Experimenting Ontology Web Services

Maurizio Giordano and **Mario Mango Furnari**
Istituto di Cibernetica "E. Caianiello" - C.N.R.
Via Campi Flegrei 34, 80078 Pozzuoli, Naples - ITALY
{m.giordano,m.mangofurnari}@cib.na.cnr.it

Abstract

One application domain for *Service Oriented Architectures* (W3C Work. Group 2004) is ontology and metadata processing in the semantic web. Ontology-based approaches (Amann *et al.* 2002) pursue the interoperability among different sources of information by envisioning the development of software systems cooperation based on the sharing of semantics about data. *Ontology Web Services* are SOA implementations of ontology middlewares and systems functionalities published on the web and accessible by external applications via SOAP. In this work we report our experiences in developing and testing the Ontology Web Services of the ReMuNa (Remuna Project 2006) framework. The paper discusses the main advantages and drawbacks of using Web Services technology to implement services acting on ontology as input/output.

Introduction

The *Service Oriented Architecture* approach is today widely used in IT area for the design and development of distributed web-based applications. SOA mainly targets modularity of system design and interoperability of software components within the same system or towards external applications: by using the Web Services technology and its associated specifications and standard protocols, like SOAP (W3C Rec. 2003) and WSDL (W3C Rec. 2006), any software system may expose services on the web that can be accessible by other applications in a SOA environment.

One application domain for service oriented architectures is ontology and metadata processing in the semantic web. The idea of enriching web sites and their contents with semantic layer and metadata descriptions goes beyond the natural usage of this additional information layer within the context of a single web-site. Indeed ontology approach pursues the goal of interoperability among different sources of information by envisioning the development of application cooperation based on the sharing of semantics about data in different and/or in the same domain, data that the applications can process and exchange to perform certain tasks.

Several ontology middlewares and systems (J. Broekstra *et al.* 2002; Volz *et al.* 2003) offer their functionalities as services accessible via web-based protocol (SOAP or HTTP), like ontology navigation and browsing, querying services, ontology reasoning, format translations, and so on. Moreover ontology middlewares and systems have often a modular design with well defined APIs, and therefore can be easily integrated and/or can communicate with other systems to build up more sophisticated ontology processing functionalities.

Ontology Web Services (OWS) (Dameron *et al.* 2004) are Web Services implementations of ontology middlewares and systems functionalities to be exposed on the web and accessible by external applications via SOAP. OWS operates on ontologies as input and output data. Ontologies can be formalized in several languages with associated specific syntax and semantics, like OWL; the same ontology language can be expressed in different notations (RDFS, N-Triples, N3 etc.). Ontologies can pertain different domains of applications. The domain and the ontology language may influence the service type and constraints defined to act on the ontology.

The motivations underneath the adoption of Web Services technologies for the development of ontology services are several. First, Web Services are the *de facto* standard for application interoperability in internet-based distributed settings. Today most software development environments support Web Services programming in different languages (C, Java, PHP, etc.). These environments are equipped with tools and libraries to automate and support SOA programming and execution. Many tools and libraries are available that allow to quickly wrap-up existing applications (and their functionalities) as Web Services ready to be integrated in external environments. Thus, with this approach, today ontology middlewares and systems can be easily redesigned as SOA applications and their functionalities and APIs can be accessed via web by means of standard protocol (SOAP) processing.

Ontology Web Services are good tests for experimenting SOA approach and Web Services technology limits and advantages when applied to the semantic web area. In fact, ontologies are stored in large and often huge ASCII files. Dealing with large data streams stresses up the communication limits of this approach, due to both the overhead of large message transmission and to the additional protocol (SOAP) processing required. One of the main issue in Web Services

design and development is datatype mapping and translation from the service provider (server) to the consumer (client) programming environment, where this conversion is accomplished through a language-aware XML-Schema data representation adopted during data transmission.

