries from a requester. Referring to Figure 1, on the left we have the 'indexing' component which is responsible for accessing the WSDL repository and the ontological information in the form of XML schemas. By accessing such information it builds reverse indexes of all available services. These indexes are stored both in primary memory and in secondary memory. The choice of keeping the index in non-volatile memory is to improve performance in case that the successive queries are executed in different runs on the same set of services.

The indexes are then used by the 'composition component', on the right side of Figure 1, for satisfying the queries of requesters. Queries come to the composer, which accesses the indexes and decides whether there is a service which can directly satisfy the request, whether there is a

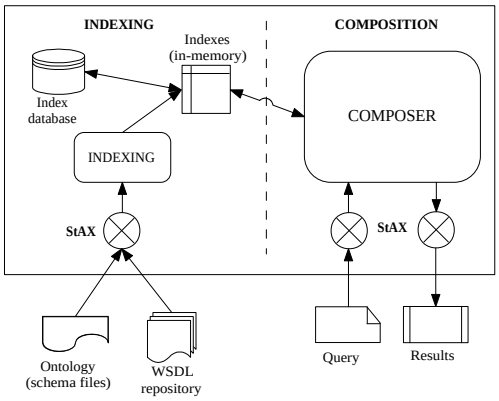

Figure 1. Overview of the VitaLab System.

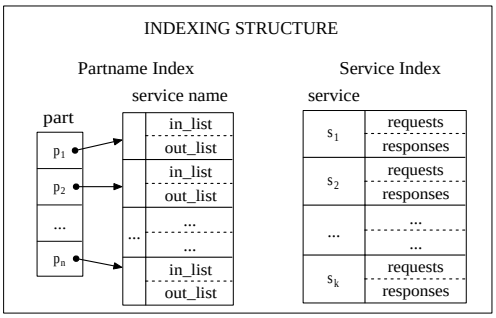

Figure 2. Index Structure.

service semantically equivalent, or whether it is necessary to compose services to satisfy the request. Next we briefly summarize the functioning of the two main elements of the VitaLab system and spend some words on the technology used to efficiently parse WSDL documents.

2.1. Efficient XML parsing with StAX

Parsing a large amount of WSDL files can be efficiently handled by using the Streaming API for XML (StAX)[1]. StAX provides an iterator based API and an underlying stream of XML events. Our parser consumes only the events which deliver information relevant for building a rich index (such as `message` and `part` information as well as the `portType`). Parsing the WSDL files and building the index is time consuming, thus, the parser uses index serialization and stable storage for enhancing the performance on subsequent queries to the system on the same repository. The data structure for indexing is described next.

2.2. Indexing

As an outcome of the parsing process, the indexer is fed with all gathered information extracted from the service repository. A cunningly designed index structure enables fast query processing and insertion of new data [3]. Our main index is implemented with hashtables because of the fast insertion and access to the stored values.

The index structure consists of two tables: Partname Index and Service Index (Figure 2). Partname Index uses a hashtable to maintain the mapping from each partname into two lists of service names, namely, *in_list* and *out_list*. The *in_list* (*out_list*) corresponding to partname p is the list of services which have p as a request (response), respectively.

[1] `http://jcp.org/en/jsr/detail?id=173`

In this way, it takes $O(1)$ to get a list of services that consume, or produce a particular partname p. The Service Index utilizes a hashtable that maps a service name into detail information of the correspondent service. The information is request partnames, and response partnames.

To take advantage of the fact that subsequent queries made to the same repository could occur during different runs of the system, we serialize the index structure as a binary file after the first indexing and store it in non-volatile memory. In a following run of the system, the index is reloaded into memory rather than recreated. In case the repository has changed to an addition/removal or modification of a service it is only necessary to updated the index structure.

2.3. Indexing with semantic information

In the indexing process one could consider not only the actual partnames found in the service descriptions, but also partnames that are semantically related to the ones found. These semantic relations are expressible using a number of different formalisms such as OWL-S. In the present approach, we consider the case of having a type hierarchy defined in XML Schema. This type hierarchy provides all types for input and output parameters where we interpret the father-son relation in the tree as a subsumption relation.

When indexing the WSDL repository, we match each partname on the semantic schemas and build two additional lists for each entry. These lists represent the services in the repository which provide partnames which are more or less general of the one found: *in_list* and *out_list*, respectively. As for the syntactic indexes, given a certain part type, we get all services that can consume (or produce) a type in constant time $O(1)$ by looking up in the Partname Index's hashtable.

