Large Scale Web Service Discovery and Composition using High Performance In-Memory Indexing

Lukasz Juszczyk, Anton Michlmayr, Christian Platzer,
Florian Rosenberg, Alexander Urbanec, Schahram Dustdar
VitaLab, Distributed Systems Group
Vienna University of Technology
Argentinierstraße 8/184-1, A-1040 Vienna, Austria
lastname@infosys.tuwien.ac.at

Abstract

With the growing number and ubiquitous usage of Web services throughout the service-oriented community, the need to find service descriptions in a given repository, as well as composing them to a desired output, becomes a major issue in both research and corporate environments. Considering emerging semantic technologies and methods for service matching that is not limited to a mere syntactic level, the need for fast discovery and composition algorithms arises. In this paper we present a system created at the VitaLab in Vienna with the purpose to overcome the obstacles which are implications of both, large service repositories and large ontologies to describe semantic relations.

1. Introduction

Considering the current research in the field of service-oriented computing, two of the most important topics are service discovery and composition. Especially when dealing with large sets of service descriptions, discovery issues move from trivia to intriguing problems. Finding matches for a given query, or even composing existing services to a desired output must be possible with an 100% accuracy rating. At the same time, the need to process such requests in an efficient way raises the need for better and faster methods than those used today.

Driven by the experiences gained during last year's challenge, we decided to lay a strong focus on the indexing method and schema parsing, since those were the most challenging fields and probably will also be in the future. It is sometimes difficult to keep the balance between an efficient index that allows fast retrieval of stored data and limited memory size to keep the system scalable. Another possibility we had to keep in mind is an approach that does not rely on indexes but works with query-based methods directly on the underlying data. That, of course, entails additional overhead during query processing, while no time has to be spent on indexing (and therefore memory requirements are extraordinary low).

Like last year we decided to stay close to the vision of an Internet populated by a huge amount of services and therefore rely on an indexing approach, even if we therefore had to investigate how to make the indexing and parsing process faster which is not service-related in the first place but necessary for the overall performance nevertheless. Furthermore, we laid a stronger focus on an efficient algorithm to select semantic relations for a given set of parameters, given in the sample queries.

This paper presents the VitaLab system VIECH (VitaLab Efficient Composition using HStrings), starting with an overview of the major components for indexing and query processing, and followed by a detailed description of the algorithms used for discovery and composition. Finally, we provide an outlook on the anticipated efficiency of both, our algorithms and our index structure.

2. The VitaLab System

The VitaLab system VIECH consists of three main modules which are illustrated in Figure 1:

- The Index component parses the descriptions of services and ontologies and creates an index for fast information retrieval.
- The Composition component performs the discovery and composition of Web services, based on the index.
- The Web service interface, based on Apache Axis2 [1], accepts query requests, forwards them to the composition component, and returns the results.

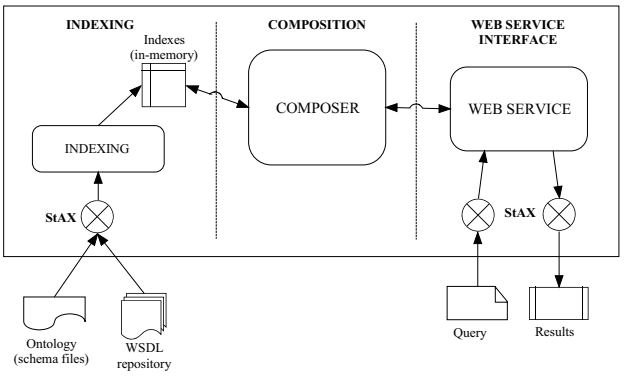

Figure 1. Overview of the VIECH System.

2.1 Parsing and Indexing

For processing large amounts of WSDL data and ontology descriptions, an efficient indexing strategy is of paramount importance. However, finding such a strategy is not trivial since requirements, such as low memory consumption and fast response time, are often conflicting. Therefore it is necessary to analyze the characteristics of the index to arrange its data structures accordingly. For instance, this analysis may include ratio and costs of updates and queries, the complexity of the indexed data, and general constraints, such as maximum memory consumption. For the development of the VIECH system we analyzed the sample input files and general requirements of Web service composition algorithms to find an optimal balance.

