Scalable Discovery of Linked Services

Barry Norton and Steffen Stadtmüller

AIFB, Karlsruhe Institute of Technology, Germany
firstname.lastname@kit.edu

Abstract. Linked Services bring together the principles and technology which define Linked Data with those of RESTful services. Services and APIs are thus enriched by, and contribute to, the Web of Data. A number of approaches to the semantic description of services have chosen SPARQL graph patterns as a means to describe input and output expectations. This paper describes two contributions. Firstly, the extension of discovery mechanisms offered over such Linked Service descriptions for use at application design time. Secondly, the realisation of this discovery approach in a distributed fashion using the Hadoop implementation of MapReduce, and an evaluation using private cloud infrastructure.

1 Introduction

Linked Data is identified with the application of a set of principles, and related best practice, regarding the use of RDF, SPARQL and HTTP in the publication of data. Linked Open Data is the result of the application of these principles in combination with the push to opening up public sector and other data.

Linked Open Services (LOS) [3] and Linked Data Services (LIDS) [9] look beyond the exposure of a fixed datasets using HTTP, SPARQL and RDF and consider how the data that results from computation over input can carry explicit semantics and be interlinked with existing Linked Data sets, and other dynamic data, thus also contributing to the Web of Data. It is natural that Linked Data's use of HTTP should be extended, consistently with REST [2], and that RDF should be made available, at least as an alternative via Content Negotiation, to encode representations to achieve these aims.

Other work [6] — which introduces the name 'Linked Services' that we consequently use to generalise over the different approaches to the combination of Linked Data and RESTful service approaches — introduces the idea that service descriptions should also be exposed as Linked Data. The original approach, pursued in the SOA4All project[1] and leading to the iServe service repository[2], uses a lightweight RDF version of the models used over many years of the literature on 'Semantic Web Services', the so-called 'Minimal Service Model'[3]. This uses semantic technology to characterise the input and output expectations of a service by annotating structural message parts with ontology classes.

[1] http://www.soa4all.eu/
[2] http://iserve.kmi.open.ac.uk/
[3] http://cms-wg.sti2.org/minimal-service-model/

LIDS and LOS both base their service descriptions on the notion that Linked Data provides a better description for services' input and output requirements: the graph patterns provided by the SPARQL query language. These provide the advantage of familiarity to Linked Data producers and consumers, but also of a more thorough description of what should be communicated and the possibility for increased tool support. One such kind of tool support is the extension of the notion of service discovery to better support a data-oriented view. This paper is organised as follows: Section 2 explores this motivation further and introduces the idea of design-time discovery of services using graph patterns; Section 4 sketches our approach to implementing such a mechanism in a distributed, scalable fashion; Section 5 details our evaluation of the current status of this implementation and justifies its general feasibility; Section 6 then provides conclusions and introduces our planned future work.

2 Motivation

In order to motivate the advantages of graph-based IO description we introduce an example based on wrapping some social network platform that only provides an API, not a dump of Linked Data (there are many such, so we do not need to be specific). A common feature of these social networks is the display of a 'birthday list', i.e. a list of people with a birthday on the day of access. Encouraging use of APIs, rather than crawling, a call might be provided to retrieve such a list. The Linked Services approach would be to wrap this call to provide the information in RDF, reusing existing vocabularies. Here the natural choice is the 'Friend of a Friend' (FOAF) vocabulary[4].

A wrapping for such a service then, putting aside issues of security and credentials for which our solution would build on WebID[5], would make explicit the input of a person identified in that social network, and an output that gives the age of certain friends. Furthermore, following Linked Open Service principles[6], reproduced in Table 1, we would use graph patterns to do so (Principle 1), where the `foaf:knows` predicate is the link between the input and output (Principle 3). The resulting input and output descriptions for the service are represented in Table 2

1. Describe services' input and output as **SPARQL** graph patterns
2. Communicate **RDF** by RESTful content negotiation
3. The **output** should make explicit its **relation** with the **input**

Table 1. (Core) Linked Open Service Principles

[4] `http://xmlns.com/foaf/spec/`
[5] `http://www.w3.org/wiki/WebID`
[6] `http://linkedservices.org/wiki/LOS_Principles`

Input: {`?user a foaf:Person; sn:id ?uid.`}

Output: {`?user foaf:knows [sn:id ?fid; foaf:age ?age].`}

Table 2. Basic Description for Example Service

We elide the namespace prefix declarations — for `foaf` and `sn`, the social network — as these are defined across the service description. Note that the reuse of the variable `?user` across input and output imply that this will have the same binding in the output as provided in the input. In SPARQL terms these would be 'safe variables' if we considered the service to be a CONSTRUCT query from the input to the output patterns (which indeed is the approach taken in Linked Data Services, where the same type of service descriptions are given) .