In this work we report our experiences in developing and testing the *Ontology Web Services* of the ReMuNa framework (Remuna Project 2006). The ReMuNa project goal is the development of an infrastructure for content repository, retrieval and classification in the Cultural Heritage domain, where content providers are autonomous and interoperate in a internet-based distributed setting. In particular this work deals with Web Services development of the ReMuNa *Ontology Server*, i.e. the software component owning semantic descriptions (ontologies) of museum web site contents in order to guarantee the shared semantic layer for sites interoperability within the ReMuNa infrastructure. More details on the ReMuNa *Ontology Server* are in (Aiello *et al.* 2006).

Web Services Background

According to the Web Services (W3C Work. Group 2004) terminology, a *webservice* is a set of operations (i.e. procedures or methods) with associated signature definitions. The complete description of a webservice is expressed by a file in WSDL format (W3C Rec. 2006) and includes, besides the operation definitions, the unique location (URL) of the software entity providing the service (*webservice endpoint*), and the communication protocol supported (SOAP). The WSDL file also includes a detailed description of the encoding/decoding rules for data exchange during communication: data supplied as operation arguments to a webservice endpoint or data received as operation results are transmitted in a language-aware XML-Schema format whose complete specification is written in the WSDL file.

In Web Services programming, a *WS proxy* is a software module (class definition) generated automatically by processing the WSDL file on the consumer side. This module can be included in the consumer code to extend its functionalities, i.e. allowing the consumer application to execute remote webservice operations as they were local methods. In fact the WS proxy implements the same set of operations of the webservice: the proxy operation code implements all instructions required to accomplish service invocation, like SOAP message packing/unpacking and transmission. Data included in the SOAP message body are formatted into XML-Schema datatypes defined in the WSDL file as well. The WS proxy also deals with input (output) data encoding (decoding) from (to) the programming internal datatypes to (from) the language-neutral XML-Schema formats adopted for data transmission. All the aforementioned actions are hidden to the webservice consumer application and are carried out transparently by the WS proxy module.

The operations of a webservice can be executed in different interaction modes:

- *one-way* - the invoked operation does not request any response by the service provider;
- *request-response* - the invoked operation does request a response from the service provider;
- *solicit-response* - the operation requires the service provider to send a request to the client that will send back a response;
- *notification* - the operation requires the service provider to send a request to the client with no response from it.

In what follows we will discuss software systems providing a set of functionalities that can be used (invoked) by other applications via web, i.e. by means of HTTP-based interaction. There are several techniques and protocols that can be used to this scope. In the rest of the paper we will focus on two different and widely used approaches to expose software system services on internet: the *HTTP connector* and the *WS connector*.

A *HTTP connector* is a software layer that acts as a wrapper for a set of functionalities (APIs) provided by a certain software system. This layer enables HTTP-clients to invoke the system services in a language-aware protocol: the client performs a HTTP/GET requests on a specific URL and then it receives the service result in the body of the HTTP response (in HTML, XML formats or as MIME types).

A *WS connector* is a software layer that acts as a wrapper for a set of functionalities (APIs) provided by a certain software system. This layer enables Web Services clients to invoke the system services by using the SOAP protocol: the client sends a SOAP message (or makes a SOAP/RPC request) and then it receives the service result in the body of the SOAP response.

The difference between the WS approach and the HTTP-based one is that data contained in SOAP messages is encoded (decoded) in (from) XML-Schema according to well-defined conversion rules stated by the webservice provider in the WSDL file. The webservice consumer application uses WS tools and libraries, available in most programming environments, to process the WSDL file and to generate the WS proxy: this software module will handle all tasks required for webservice invocation on behalf of the consumer application, including data encoding/decoding required for message transmission.

In the rest of the paper we call *service* any functionality of a generic software system that is accessible on internet by means of a *HTTP connector* or a *WS connector*, that is either accessible with HTTP/GET request-response interaction or via SOAP messaging (and SOAP/RPC call mechanisms).

