Developing RDF-based Web services for supporting runtime matchmaking and invocation

Conference or Workshop Item

How to cite:


For guidance on citations see FAQs.

© 2011 IEEE

Version: Accepted Manuscript

Link(s) to article on publisher’s website:
http://dx.doi.org/doi:10.1109/NWeSP.2011.6088211
http://www.mirlabs.org/nwesp11/

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online’s data policy on reuse of materials please consult the policies page.
Developing RDF-based Web Services for Supporting Runtime Matchmaking and Invocation

Hong Qing Yu, Dong Liu, Stefan Dietze and John Domingue
Knowledge Media Institute
The Open University
Milton Keynes, United Kingdom

Abstract—In our previous research, we proposed an Autonomous Matchmaking Web Services (AMWS) framework, in which Web services broker themselves to notify the service registry whether they are suitable to the service requests. This framework is designated to more efficiently work for dynamically assembling services at runtime in a massively distributed environment. The central point of the AMWS is to use RDF (Resource Description Framework) to carry all exchanging messages. However, the implementation detail has not been discussed yet. In this paper, we focus on showing the two most important implementation parts of (1) transforming existing services to become AMWS compliant services that can consume and produce RDF messages; (2) service semantics can be annotated and self-brokered by using our developed Development Time Semantic Annotation and Lowering (DtSAL) Library at programming time.

Keywords: Web services, Autonomous Matchmaking, Semantic Web, Semantic Web Services, RDF.

I. INTRODUCTION

Automatic service discovery, selection, orchestration and invocation are the main research topics in the fields of service oriented computing. The majority of research efforts towards service automation is based on Semantic Web Services and targets the issues at the semi-automatic level, in which each individual task (e.g. discovery, selection, orchestration and invocation) is completed automatically by machines but human coordinates the whole service consumption lifecycle. Figure 1 shows the common process of development lifecycle based on current ideas of Semantic Web Services. Services are annotated with semantics that are stored in the service semantic repository. When service users want to consume services, they usually first discover services through a semantic broker, then invoke the corresponding services.

The work at the semi-automatic level can be further divided into two categories A and B (see Figure 2).

Category A fully trusts and depends on the Web service broker. In this way, the client is thin by just sending service specification and waiting for the expected response. The broker is a fat by taking care of all the dynamic activities. The main research contributions in this category are WSMX architecture [1] (e.g. IRS-III [2]) and OWL-S [12] based architectures (e.g. service composition application [4] based on planning algorithms). However, to be able to use these architectures, the user has to understand the data model of the services that registered in the broker and the complicated OWL or WSMO ontologies in order to invoke the discovered services and represent the output data correctly. Due to these prerequisites, all proposed orchestration managers are pre-defined static orchestration rather than runtime configurable.

Category B uses registration style to dynamically discover suitable services but leave other dynamic activities to the client side. In this way, client side can control service orchestration or select business strategy. The most recent contribution is Linked Services Concept [18] that uses Linked Data [13] principles to publish service semantics in to the Linked Data cloud.
Application [15] and [19] are developed based on Linked Services.

Semi-automation benefits to system development at design time, but it is not suitable for runtime environment in which system context changes dynamically. Furthermore, the following two issues have to be addressed in order to achieve service consumption at full-automatic level, in which the whole service consumption process lifecycle is automatically completed by machines without any human interactions.

- **Hidden data representation model** which causes the problem of sharing the common understanding of the service input and output data representations. The major reason is the contradictions between semantic description/annotation and non-semantic Web service implementation. First of all, all current service description languages focus on describing service interface rather than services themselves [26]. Secondly, no matter how services are semantically described or annotated, the underlying service invocation and response messages are still based on non-semantic invocation messages, such as SOAP [8], XML or Json1. Therefore, manually setting up the invocation message to map the parameters is unavoidable [22].

- **Out-of-date semantics** which leads the invocation faults. Current Semantic Web Service standards require service providers or users to register service semantics once service is implemented or used to a third party controlled service repository. However, the changes of service cannot be automatically reflected, especially for users who registered the service semantics but cannot know any changes on service side.

The main conceptual frameworks and specifications for semantically describing services (e.g. WSMO, OWL-S and SAWSDL which derive from WSDL-S [16]) are very comprehensive. Most SWS initiatives were built upon the enrichment of SOAP-based Web services to have semantics. Although some tools have been developed to foster to use these standards such as WSMO studio [20], these comprehensive semantic standards are too heavy to only show interface semantics of the service and still not describing the important part of the service – data representation model.