The advantages of this graph-based approach to the description of inputs and outputs can be seen in comparing the description to a traditional 'semantic web service' model such as OWL-S[7], where syntactic messages (with the implicit assumption that these will be bourne in plain XML) are annotated simply with classes. Here both the input and output message of the service would simply have to be annotated with the `foaf:Person` class. Any other information on which is required for input, and can be expected from output, would either be hidden in non-semantic transforms for 'lifting' and 'lowering' (usually using XSLT) and/or in pre- and post-conditions in a non-standard rule language (usually SWRL). In fact it is not clear whether properties applied to instances in a postcondition are expected to the returned in the services communications as these are also used to indicate changes in the state of the world outside of the communications (e.g., the dispatch of a physical book in a book ordering service).

The Web Service Modeling Ontology (WSMO)[8] improves somewhat on the vague 'semantic' description of accepting instances of Person, and providing back instances of Person, by two means. Firstly, WSMO models an explicit *choreography*, which in the Web Service Modeling Language (WSML) is defined by a language fragment that indicates not just which classes (called concepts in WSML) are communicable, but also which *relations* (a n-ary generalisation of binary RDF properties) are. Unfortunately, indicating that instances of `foaf:age` are communicated does not tell the user whether the submitted user's age is returned, or whether their friends' ages are. In other words it could, in graph pattern descriptions, correspond either to the output shown in Table 2 or of: `?user foaf:age ?age; foaf:knows ?friend.`

Secondly, WSMO distinguishes *assumptions* from preconditions, and *effects* from postconditions, so that pre- and post-conditions may apply only to communicated data. Unfortunately WSMO implementations do not respect this difference. In fact IRS-III [1], a version of the Internet Reasoning Service that is one of the two reference platforms for WSMO, deliberately uses conditions in the assumption field to judge the applicability of services, given concrete input data (and not the exterior 'state of the World' that they are otherwise specified to model).

[7] `http://www.w3.org/Submission/OWL-S/`

[8] `http://www.w3.org/Submission/WSMO/`

The LOS and LIDS approaches to the use of graph patterns for service descriptions can be traced back to an extension of the OWL-S, in the work described in [8]. Here, though, this was cast slightly differently: as pre- and post-conditions. It is our belief that simply replacing the entire class-based annotation of messages with patterns is ultimately more comprehensible and useful than trying to reconcile an annotation with these conditions, especially since they have unclear semantics that are not directly related to communication. It is our hope, furthermore, that this approach will continue to grow in popularity due to the more useful descriptions that can be formed, while sticking to the 'lightweight' semantic languages — RDF(S) and SPARQL — that have been at least partly responsible for the rapid growth of Linked Data. For this reason we introduce a discovery mechanism that aids in the use of such service descriptions.

3 Approach

In our approach to discover Linked Services with graph pattern-based I/O descriptions we allow service requests to be formulated as *service templates* with the the same syntax than the service descriptions. So a service template is also a pair of SPARQL graph patterns: one representing the set of all possible *input* RDF graphs an agent (human or machine) can provide for the invocation of a service and one representing a template of *output* RDF graphs such an agent expects to be delivered from the service. Therefore service templates encapsulate all necessary request information needed, including the expected relation between the input and output.

Therefore the question of whether a given service description matches a service template correlates to the problem of graph pattern containment. The input graph pattern of a service description must be contained in the service template's pattern; every graph that satisfies the template input graph pattern must also satisfy the service description's input graph pattern. Intuitively this is to say that the input an agent can provide fulfils the needs of the service to be invoked. Note that this also allows for the agent to talk about additional data he can provide for service invocation even though a matching service does not require them.

Matching the output graph patterns works in an analogous way. The output graph pattern of a service description contains the output graph pattern of a template; i.e., every graph that satisfies the service description output graph pattern also satisfies the template output graph pattern. So the required containment relation of the output patterns is dual to that of the input graph patterns. Intuitively again this means a service output has to provide enough to satisfy the request, but can provide more.

The matching based on graph pattern containment subsists in two binary decisions (one for the input and one for the output), answering if a service description completely matches a service template. In addition to this, in order to provide a more flexible discovery approach, we allow for the ranking of service descriptions against service templates by providing a number of continuously-valued matching metrics.