The Ontology Server of ReMuNa

The ReMuNa (Remuna Project 2006) infrastructure is a cooperative and distributed system for the access and management of museum information, structured and organized in a network of web sites. The management of museum contents is autonomously carried out by independent information providers in the cultural heritage area in a decentralized way.

The *Ontology Server* is one of the component of the ReMuNa framework (Aiello *et al.* 2006). It provides access and management functionalities to authority knowledge information structured and expressed in forms of ontologies. This knowledge is used both for structuring and enriching

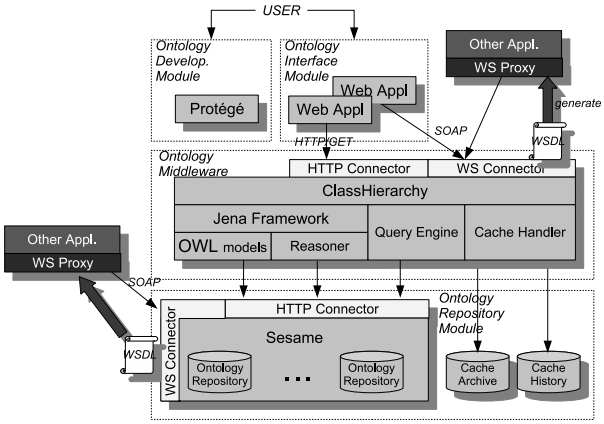

Figure 1: the *Ontology Server* architecture

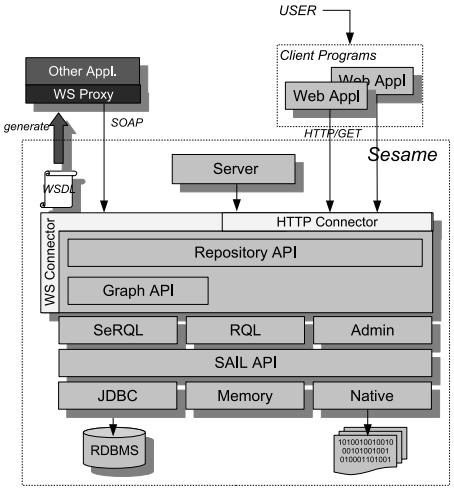

Figure 2: Sesame architecture

metadata descriptions of museum web contents, and to guarantee a semantic shared context for museum sites interoperability.

The main components of the Ontology Server are:

- *Ontology Repository Module* (ORM), implemented by the Sesame system (J. Broekstra *et al.* 2002) for the storage and management of ontologies in OWL/RDF.
- *Ontology Development Module* (ODM), implemented by the Protégé ontology editor (Noy *et al.* 2001) augmented with the development of a new plugin for ontology storage in the ORM.
- *Ontology Interface Module* (OIM), providing an interface to explore the hierarchy and query concepts and properties in the ontology graph. This module can be used to build-up more sophisticated user-interfaces for the query of the ReMuNa web contents repositories.

Ontology Middleware

Access and management services offered by the *Ontology Server* of ReMuNa are implemented in a *middleware* (Mahmoud 2004), that is a software layer interconnecting the interface module (OIM) to the repository module (ORM). We called this software layer the *Ontology Middleware* (OM).

The OM is a multilayer software. Its main component (the `ClassHierarchy` subsystem) provides navigation and browsing facilities on ontology schemas and individuals; other subsystems are included in the OM design to provide an inference engine and a caching subsystem for the efficient management of changes to the ontology repository.

As described in figure 1 the Ontology Interface Module of the OS is a set of web user-interfaces for the navigation, browsing and modification of ontology contents. These interfaces access data coming from the ontology repositories and processed by the OM layer. To do that, the web interfaces forward user requests to the OM *service layer*. The ontology middleware services are provided by means of a *HTTP connector* and a *WS connector*. Both connectors have been implemented on top of the Ontology Middleware to expose its services and make them remotely accessible by external applications.