2.1.1 Parsing of XML

Traditionally XML parsing can be done by using either tree-based parsers such as DOM, or event-based ones such as SAX. While tree-based parsers read the whole document, keep it in a tree structure, and allow to operate on it in a convenient way, they are afflicted with an inferior performance and a large memory footprint. In contrast to this approach, event-based parsers return the XML document as a stream of tokens, which is a fast and lightweight technique, but requires the application to keep track of the document structure. Although both approaches have advantages and disadvantages, depending on the field of their application, a tree-based processing of large XML files is not reasonable. Therefore we use the StAX [3] parser WoodStox [4] to process only the relevant parts of the XML files, which results in high throughput and low memory consumption.

2.1.2 Indexing of Ontologies

Semantic composition of Web services is based on ontologies which describe types and their hierarchies. As these hierarchies solely contain subtype-supertype relations, the possible queries are also limited to finding (a) all existing types, (b) their subtypes, (c) their supertypes, and (d) checking whether two types are in any relation to each other at all. Therefore, the Type Index can be kept in simple hashtables, as illustrated in Figure 2.

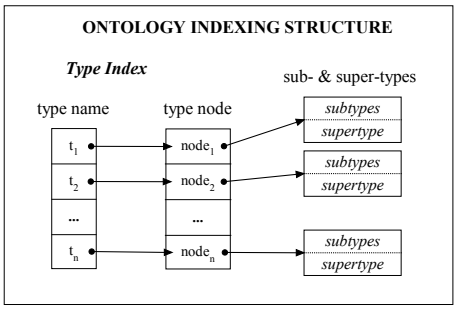

Figure 2. Structure of the Ontology Index.

For all existing types a *TypeNode* object is stored in the Type Index, which in turn contains all direct sub- and supertypes. Although, this implies that a retrieval of all – not only the subsequent – sub- or supertypes has to be performed in a recursive manner, it allows to keep the Type Index lightweight and scalable.

2.1.3 Indexing of WSDL Descriptions

The WSDL index structure consists of two tables: Partname Index and Service Index (see Figure 3).

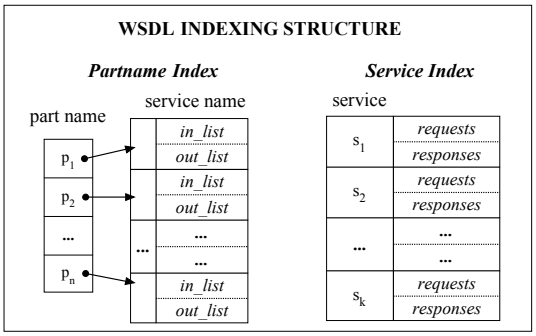

Figure 3. Structure of the WSDL Index.

The Partname Index uses a hashtable to maintain the mapping from each part name into two lists of service names: the *in_list* and *out_list* corresponding to part name

p are the lists of services which have p as request and response, respectively. In this way, it takes $O(1)$ to get a list of services that consume, or produce a particular part name p. The Service Index utilizes a hashtable that maps a service name into detailed information of the correspondent service (namely request and response part names).

2.1.4 Flyweight Design Pattern

For building the indexes we make intense use of *HString*, a Flyweight [6] implementation of strings. This technique aims at reducing the memory consumption in situations where identical objects can be mapped to a single one, instead of keeping an instance for each of them. We regard this pattern as especially useful for the indexes, since they consist of names of services and input-/output-types which occur in multiple hashtables concurrently. Moreover, we use the high performance collections GNU Trove [2] to efficiently manage and access the indexes.

2.2 Processing Algorithms

In this section we present the algorithms we use to solve semantic discovery and composition of Web services where the type information of input and output messages is encoded in XML schema and referenced by all WSDL files.