To achieve this, we introduce two additional metrics:

- The *predicate subset ratio* (psr) measures to what degree the set of predicates used in one pattern are subsumed within the set used in another.
- The *resource subset ratio* (rsr) measures to what degree the set of named resources, in subject or object position, used in one pattern are subsumed with those of another pattern.

Intuitively these metrics indicate to what degree a service description and a service template are using the same vocabulary. If a service description does not match a template in terms of pattern containment completely, they allow to test whether they at least use some of the same resources and predicates (and to what degree). Therefore they provide a mechanism to discover services, which are close to a given template, but are not necessarily completely matching.

Similarly to the pattern containment we have to distinguish between the metrics for input and output. Since a template input graph pattern can offer more data than actually needed by a described service without endangering their compatibility, the subset ratios for the input patterns have to measure, to what degree the resources (respectively predicates) in the service descriptions are used in the service template. For the subset ratios of the output patterns this works the other was around, because a described service can offer more output than required be the template. So in this case the subset ratios have to measure, to what degree the resources (respectively predicates) in the template are used in the service description. The Equations (1)-(4) show how the metrics are calculated.

$$psr_{input} = \frac{\#\left(\{predicates\ in\ template\} \cap \{predicates\ in\ service\ description\}\right)}{\#\left\{predicates\ in\ service\ description\right\}} \tag{1}$$

$$psr_{output} = \frac{\#\left(\{predicates\ in\ template\} \cap \{predicates\ in\ service\ description\}\right)}{\#\left\{predicates\ in\ template\right\}} \tag{2}$$

$$rsr_{input} = \frac{\#\left(\{resources\ in\ template\} \cap \{resources\ in\ service\ description\}\right)}{\#\left\{predicates\ in\ service\ description\right\}} \tag{3}$$

$$rsr_{output} = \frac{\#\left(\{resources\ in\ template\} \cap \{resources\ in\ service\ description\}\right)}{\#\left\{resources\ in\ template\right\}} \tag{4}$$

Note, that if a graph pattern is contained in another one (i.e., the binary decision is positive), the subset ratios must necessarily result in a metric of 1.0.

As an example again consider an agent looking for a service that takes information about a person represented in the FOAF vocabulary as input, and provides the name and the age of the friends of this specified person as output. Further consider two potentially useful service descriptions. The first is for a service that expects input exactly as proposed by the template, but only provides the age of the person with the `foaf:age` predicate. The second is for a service that provides the name, age and the OpenID as output with the predicates `foaf:name`, `foaf:age` and `foaf:openid` respectively, but additionally requires a userID from a specific social network (from which the service would actually obtain its data). In this case the input graph pattern of the template does not promise all the data needed to invoke the service. On the other hand, only this service covers all the data the template requires in the output (as well as some additional, not needed).

The discovery process is depicted in Figure 1. In this example we would find that the agent who submitted the template can invoke the service described with the first service description but he only gets a subset of the output data he demands for. But taking into account that the $psr_{output} = 0.75$ and the $rsr_{output} = 1.0$, this service is still close and could be presented to the agent, for example with an additional notification, that some of the requested output will not be there.