It is only most recently that lightweight services (e.g. Web APIs and RESTful services) and service annotations have been researched. The main results of these recent studies are SA-REST [16], hRESTs [25], WSMO-Lite [14], MicroWSMO [17] and MSM (Minimal Service Model) [21]. These standards are easily to be understood and adopted. However, current processes to use these lightweight semantics are still focusing on service annotations for implementing a big middle broker layer rather than thinking of adding semantic values directly to services themselves (e.g. iServe platform [21]). Therefore, the issues of runtime service invocation and out of date semantics are still remaining.

An Autonomous Matchmaking Web Services (AMWS) framework proposed in our previous work [9] aims to tackle the above issues and introduces a semantic message (RDF) based Web Service standard. In this paper, we move one step forward to discuss how semantic message based autonomous matchmaking Web services can be implemented. A development-time semantic annotation and lowering library is implemented to minimize the out-of-date semantics.

The remainder of this paper is organized into four sections. Section II plains the background and our motivation. Section III introduces the details of the service development. Section IV discusses the related work. Section V draws a conclusion and discusses our future research directions.

II. OVERVIEW OF THE AUTONOMOUS MATCHMAKING WEB SERVICE FRAMEWORK

Our initial work [9] has introduced the fundamental concepts of the Autonomous Matchmaking Web Service (AMWS) framework (see Figure 3). Comparing to traditional Semantic Web Services approach, there are two key changes:

- using RDF-based semantic message exchanging protocols rather than syntax-based message protocols (e.g. SOAP); and
- introducing semantic query endpoint for the service.

The invocation endpoint is as same as normal service invocation endpoint but only consumes and produces RDF messages. The semantic query endpoint takes RDF service request message as inputs and responses to the user by dynamically checking whether its own semantics satisfies the request. In this way, service registry only charges to broadcast the service requests in the suitable service category and let services themselves to decide if they are the matched services.

The AMWS framework brings at least two benefits for service-oriented computing.

- **Facilitating the full-automation of the service consumption process:** all information and communication messages are semantically understandable by using unified RDF data structure and LOD semantics. As result, the data structure and semantics are known at the same time, which is a fundamental requirement to enable services to be automatically assembled and invoked.

---

1 http://json-schema.org/
Balancing workload among services, requester and registry: each part of the three takes their own responsibilities to efficiently finish the service consumption life-cycle. Therefore, the AMWS framework is suitable for the large scale distributed applications.

However, current Web services standards are not support the semantic query endpoint and only have invocation endpoint. Therefore, converting existing services to enable implementing Autonomous Matchmaking Web Service Framework is the core challenge. The following section will give details of our current approach to transforming existing web services to the AMWS-compliant services.

III. IMPLEMENTING AMWS-COMPLIANT WEB SERVICES

This paper focuses on transforming existing services to become Semantic-Message-based Web services. There are following three key steps.

1. Transforming communication protocol to RDF: this is done by wrapping the existing Web service functions to have only one input parameter that is a RDF message string and only one output parameter that is a RDF message string too (see Subsection A for details).

2. Semantic annotating services while developing: Unlike Semantic Web Service technology, our annotation work is added while developers is wrapping the function or developing a new function. Therefore, the published semantics keeps with the latest updated functional information (see Subsection B for details).

3. Developing an extra function to be invoked as the semantic query endpoint: this function implements service matchmaking algorithms and determines whether the service is matched for the request (see Subsection C for details).

A. An illustration example: Wrapping DBpedia Web service

We use the DBpedia SPARQL query RESTful service\(^2\) as an illustration example to show the wrapping process. The detailed service information is listed in Table I.

<table>
<thead>
<tr>
<th>Table I. THE SPECIFICATION FOR DBPEDIA QUERY FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service properties</strong></td>
</tr>
<tr>
<td>Endpoint</td>
</tr>
<tr>
<td>HTTP method</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>format</td>
</tr>
<tr>
<td>query</td>
</tr>
<tr>
<td>debug</td>
</tr>
<tr>
<td>timeout</td>
</tr>
</tbody>
</table>

Figure 4 shows one instance of the RDF input message based on the defined RDF schema. It contains all the corresponding parameters for lowering.