2.2.1 Semantic Discovery Algorithm

Considering a set of services s, the problem inherent in Web service discovery can be summarized as finding a result set of services that take as input a set of part names $provided$, and return as output a set of part names $resultant$. The part names are organized in a type hierarchy. For input part names, all subtypes have to be considered, while all supertypes can be used for output parts.

Algorithm 1 $semDiscovery(provided, resultant)$

Require: $resultant \neq \{\}$
1: $result \leftarrow \{\}$
2: $goals \leftarrow p_{response}(resultant_0)$
3: **for** $i = 1$ to $|resultant| - 1$ **do**
4: $\quad goals \leftarrow goals \cap p_{response}(resultant_i)$
5: **end for**
6: **if** $provided = \{\}$ **then**
7: $\quad$ **return** $goals$
8: **end if**
9: **for all** $s_i \in goals$ **do**
10: $\quad$ **if** $provided \subseteq_{SEM} in(s_i)$ **then**
11: $\quad\quad result \leftarrow result \cup s_i$
12: $\quad$ **end if**
13: **end for**
14: **return** $result$

Algorithm 1 works as follows: The set of services that can produce a goal is constructed from Line 2 to 5. We use $p_{response}(r)$ to get all services that have the part type (or subsuming types) r in their response. If $provided$ is empty, the set of goals is returned (Line 6 to 8). Otherwise, for all possible goals the service is added to the result, if the input parameters of the service represent a subset of the expanded set of provided part names and subsumed types, as expressed by the SEM suffix in $\subseteq_{SEM}$ (Line 10 to 12).

2.2.2 Semantic Composition Algorithm

Algorithm 2 relies on the semantic indexing as well as the semantic discovery algorithm described above to find a sequence of services that satisfy a given query. A service s_x is understood as a concept representing a tuple $\{in(s_x), out(s_x)\}$, holding both a listing of the input and output part names of the service. A composition query consists of two lists I and O, containing the given input parts of the query, as well as the desired output parts. Hence, to satisfy a query, a sequence of services has to be found which is able to return these output part names with the provided input parts.

Algorithm 2 $semComposition(I, O)$

Require: $I \neq \{\}$ and $O \neq \{\}$
1: $O' \leftarrow O$
2: $I' \leftarrow I$
3: **repeat**
4: $\quad s \leftarrow semDiscovery(\{\}, O')$
5: $\quad$ order s based on $max(in(s_x) \in I')$
6: $\quad$ **for all** $s_i \in s$ **do**
7: $\quad\quad$ **if** $in(s_i) \subseteq_{SEM} I'$ **then**
8: $\quad\quad\quad Mappings \leftarrow Mappings \cup \{out(s_i), s_i\}$
9: $\quad\quad\quad I' \leftarrow I' \cup out(s_i)$
10: $\quad\quad\quad O' \leftarrow O' \setminus out(s_i)$
11: $\quad\quad$ **else**
12: $\quad\quad\quad$ **if** $!(in(s_i) \subseteq_{SEM} O')$ **then**
13: $\quad\quad\quad\quad O' \leftarrow O' \cup in(s_i)$
14: $\quad\quad\quad$ **end if**
15: $\quad\quad$ **end if**
16: $\quad$ **end for**
17: $\quad$ **if** $O \subseteq_{SEM} I'$ **then**
18: $\quad\quad found \leftarrow true$
19: $\quad$ **end if**
20: **until** $found = true$
21: **return** $resolve(Mappings)$

Our approach uses an iterative backward search for achieving this goal, where in each run first all services are discovered that can deliver the required output parts (Line 4). This listing is then ordered by the degree to which their input parts match the already available inputs found by

the algorithm so far (Line 5). This ensures that promising services are prioritized over services with more unsatisfied parts and hence greatly optimizes runtime-performance.

In the next step (Line 6 to 16), all these services are checked whether the available input parts I (and subsumed types) already fulfill their signature, in which case their output parts are added to the list of available input parts, and removed from the list of required outputs (Line 9 and 10). Besides, for each output part an entry is added to a mapping table that holds information on how to get to the respective part, allowing to backtrace the solution in the end (Line 8).