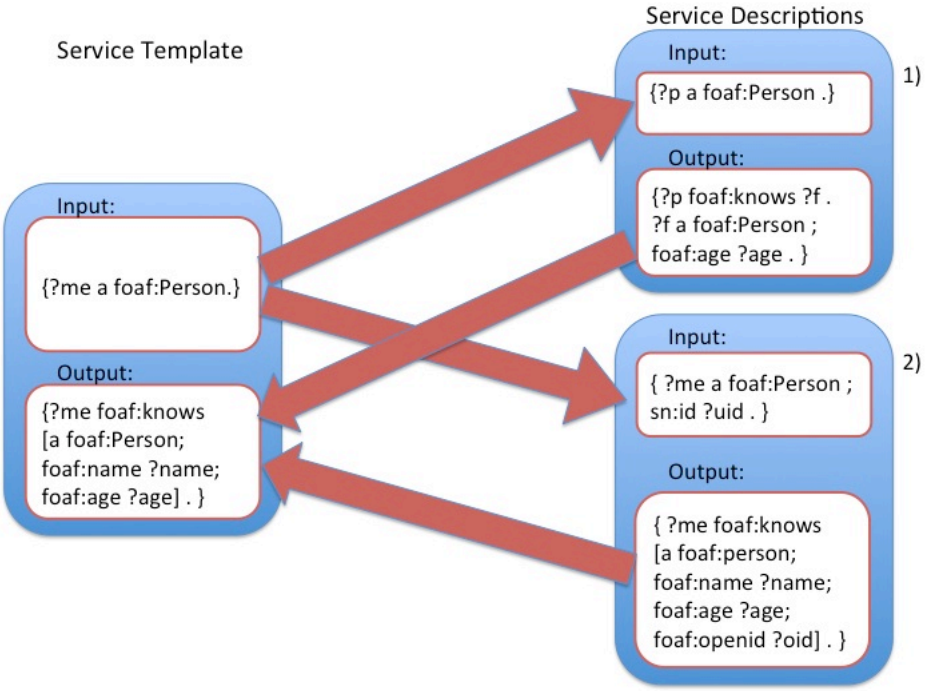
The second service description on the other hand represents a service that provides all the output data the agent is looking for. Here it is the input in the template that does not promise all the necessary data, from the service's point of view. Again, however, by looking at the subset ratios ($psr_{input} = 0.5$ and $rsr_{input} = 1.0$) it becomes apparent that this service could be useful for the agent, since it is very close to the template. In this case the agent could be specifically asked to provide the missing input for the service invocation.

To realize the described discovery approach and to address scalability we propose a discovery cloud, which provides as service description repository as well as template repository. This distribution cloud offers a RESTful [2] repository API to manage the service descriptions and to allow agents to submit new templates and retrieve the calculated metrics, or rather the discovered services ranked according to these.

Now the described semantic discovery process is applied at the following stages:

- When a new service template is uploaded to the discovery cloud, semantic discovery is used against every service description.
- When a new service description is uploaded to the discovery cloud every service template will be checked against this new service.

The insight pursued in our implementation, described in the following section, is that both of these problems can be structured as a MapReduce problem, with a map over the other type of resource, followed by a simple reduce. An important consideration, as with any MapReduce problem, is the locality of data; i.e., that the computation is reasonably well isolated from the communication of large amounts of data.



	Service Description 1)		Service Description 2)	
	input	output	input	output
pattern containment	YES	NO	NO	YES
psr	1.0	0.75	0.5	1.0
rsr	1.0	1.0	1.0	1.0

Fig. 1. Example for the discovery process

4 Implementation

In order to realise this approach for the discovery of Linked Services we implemented a system, that stores service descriptions and templates, both of which can be considered to consist of a URI as unique identifier and two SPARQL graph patterns that describe service input and output.

Work already exists on the provision of RESTful repositories for service descriptions, and provision of these according to Linked Data principles [4], [7]. Our implementation applies the same approach also to templates. In practical terms, when an RDF representation of a service description or template is HTTP POSTed to the repository, a URI is returned by which it is identified as a resource. In the standard REST manner, the resource can be updated by PUTting a new representation to the same URI, and it can be removed by a DELETE.

By managing templates as persistent resources, discovery against the template becomes an on-going task. Each retrieval of the list of services matching a template may be different due to two forms of dynamicity. First new service descriptions may be added to the repository, or indeed removed, between retrievals. Secondly, dynamic aspects of the service may affect ranking of each list; for instance the Quality of Service provided by the described service, as this is monitored over time.

When an agent (human or machine) would like to find services for a given task, then, a template can be POSTed. Every submitted template is stored and matched with all currently stored service descriptions. We use Jena[9], a Java framework for building Semantic Web applications, and ARQ[10], a query engine for Jena, to implement the matching mechanism. To determine if a service description matches a template we calculate, if the graph representing the input in the service description is 'contained' in the graph representing the input in the template; equivalent to checking whether the promised input is enough to invoke the described service. Analogously it is determined if the graph pattern representing the output of the template is contained in the graph representing the output in the service description. This is equivalent to checking, if the output of the described service is enough to satisfy the demands of the agent.

To determine these containment relations, the patterns for input and output (of templates and service descriptions respectively) are parsed and used as "WHERE"- clause of ARQ SPARQL ASK queries. Additionally the variables in both patterns are substituted by generated resources, resulting in graphs, which are skolemized versions of the original patterns. To test if a graph pattern A is contained a graph pattern B, we execute the ASK query of A over the skolemized version of B. The query will result in a positive answer, iff B contains A.