While the HTTP connector was designed for internal use within the ReMuNa infrastructure and as an API between the Ontology Interface Module and the OM services, the WS connector was conceived to make OM services usable by other software systems outside the ReMuNa framework. Indeed this is one of the main features of SOA programming, where software functionalities (*services*) originally designed in one application context can be recoded as Web Services and later reused in different application contexts.

The WS connector has an associated WSDL file describing its services, operation signatures and data encoding rules for messaging. The WSDL is public and can be processed by any application developed in one of the widely used programming languages (C, Java, Python, PHP, etc.) by using a SOAP/WSDL library. For more details about the OM services see (M. Giordano and M. Mango Furnari 2006).

Ontology Repository Module

The Ontology Repository Module is implemented by the Sesame system (J. Broekstra *et al.* 2002). It is an ontology storing system based on RDF triples persistence in relational databases. Sesame supports several DBMS as back-end data stores. It also provides a set of services built upon the *storing layer* (SAIL) and implemented in a middle layer of its architecture (see figure 2).

Some services (the *Graph API* of figure 2) perform ontology schema and individuals navigation and browsing, like for example *"get the list of concepts"*, *"get individuals of a concept"*, *"get the range of a property"* and so on. Other services deal with ontology querying in different query languages (RQL, SeRQl, etc.). Finally, in the same layer we find services for ontology repository management (the *Repository API* of figure 2), like inserting/removing schema assertions or ontology individuals, extracting ontology fragments from the repositories, and so on.

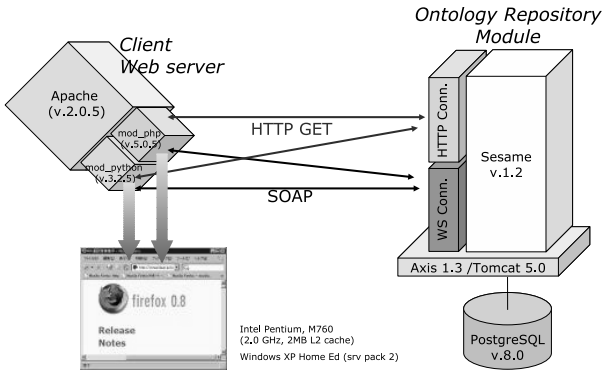

Figure 3: experimental setup

The Sesame system includes in the sources an HTTP connector: this software module makes accessible from outside only the services defined in the Repository API.

In the Ontology Server architecture, the Sesame HTTP connector is an interface between the Ontology Middleware core implementation and the underlying ORM: the user makes requests of ontology storing, updates and queries by using the web interfaces of the OIM; this module forwards the user requests to the OM; the middleware is responsible for the control and the completion of the requested tasks: in order to accomplish user requests the OM invokes services provided by the ORM through the associated HTTP connector.

We developed a WS connector for Sesame that implements the same set of services offered by its HTTP connector plus those related to ontology navigation and browsing. With this choice any software system outside the ReMuNa framework can directly invoke the Sesame ontology navigation/browsing webservices. On contrary, systems that are parts of the ReMuNa architecture use exclusively ontology navigation facilities provided by the OM *service layer*, since the OM is the software responsible for these tasks.

Experimental Results

In this sections we discuss a set of experiments we made to evaluate communication costs in the interaction between client programs (coded in Python and PHP) and the HTTP and WS connectors of Sesame. The tests refer to the *ontology extraction* facility of Sesame, that allows to select the ontology schema, individuals, or both with the option of selecting the ontology output format (RDFS, N-Triples, and N3).

Experimental setup

All tests were done by executing the service provider and consumer applications on the same host with the hardware configuration described in figure 3. Thus, the two programs communicate through the loopback ethernet interface with no access to the network.