Figure 5 shows the lowering technique used to translate the DBpedia SPARQL query function call from RDF request to the actual service request. HTTP GET method is used to invoke the function. The RDFInputParser is a key function from DiSAL (see subsection B) for lowering input RDF message based on a SPARQL statement. The fixed parameter value `format=rdf/xml` is used to create response in RDF format. Other parameter values are defined by parsing the input RDF message. After the service call, the results from the response are translated back to RDF format.

\[ <rdf:RDF> \]
\[ <dbpedia:Query rdf:id="query"> \]
\[ <dbpedia:default-graph-url> \]
\[ http://dbpedia.org/ \]
\[ <dbpedia:default-graph-url> \]
\[ <dbpedia:query> \]
\[ Select * Where { ?p a <dbpedia:Person> } Limit 10 \]
\[ </dbpedia:query> \]
\[ <dbpedia:debug off='true'/> \]
\[ <dbpedia:timeout>30</dbpedia:timeout> \]
\[ </dbpedia:Query> \]

\[ </rdf:RDF> \]

Figure 4. An example RDF input for the wrapped DBpedia service

```java
public String searchDBpedia(String RDFInputMessage){
    String RDFResponse=null;
    String sparqlQuery="Select ?query ?debug ?timeout Where...";
    String endpoint="http://dbpedia.org/sparql";
    RDFInputParser rdf=new RDFInputParser();
    HashMap<String, String> parameterIndividual=
    rdf.parseInputRDF(RDFInputMessage, sparqlQuery);
    String query=parameterIndividual.get("query");
    String debug=parameterIndividual.get("debug");
    String time=parameterIndividual.get("time");

    Httpclient client = new Httpclient();
   getMethod method = new GetMethod(endpoint);
    method.setQueryString(new NameValuePair[]{
        new NameValuePair("format", "rdf/xml"),
        new NameValuePair("query", query),
        new NameValuePair("debug", debug),
        new NameValuePair("timeout", time)},
    );
    try {
        client.executeMethod(method);
        RDFResponse = method.getResponseBodyString();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return RDFResponse;
}
```

Figure 5. Wrapping the DBpedia service to RDF-based AMWS.

B. Development-time Semantic Annotation and Lowering Library

To address the out-of-date semantic usages issue, we propose to add semantic annotations inside the code while developing or modifying service functions. To achieve this, we develop a Development-time Semantic Annotation and

\(2\) http://wiki.dbpedia.org/OnlineAccess
Lowering (DISAL) Java Library, whose implementation structure is shown in Figure 6. The DISAL uses openRDF\(^3\) and RDF2Go\(^4\) Java library to create RDF statements. A set of service description ontologies is applied to enable developer to choose differently according to the type of wrapped service (e.g. SOAP or RESTful). In the current implementation, the service description ontologies include hRESTs, WSMO-lite, micro-WSMO and MSM. Moreover, DISAL supports to lower the RDF document by giving a SPARQL query. The lowering function foster to both Semantic Message based Web services wrapping and new development processes.

![RDF lowering Annotation DISAL Java library Service Semantic ontologies openrdf & rdf2go Java library](image)

Figure 6. Implementation structure of the DISAL Java Library

Figure 7. illustrates the annotation code for the DBpedia SPARQL query RESTful service. The annotation mainly includes four parts: (1) constructing the namespace, service name, operation name, and invocation endpoint; (2) categorising service; (3) declaring HTTP invocation method; (4) defining input parameters and their model-references; (5) defining output parameters and their model-references.

```java
ra.setCategory("http://www.w3.org/2003/01/rdf-schema#Class");
ra.setMethodName("http://www.w3.org/2003/01/rdf-schema#Class");
ra.setInputElement("message\_i\_1", "query");
ra.setInputElement("message\_i\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ont\_create\_query\_endpoint");
ra.setInputElement("message\_i\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ontology\_Optional");
ra.setInputElement("message\_i\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ontology\_Optional");
ra.setInputElement("message\_i\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ontology\_Optional");
ra.setInputElement("message\_i\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ontology\_Optional");
ra.setInputElement("message\_i\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ontology\_Optional");
ra.setAddress("http://www.example.com/service\_input/");
ra.setOutputElement("message\_o\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ontology\_Optional");
ra.setOutputElement("message\_o\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ontology\_Optional");
ra.setOutputElement("message\_o\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ontology\_Optional");
ra.setOutputElement("message\_o\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ontology\_Optional");
ra.setOutputElement("message\_o\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ontology\_Optional");
ra.setOutputElement("message\_o\_1", "http://www.w3.org/2003/01/rdf-schema#Class" + "ontology\_Optional");
```

Figure 7. An hRESTs service annotation example

C. Adding Extra Functions to Support AMWS Framework

By using DISAL library, each wrapped service has its semantic to be computed in order to check whether the service can do the job according to the RDF based semantic service request message. Therefore, an extra function (semantic check endpoint) should be added to the service for answer the YES/NO to the registry by parsing the RDF request semantics and comparing to its own semantics if the service tries to support AMWS framework (see Figure 2). The response message of the semantic query endpoint follows AMWS Confirmation Response Message (CRM) standard that is represented in Figure 9. The CRM is composed by two parts of yes/no confirmation and runtime service invocation information.