To allow for a sophisticated ranking, instead of just a binary discovery decision, we additionally calculate two other metrics: The *predicate subset ratio*, which measures the fraction of predicates used in the input graph pattern of the service description, that are also used in the input graph pattern in the template (and vice versa for output graph patterns). The *resource subset ratio* analogously measures the fraction of resources in subject or object position of the triple patterns in one graph that are used in the other.

[9] `http://openjena.org`
[10] `http://jena.sourceforge.net/ARQ`

These two metrics provide a continuum of matches be expressing to what degree the template and the service description use the same vocabulary. To calculate the two ratios for input and output respectively ARQ SPARQL SELECT queries are executed over the skolemized graphs to extract the set of predicates (resources) used in the graph patterns. Each set of metrics, generated in this way, for every combination of template and service descriptions is tagged with an identifier consisting of a combination of respective service description URI and template URI.

To allow service descriptions to be updated, or to populate the system with new service descriptions, an analogous process is employed. A submitted (via HTTP POST) service description is stored in the system and matched with all service templates. The resulting metrics are also tagged with an identifier and complement the already existing results. Thus, every combination of template and service description has a set of results that is persistently saved and can be retrieved from the system via HTTP GET.

If the system is populated with several thousands of service descriptions, the amount of calculations for determining all the metrics can be quite high. However, to provide for scalability of our approach we use Apache Hadoop[11], the open-source implementation of Google MapReduce. The Hadoop software is designed for distributed computation by dividing computation jobs into smaller sub-tasks, which can be executed in parallel on different nodes in a cluster of machines (map function). The results of these sub-tasks are retrieved and combined to achieve the overall goal of the original computation job (reduce function). For this purpose Hadoop implements a distributed file system (HDFS), which spans over an arbitrary number of computers. Data is stored on blocks of this file system; these blocks are distributed randomly over all nodes in the cluster. If the input data of a computing job is spanning over several blocks, a sub-task for every block is created and executed on the node where the block resides. Additionally the blocks can be replicated several times to provide a safe mechanism against failure of nodes.

To deploy our system we use OpenCirrus[12], a collaboration of several organizations to provide a research testbed for cloud-based systems. The Karlsruhe Institute for Technologies cluster within OpenCirrus makes use of OpenNebula[13] toolkit, an open source software used by OpenCirrus to build an "infrastructure as a service"-cloud (IaaS). This environment allows us to easily create and configure virtual machines that act as independent computers. We use these machines to set up a Hadoop cluster. This implies, that our cluster runs on top of a cloud, further abstracting from actual physical hosts.

We store the service descriptions, templates and matching results on a distributed HDFS storage, hosted on several virtual machines, as described. When a template, or service description, is submitted, Hadoop calculates the matching metrics by transferring and executing the code, that implements the matching mechanism together with the submitted template, to the nodes where the service descriptions (templates) are stored, rather than moving the data to the code. Furthermore Hadoop is "rack-aware", which means it always tries to use nodes close to each other (e.g. blades on the same rack in a datacenter) to reduce network traffic in the cluster.

[11] `http://hadoop.apache.org`
[12] `http://opencirrus.org`
[13] `http://www.opennebula.org`

Since our Hadoop system runs on virtual machines, whose communication is balanced by the OpenNebula Toolkit, we chose a flat structure, acting as if all nodes are hosted on the same rack. Finally Hadoop tries to balance the workload of the nodes, taking into account that some nodes contain the same data blocks due to the described replication mechanism.

After calculation of the metrics, the map function assigns a timestamp and the identifier to every set of metrics and passes the generated results to the reduce function. In our case the reduce function just gathers all the results and saves them persistently on the distributed file system. Since our system also allows for updating templates and service descriptions by re-submitting a new version of them with the same identifier, we have to run a second house-keeping MapReduce job. This second job compares the newly generated results with the results that are already stored. If a submitted service description is tagged with the same URI than a preexisting service description (i.e., an update is intended), some of the generated results will also have the same identifiers. In this case the older results are deleted, which can be checked by using the mentioned timestamps.

5 Evaluation

To evaluate our discovery system, especially in terms of scalability and in the absence of a large number of existing real service descriptions (since these number only in the double-figures so far, we have developed a generator. This create random pairs of related SPARQL graph patterns within boundaries, set by certain parameters, described below. These graph patterns can be interpreted as input and output of a service description or a template, because these are essentially equivalent.