The rightmost part of figure 3 shows the service provider of the experimental setup: the Sesame system executes in a Tomcat application server (Apache Tomcat Project 2006) augmented with the Axis library (Apache Axis Project 2006), representing the Web Services runtime environment required to execute the Sesame WS connector. The Sesame HTTP connector is a java *servlet* implementation in the Tomcat environment. Ontology schema and individuals are made persistent by Sesame in a PostgreSQL DBMS storage system.

The leftmost part of figure 3 shows the service consumer of our experimental setup. We choose two programming languages for the client side: Pyhon and PHP. They are two widely used web-programming (scripting) languages that offer good performances compared to more efficient compiled-based languages like C.

In order to set up a common execution environment for clients coded in Python and PHP, we choose to run the clients programs as server-side scripts on an Apache web server equipped with extension modules for the two languages. In figure 3 with the name *Client Web server* we indicates the PHP and Python clients programs running in the Apache web server execution environment and interacting with the Sesame connectors.

The web browser in figure 3 represents the graphic user interface: the programmer uses the browser to execute the client programs and to insert the arguments required for service invocation. The browser is then used to display the result that the service provider sends back to the consumer.

Measurements

In all experiments we measured the *communication time* of the service provider-consumer interaction: service invocation occurs once via the HTTP connector, and a second time through the WS connector.

Time measurements were done on both the service producer and consumer side: on the consumer side the *response time* is measured, defined as the time spent from the start of request sending to the completion of the service response reception; on the provider side, the *computation time* is measured, defined as the time required to execute the body of the method (procedure) implementing the service.

In the experiments we are interested in evaluating the *communication time* defined as the *response time* minus the *computation time*. In heterogeneous environments server and clients have different hardware, operating systems and TCP/IP implementations and network protocols libraries. Each of these aspects gives a contribution to the communication time; thus, it is difficult to isolate and estimate each single contribution within the system global behavior. Moreover the network traffic congestion in a LAN configuration can make time measurements unpredictable.

Since we are interested in evaluating the overheads of the HTTP and SOAP processing during communication, we choose an experimental setup (see figure 3) that completely removes the effects of network traffic. In this setup the server and client programs share the same hardware, operating system and TCP/IP libraries.

In all tests the service providers core implementation is the same: the same code implements the service methods regardless of the particular connector used; of course the

way HTTP and SOAP protocols are processed by service providers makes the difference in terms of performances. Note that both connectors run as web applications in the same application server environment (Tomcat).

Due to the particular setup of our experiments, if we fix the WS (or HTTP) connector for the service provider-consumer interaction, and we measure the communication time (defined as before), we are able to evaluate the performance of Python and PHP implementations of the SOAP/WSDL (or HTTP) processing libraries.

Service consumers

The diagrams in figure 4 describe the service consumer programs used in the experiments. The diagrams refer to the algorithms of two applications sending a service request to the HTTP connector (figure 4(a)) and to the WS connector (figure 4(b)) of Sesame. The algorithms are general enough since they are independent on the particular programming language and on the specific service invoked.

The two diagrams show the operations involved during service provider-consumer interaction. They also clarify the time measurements definitions of the previous section.

When the service consumer uses the HTTP connector it needs first to set the URL (*query string*) denoting the service name and location, as well as the parameters inputs. Then the client performs a HTTP/GET request on this URL starting the communication. This operation ends when all data have been received on the consumer side; the data can be printed (rendered by a web browser) or processed by the consumer application.

On the provider side, once the HTTP/GET request has been received, the parameters and service operation infos are extracted by the requests (*query string*) and the particular service operation is executed. The service provider sends back a HTTP response including in the body the operation result.

When the service consumer uses the WS connector it needs first to include the software module implementing the WS proxy of the webservice. As we already mentioned in the first section of this paper, the WS proxy is code that can be generated by means of tools available in most SOAP/WSDL libraries. The consumer application locates the WSDL file and processes it by means of such tools to produce the proxy module. The application invokes the proxy methods to actually execute the remote webservice operations. In this way SOAP message (packing/unpacking) processing, transmission and data encoding/decoding are transparently handled by the proxy module.