2.4. Service composition algorithm

The service composition algorithm relies on the syntactic and semantic indexes to find a sequence of services satisfy-

ing a query q. More formally, a *service repository* is a tuple $\langle S, P, in, out \rangle$, where S is a set of services, P a set of part names and in, out are functions $S \rightarrow Powerset(P)$ that return the set of input and output part names for a service.

The *satisfaction of a service query* q is a service sequence $s_0 \ldots s_n \in S$ such that $in(q) \supseteq in(s_0)$, $in(s_k) \subseteq \bigcup_{i=0}^{k-1} out(s_i)$, and $out(q) \subseteq \bigcup_{i=0}^{n} out(s_i)$. If $n = 1$ we talk about service discovery, otherwise we talk about service composition. If we substitute for in and out in the above definition of query satisfaction respectively $in_{sem}(s) = \{p \in P \mid p \in in(s)$ or p is the child of a part name in $in(s)$ in at least one semantic tree$\}$, $out_{sem}(s) = \{p \in P \mid p \in out(s)$ or p is the parent of a part name in $out(s)$ in at least one semantic tree$\}$, we talk of *semantic discovery and composition*.

The VitaLab algorithm for query q satisfaction works according to the following procedure.

Algorithm 1 Discovery & Composition Algorithm

for all s_0 such that $in(q) \supseteq in(s_0)$ **do**
  $Solution \leftarrow \{s_0\}$
  $current \leftarrow 0$
  repeat
    $level \leftarrow$ set of $s_{candidate}$ such that
      $s_{candidate} \in S \setminus Solution$ and
      $in(s_{candidate}) \subseteq \bigcup_{i=0}^{current} out(s_i)$
    order $level$ based on $max(|out(s \in level)|)$
    for all $s_l \in level$ **do**
      add s_l to $Solution$
      $current \leftarrow l$
      if $out(q) \subseteq \bigcup_{i=0}^{current} out(s_i)$ **then**
        return path found
      end if
    end for
  until $level = \emptyset$
end for

The service composition algorithm just presented handles both the semantic and the syntactic composition of services depending on whether the functions in_{sem} in place of in, and out_{sem} in place of out are used, respectively.

3. Preliminary Evaluation Results

We present some preliminary evaluation results for the syntactic discovery and composition. The experiments were carried out on a Pentium 4, 3 GHz with 1 GB RAM running Fedora Core 4 and Eclipse. In Table 3 the query results for the syntactic discovery and composition for different WSDL repositories as provided by the WS-Challenge organizers are shown. The repositories are organized as follows: discovery-100-32 denotes that each WSDL file contains 32 (up to 36) input and output parameters and 100 denotes the number of services involved per query.

For the discovery and the composition, the times values indicate the duration needed to fulfill 11 discovery/composition requests in a sequence (comprised in one composition/discovery routine).

Dataset	#WSDL	Duration
discovery-100-32	3101	12 msec
discovery-100-4	3101	15 msec
discovery-50-32	2601	11 msec
discovery-50-4	2601	10 msec
composition1-100-32	4156	2313 msec
composition1-100-16	4156	562 msec
composition1-50-32	2656	1809 msec
composition1-50-16	2656	423 msec

Table 1. Syntactic Discovery and Composition Results

Note that these numbers are solely for the query execution itself. Additional results for building the initial index, as well as for indexing and querying the semantic repositories are omitted for space reasons.

4. Concluding Remarks

The VitaLab system is able to search and compose Web services to satisfy queries from a requester. The system builds a rich index structure for efficiently replying to queries at the expenses of having slow bootstrapping performances when new sets of services need to be indexed. Indexes are built considering both syntactic and semantic information available in the form of trees of 'is-a' relations. Composition is achieved resorting to well-known graph search heuristics. The system is the VitaLab entry for the Syntactic and Semantic WS-Challenge 2006.

References

[1] M. Blake, K. Tsui, and A. Wombacher. The eee-05 challenge: A new web service discovery and composition competition. In *2005 IEEE International Conference on e-Technology, e-Commerce, and e-Services (EEE 2005)*, pages 780–781. IEEE Computer Society, 2005.

[2] G. Gonzalo, F. Verdejo, I. Chugur, and J. Cigarran. Indexing with wordnet synsets can improve text retrieval. In *Proceedings of the COLING/ACL'98 Workshop on Usage of WordNet for NLP*, 1998.

[3] C. Platzer and S. Dustdar. A vector space search engine for web services. In *IEEE European Conference on Web services (ECOWS)*. IEEE Computer Society, 2005.