For evaluation we generated 10 000 tuples used as service descriptions. Both patterns in these tuples are composed with a random number between 5 and 50 of triple patterns. The triple patterns for every respective pair are generated with resources in subject or object position, randomly drawn out of a local resource pool consisting of 10 to 50 different (URI-identified) resources. These local resources are randomly drawn out of a global pool of 500 resources. The predicates in the triple patterns of the tuples are also randomly drawn out of 3 to 25 different predicates in a local predicate pool. And again this local pool is randomly chosen out of a global pool of 250 predicates. So the difference between the local and global pools is, that the global pools of resources and predicates are used for all tuples, whereas the local predicates and resources are only consistent for both of the graph patterns within a tuple. This approach is chosen to establish a credible relationship between input and output.

Additionally the generator uses variables, rather than resources, in subject or object position with a probability of 0.3 in each case. A variable is used in predicate position with a probability of 0.2. In every tuple between 2 and 10 different variables are used. Since variables are already locally valid within one tuple, no global variable pool to draw from is needed. Additionally we generated a tuple of graph patterns used as template with the same parameters.

We populated the system with the service description using different setups with one, two, five, eight and ten worknodes in the Hadoop cluster, deployed on virtual machines of the OpenCirrus test bed. With every setup one additional virtual machine is needed to act as namenode for the Hadoop cluster. The namenode is used for the coordination of the distributed computation tasks, but does no computation itself. The distributed HDFS storage was configured with a block size of 1MB with a replication of factor 3 for every used block. The 10 000 service descriptions corresponded to 8.16MB of data and were therefore stored over 3 x 9 blocks on the cluster.

Then the matching process for the generated template over all service descriptions was triggered on every setup. We measured the execution time needed for the matching itself (i.e., first MapReduce job) and the overall execution, which includes the time needed for the second MapReduce job to combine the newly calculated with the preexisting metric sets. To provide for comparable results regarding the overall time, we did not prepopulate the system with results. Therefore the second MapReduce job used every time only the 10 000 newly calculated metric sets as input (i.e., one for every combination of service description and template).

worknodes	execution	time (sec)	mean (sec)	standard deviation (sec)	standard error (sec)
1	1.	477.3			
	2.	463.3	470.3	9.9	7.0
2	1.	283.7			
	2.	277	280.4	4.7	3.3
5	1.	169.7			
	2.	156	162.9	9.6	6.8
8	1.	155.3			
	2.	167.1	161.2	8.2	5.9
10	1.	134			
	2.	121.7	127.8	8.7	6.2

Table 3. measurements of overall execution time

worknodes	execution	time (sec)	mean (sec)	standard deviation (sec)	standard error (sec)
1	1.	394.3			
	2.	395.8	395	1	0.7
2	1.	223.6			
	2.	219.3	221.5	3	2.1
5	1.	120.6			
	2.	124	122.4	2.4	1.7
8	1.	121.7			
	2.	117.2	119.5	3.2	2.3
10	1.	81.4			
	2.	82.1	81.8	0.5	0.4