Once the WS proxy is instantiated and configured, the consumer application performs the communication within the extent of a proxy method invocation: the proxy sends the request and waits for the response from the service provider. When the proxy method returns the service result is soon available in the consumer application environment since it is transparently translated into the language-specific datatypes by the WS proxy.

On the provider side, once the SOAP request has been received, the WS connector processes it by identifying the

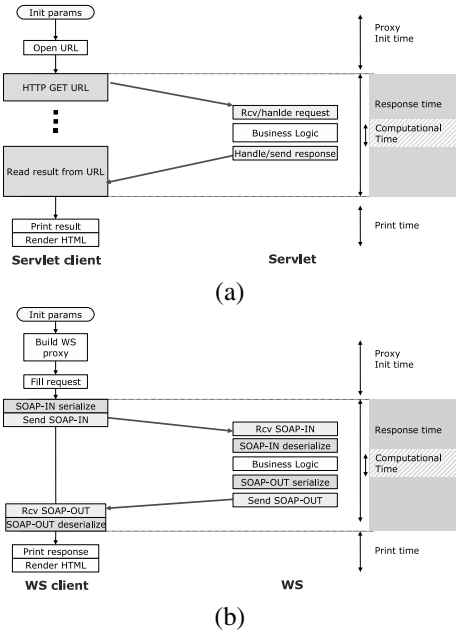

Figure 4: service provider-consumer interactions

operation to invoke and by extracting and decoding its arguments. The operation (method) body is executed to satisfy the request. At the end of the computation (*business logic*) the SOAP response is packed and sent back to the consumer. The WS connector before sending the SOAP response needs to translate output data into the XML-Schema formats defined in the WSDL file.

The echome test

In this paragraph we discuss a preliminary testbed we set up in order to simulate the behavior of the Ontology Server extraction service. To this scope we developed a synthetic application that provides a service with a single operation called *echome*: the operation returns a string of random characters with size equal to a given integer input. We developed the core implementation of the *echome* operation as well as its HTTP and WS connector.

The *echome* operation allows to vary the response message size by fine-tuning of the input parameter. The ontology extraction service has a similar behavior: it is possible to combine input filters to request the extraction of different parts of the ontology, thus affecting the size of the response message.

We run the *echome* service provider-consumer interaction tests with the experimental setup of figure 3: the set of measurements are reported in figure 5. We used these results in successive experiments as references for communication times spent in requesting (via HTTP or SOAP) a general service with response message sizes ranging from 1 MB to 8 MBs, with increments of 500 KBs.

The figure shows very small communication times (in the order of few hundred of milliseconds) for the HTTP connector. The communication time in the webservice provider-

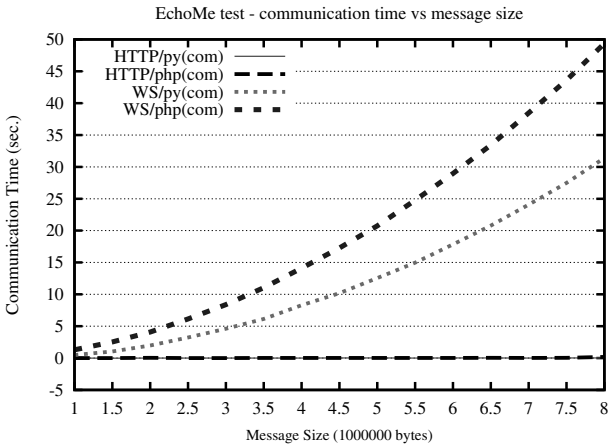

Figure 5: the echo test

formato	filtro	Msg size (B)
n3	DATA(no-inf.)	407514
	SCHEMA(no-inf.)	347088
	ALL(no-inf.)	755433
	DATA(with-inf.)	934688
	SCHEMA(with-inf.)	737176
	ALL(with-inf.)	1672705
rdf	DATA(no-inf.)	768157
	SCHEMA(no-inf.)	864865
	ALL(no-inf.)	1638659
	DATA(with-inf.)	1790276
	SCHEMA(with-inf.)	1831079
	ALL(with-inf.)	3626992
n-triples	DATA(no-inf.)	870646
	SCHEMA(no-inf.)	1293855
	ALL(no-inf.)	2164501
	DATA(with-inf.)	2855154
	SCHEMA(with-inf.)	3310385
	ALL(with-inf.)	6165539