In case the necessary input parts are not available, they will be added to the list of required output parts in Line 11 to 15 (as long as they or subsumed types are not already in either of the two lists). After all services are processed the search will be redone with a broader scope. Using this approach of an adaptive search pattern instead of a direct recursion favors building broad over deep search-trees, and leads to improved performance especially on large data sets.

Each iteration ends by checking if all required output types are already in the list of available input types (Lines 17 to 19). This means that a path to the desired goal has been found and the program will terminate. In a final step the mapping table is resolved to a path by recursively navigating through the mappings starting with the final output of the composition (Line 21).

3 Preliminary Evaluation

After our promising results from last year [5], we significantly enhanced the type and service indexing, as well as the algorithm used for service composition.

Our experimental evaluation was performed on a Dell Blade, 3.2 GHz Dual Xeon processor with 2 GB memory and SCSI hard disk with 10000 rpm. We used all the data sets provided by the WS-Challenge organizers. The results of the semantic discovery compared with the results of our solution from the WS-Challenge 2006 are illustrated in Figure 4. The tests were performed ten times for each data set and the average value was calculated. Each query file consists of ten different discovery queries per data set. The main time is consumed during the first run, where the complete in-memory index has to be built.

The performance is significantly better than last year, we now need approximately 18 % of the original time for the indexing and 25 % for the discovery process.

The results of the semantic composition are currently not stable for publication, therefore, we do not present detailed performance numbers here. Nevertheless we can say that the performance of the composition also increased significantly due to a faster index structure.

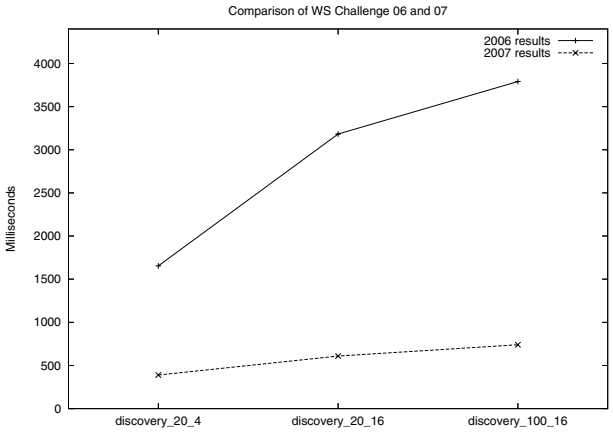

Figure 4. Performance comparison.

4. Conclusion

The VIECH system represents our contribution to this year's Web Service Challenge. The goal of this challenge is to implement a system that provides semantic discovery and composition of Web services. For the construction of our system, we learned from the experience gained during last year's Web Service Challenge. Primarily, we re-used the VIECH system from last year as foundation. Furthermore, we introduced a completely new composition algorithm, and also significantly improved the indexing component using the Flyweight Design Pattern and high-performance collections. The preliminary results clearly show an enormous performance gain for indexing and discovery. However, the composition algorithm still has to be thoroughly evaluated. Finally, to meet the requirements of the challenge, we adapted the VIECH system to provide its functionality as a web service using the Axis2 framework.

References

[1] Apache Axis2. `http://ws.apache.org/axis2`.

[2] GNU Trove - High Performance Collections for Java. `http://trove4j.sourceforge.net/`.

[3] *JSR 173: Streaming API for XML.* `http://jcp.org/en/jsr/detail?id=173`.

[4] Woodstox - High Perfomance XML Processor. `http://woodstox.codehaus.org`.

[5] M. Aiello, C. Platzer, F. Rosenberg, H. Tran, M. Vasko, and S. Dustdar. Web service indexing for efficient retrieval and composition. In *Proceedings of the IEEE Joint Conference on E-Commerce Technology and Enterprise Computing, E-Commerce and E-Services (CEC/EEE'06), San Francisco, CA, USA*. IEEE Computer Society, 2006.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1997.