Table 4. measurements of exclusive matching time

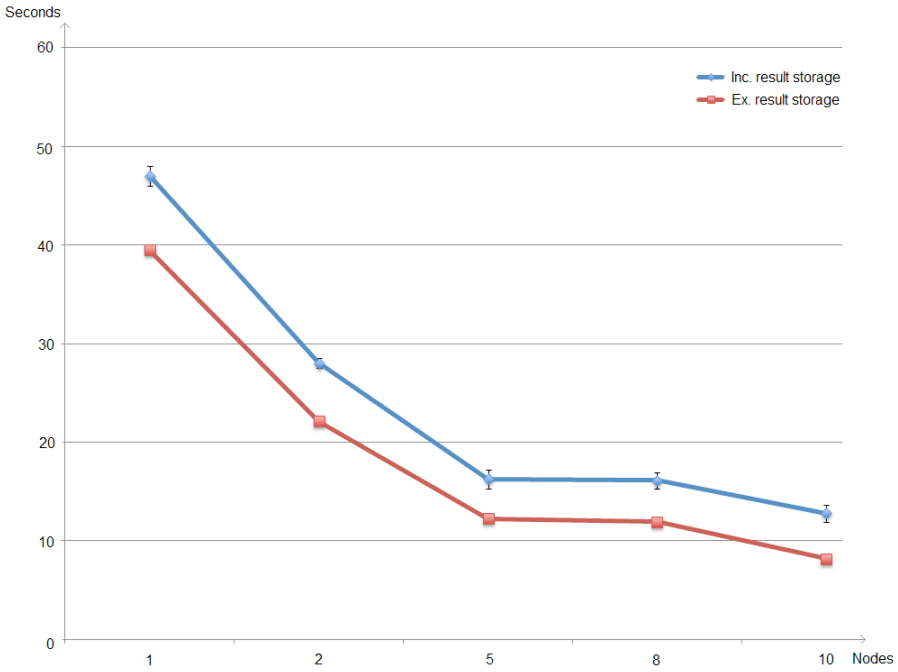


Fig. 2. graphical representation of measurements

To account for fluctuations in network traffic we measured the matching on each setup twice. The results are shown in Table 3 and Table 4 with a graphical representation in Figure 2. The calculation of the metrics alone took between 81.4 sec, on ten worknodes, and 395.8 sec, on one worknode. The overall execution time was measured between 121.7 sec using ten worknodes, and 477.3 sec on one worknode. It can be seen, that the system scales well by adding additional worknodes between one to five nodes. Between five and eight nodes the execution time stagnates almost completely. By using ten worknodes in the Hadoop cluster the measured times are further decreased compared to the setup with eight worknodes, although the improvement is less significant than in the area between one and five worknodes.

The behavior between one and five worknodes is easily accounted for by the possibility to execute more computations simultaneously. The more nodes are on the system, the more sub-tasks can be launched at the same time. By employing up to eight worknodes above five no further decrease in execution time is achieved because the Hadoop system was able at these settings to balance the workload with fewer nodes. Since the maximum number of map tasks executed by the system is determined by the amount of block the input data fills on the HDFS storage. So in our case not more than nine map tasks can be launched (eight, and one with a small input size of 0.19MB). Therefore on settings with one to five worknodes almost every worknode has to execute at least two map tasks. This provides the namenode with more posibilities to distribute the map tasks among the worknodes.

Such a distribution takes into account the fact that the worknodes contain non-disjunctive subsets of the overall input data set. The tasks can therefore be assigned in such a way that all employed worknodes contribute an equal amount of work to the calculation of the metrics. By using, for example, eight nodes, the namenode loses this possibility to some extent, since it always prefers parallel computation over load balancing (i.e., it will not wait for a worknode to finish if another worknode is already available). This also explains why a further, though diminishing, decrease of execution time is achieved by using more than eight nodes. In this case the namenode can (and must) decide not to use one of the nodes (and assigning the insignificant small map job to another). In this situation the least useful worknode is chosen to be disregarded by the namenode.

The effect of losing the possibility to balance the workload between the nodes can easily be avoided by choosing a smaller blocksize that allows for more map tasks than available nodes. But for our evaluation this observation also shows, that the improvement in terms of execution speed can not only be attributed to the increase of computation resources (i.e., adding additional CPUs and memory with every worknode), but also to the strategic distribution and execution of matching sub-tasks.

By comparing the results of the overall execution time and the matching time without housekeeping, similar observations can be made. The time needed to execute this second MapReduce job decreases due to the employment of a second worknode compared to a setup with only one node, but no further improvement can be achieved by adding additional nodes. The inputs for this second job are the calculated metrics, for every combination of service descriptions and template. They amounts to 2.68MB of data. So only 3 MapJobs can be start simultaneously.

The standard deviation and standard error of the individual results are also shown in the tables, and represented in the figure (as bars). For the overall execution time the standard deviation ranges between 4.7 sec and 9.9 sec, which results in a standard error between 3.3 sec and 7 sec. For the exclusive matching process the standard deviation is measured between 0.5 sec and 3.2 sec, which results in a standard error between 0.4 sec and 2.3 sec. Those values clearly indicate the stability of the system. The difference between the overall execution time and the exclusive matching time can be explained by the second MapReduce job that is included in the overall time, since most fluctuations in our measurements are due to the overhead time, that is needed to start a new MapReduce job.

Our results are not only valid for the matching a template over service descriptions, but also for populating the discovery system with a new service description, because they are syntactically equivalent and the process to submit a new service description is symmetrical to the process of submitting a new template. In other words the 10 000 used graph pattern tuples could have just as well been interpreted as templates, that are already stored on the system and one new service description (i.e., the former template) is submitted.

6 Conclusions and Future Work

In this paper we have: motivated the use of SPARQL graph patterns for the description of Linked Services; given an overview of one approach to discovery where both services and templates, representing data requirements from services, are described in this way; detailed an interface to such a discovery approach extending the existing notion of provision of RESTful and Linked Data-compliant registries; detailed a distributed implementation based on Hadoop; and described an evaluation of the scalability of such an implementation based on realistic parameters.