![CRM ontology that is introduced in AMWS framework](image)

We implement an example of the semantic query function by using RDF2Go to parse the request semantics and checking the service semantic capability. The checking workflow is illustrated in Figure 10.

The matching algorithm used in the last step of the semantic query workflow is

---

\(^3\) http://www.openrdf.org/
\(^4\) http://semanticweb.org/wiki/RDF2Go
publishing existing Web services semantics to the L
cycle.
foundations and agree that service should communicate using
messages.
compliant to the AMWS framework to checking endpoint, the example of DBpediaSparql service is
implementation.
here only show a possible example of the semantic checking
workflow and matchmaking workflow
business logics and concerns, the implementation of the
semantic checking workflow and algorithm illustrated
here only show a possible example of the semantic checking
implementation.
With both service invocation endpoint and semantic
checking endpoint, the example of DBpediaSparql service is
compliant to the AMWS framework to enable to be asked in
broadcast way and invoked via RDF based invocation
messages.

IV. RELATED WORK
The Linked Services [21] and Linked Open Services (LOS)
[22] approaches both use Linked Data theory as their technical
foundations and agree that service should communicate using
RDF for supporting automatic web service consumption life
cycle.
The Linked Services approach focuses on annotating and
publishing existing Web services semantics to the Linked Data
cloud in order to discovery services using Linked Data theory.
Based on Linked Service proposal, service annotation model
(e.g. MSM), service annotation tools (e.g. Sweet [23] and
SmartLink [24]) and semantic description repository (e.g.
iserve) have been developed. Although, a preliminary service
invocation engine has been developed to use the semantic data
of the published services, it is still working on dynamic service
invocation level because it is very handful to setting up the
invocation RDF to match exactly the service invocation
lowering annotations. Moreover, two more issues remains:
(1) the underlying services are not added semantic values to
themselves as the semantics are stored in third party repository
that need to be manually updated when changing takes place;
(2) centralised service semantic repository leads to service
discovery and invocation are also through the centralised
environment, therefore, scalability issue cannot be resolved
easily.

The LOS approach focuses on wrapping existing RESTful
services or SPARQL endpoints to manipulate services that can
consume and produce RDF messages by using SPARQL
language such as Ask or Construct to support dynamic
lowering and composition. Therefore, LOS service semantic
model mainly described SPARQL graph pattern of input and
output. As far as our best knowledge, LOS does not support to
publish service semantics and dynamic service discovery
methodology. Moreover, LOS again only supports dynamic
service invocation rather than automatic because, the service
users need to know the lowering schema first in order to enable
LOS understand their RDF invocation requests.

V. CONCLUSION AND FURTHER WORK
In our previous research work, AMWS framework is
proposed to fully use Semantic Web technology to exchanging
messages between Web service communication protocols. The
advantages of AMWS are fully supporting automatic service
discovery and invocation at runtime. However, the feasibility
and implementation details have not been investigated. In this
paper, we introduce an implementation process to wrap
existing Web services (DBpedia SPARQL query RESTful
service in our case) to become an AMWS framework
compliant semantic message based Web service. The wrapping
process includes (1) reengineering Web service to enable
receiving and sending RDF semantic input and output
messages; (2) annotating services on Development Time using
DiSAL library and (3) developing and adding the service
semantic checking function to the service.

To fully support the whole working process to the AMWS
framework, many researches and implementations are
remained. We list three priorities in below:
• investigating the optimization algorithm to deal with
  the service selection issue caused by broadcasting
discovery methodology.
• developing an automatic mediation engine to mediate
different service annotation referenced terms from
different annotation ontologies. One possible way is
to enrich the Service Searching Message (SSM) with
the medication results.

![Figure 10. The semantic query workflow.](image-url)
• developing different DTsAL libraries to different kinds of current Web service programming language (e.g. C#).

ACKNOWLEDGMENT
The authors thank to EU funded Multi-type Content Repurposing and Sharing in Medical Education (mEucator) ECP2008EDU418006 project for supporting this work.

REFERENCES