In future work we will expand on our coverage of SPARQL and RDF(S) in two major ways. Firstly, while the discovery approach detailed here concentrates on the conjunctive graph patterns, to which Linked Data Services restrict themselves, the Linked Open Services approach has motivated the use of disjunction. Consider, for example, a social network that allows its users to hide their full date of birth but to expose their birthdays. This means that for some friends they would be included in the results for a birthdays-based API call, but their age would not be included. In this case we might model the output as follows:

```
{?user foaf:knows ?friend.
 ?friend sn:id ?fid.
 OPTIONAL {?friend foaf:age ?age}.}
```

Similarly we have considered, related to the geospatial services considered in our previous work [5], that a service might be flexible in the use of vocabularies encoding input. While our existing services have used the Basic Geo Vocabulary[14] (usually given the prefix `wgs84`, `wgs84_pos`, or `wgs84_loc`), the Geo OWL ontology[15] follows GeoRSS in using Geography Markup Language (GML)-style objects (declared in a namespace usually given the prefix `gml`[16]) containing complex GML-defined literals. A service that accepts, as input, point in either of these encodings could be described as follows:

```
{{?point a wgs84:Point; wgs84:lat ?lat; wgs84:long ?long.}
 UNION
 {?point a gml:Point; gml:pos ?pos.}}
```

Similarly it could be a promising avenue to consider the use of FILTER expressions within the SPARQL graph patterns, indeed we could restrict, given dates of birth, the returned people in the running example to actually have their birthdays at the time of request by these means. This, however, seems like a better of use true rule-based post-conditions (or effects, in WSMO terminology) because it is not necessarily a property of the data communicated and the strength, demonstrated in this paper, of graph-based patterns is that they directly capture the information communicated by a service.

[14] `http://www.w3.org/2003/01/geo/`

[15] `http://www.w3.org/2005/Incubator/geo/XGR-geo-20071023/W3C_XGR_Geo_files/geo_2007.owl`

[16] `http://www.opengis.net/gml`

The provision for inference in the containment of one graph pattern by another is also part of our immediate future work. In our running example, the specification of an instance of `foaf:Person`, and likewise `Point`, is redundant since this is the domain of required predicates. There may be many such ways in which a pattern that is not directly contained may necessarily infer, in all matching patterns, the matching of another pattern. Inference also affects our continuous metrics in useful and interesting ways. This leads on to one final notable piece of on-going work: an attempt to establish the most effective means to combine matching metrics to define a useful ranking of services with respect to a template.

Acknowledgement: The work is supported by the EU FP7 projects SOA4All (IP 215219), and PlanetData (NoE 257641). We thank our colleagues from these projects for valuable discussions on the topics of this paper.

References

1. Cabral, L., Domingue, J., Galizia, S., Gugliotta, A., Norton, B., Tanasescu, V., Pedrinaci, C.: IRS-III: A broker for semantic web services based applications. In: Proceedings of the 5th International Semantic Web Conference (ISWC2006). Athens, Georgia, USA (Nov 2006)
2. Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
3. Krummenacher, R., Norton, B., Marte, A.: Towards Linked Open Services. In: 3rd Future Internet Symposium (September 2010)
4. Norton, B., Kerrigan, M., Marte, A.: On the use of transformation and linked data principles in a generic repository for semantic web services. In: Proceedings of the 1st Workshop on Ontology Repositories and Editors for the Semantic Web (ORES-2010). No. 596, CEUR-WS (2010)
5. Norton, B., Krummenacher, R.: Geospatial linked open services. In: Proceedings of Workshop Towards Digital Earth (DE-2010). No. 640, CEUR-WS (2010)
6. Pedrinaci, C., Domingue, J., Krummenacher, R.: Services and the Web of Data: An Unexploited Symbiosis. In: AAAI Spring Symposium (March 2010)
7. Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecky, J., Domingue, J.: iServe: a linked services publishing platform. In: Proceedings of the 1st Workshop on Ontology Repositories and Editors for the Semantic Web (ORES-2010). No. 596, CEUR-WS (2010)
8. Sbodio, M., Moulin, C.: SPARQL as an Expression Language for OWL-S. In: Workshop on OWL-S: Experiences and Directions at 4th European Semantic Web Conference (June 2007)
9. Speiser, S., Harth, A.: Taking the lids off data silos. In: Proceedings of the 6th International Conference on Semantic Systems (iSemantics). ACM International Conference Proceeding Series (2010)