Table 1: response message sizes vs extraction filters

consumer interaction does no scales, since it increases not-linearly with the response message size. Moreover the PHP client performs worst than the Python client. During the ontology extraction experiments we found the same results that we comment in the next paragraph.

The ontology extraction test

The experimental results discussed in this paragraph refer to the communication times spent by a Python or PHP application to execute the Sesame ontology extraction service, by using its HTTP and WS connectors. The graphs of figures 6 and 7 report these results.

The graph of figure 6 shows that the invocation of the ontology extraction webservice involves communication overheads increasing with the response message size. On the x-axis of the graph we report the ontology filters: they are ordered according to the increasing size of the message obtained by the ontology extraction.

So, for example, the `ALL(with-inf)` filter corresponds to the selection of all statements in the ontology, i.e. those related either to the schema or to individuals, with the inclusion of statements inferred by the Sesame system; the `SCHEMA(no-inf)` filter corresponds to the selection of only the statements of the ontology schema explicitly inserted by the user in the Sesame repository.

The target ontology, before inferencing, has almost 460 concepts, 1600 properties and 1000 individuals. Table 1 reports, for the target ontology, the response message sizes obtained by invoking the ontology extraction service with all possible combinations of filter and output formats. From this table it is clear that the N-Triples format produces the larger message. This fact is also clear from the topmost graph of figure 6 where the longer times correspond to ontology fragments transmission in N-Triples format.

The bottom graph of figure 6 reports the computation time required by the WS connector of Sesame to extract data from the repository and to prepare messages for data transmission. Notice that this time does not depend significantly on the ontology output format, while the communication time does. The Sesame WS connector gets the schema information from the repository in a time which is twice the one needed for individuals extraction. This result is in agreement with the numbers of individuals, concepts and properties of our ontology.

Different colored bars in figure 6 correspond to different clients (Python and PHP) invoking the same ontology extraction webservice but requesting the response in different output formats. For example, `WS/php(n3)` refers to a PHP client requesting an ontology extraction with N3 output format.

It is possible to associate the communication times of figure 6 with the corresponding message sizes of Table 1, and then to place these results in the graph of the *echome* testbed reported in figure 5. If you do so you may notice that the communication times of the the two tests overlap.

All time estimates in the topmost graph of figure 6 show that the Python clients perform better than PHP clients. The same behavior was obtained in the *echome* testbed: this is due to the bytecode-execution mode of Python scripts which is faster compared to the interpreted-execution mode of PHP scripts. Python performances cannot be attributed to the SOAP/WSDL library implementation. In fact the ontology extraction test involves a quite simple data conversion from the provider to the consumer application: the service output data is a string of characters representing the ontology fragment in a chosen format (RDFS, N-Triples or N3). The string conversion from Java language to XML-Schema and, then, to Python (or PHP) is carried out by the SOAP/WSDL libraries with minimal overhead.

In some situations the use of a particular SOAP/WSDL library implementation may introduce significant overheads compared to other solutions. An example is the invocation of Sesame query services. The results of queries are *vectors of tuples*, that is vector of arrays of URLs (strings). In this case the webservice interaction involves complex datatype translation. The query service WSDL file includes the encoding (decoding) rules for vectors of tuples into (from) XML-Schema. The datatype conversion may be very time consuming also for message size in the order of few MBs. The graph of figure 8 shows the communication time of querying with different response message sizes. You may note how the Python client behaves worst than the PHP

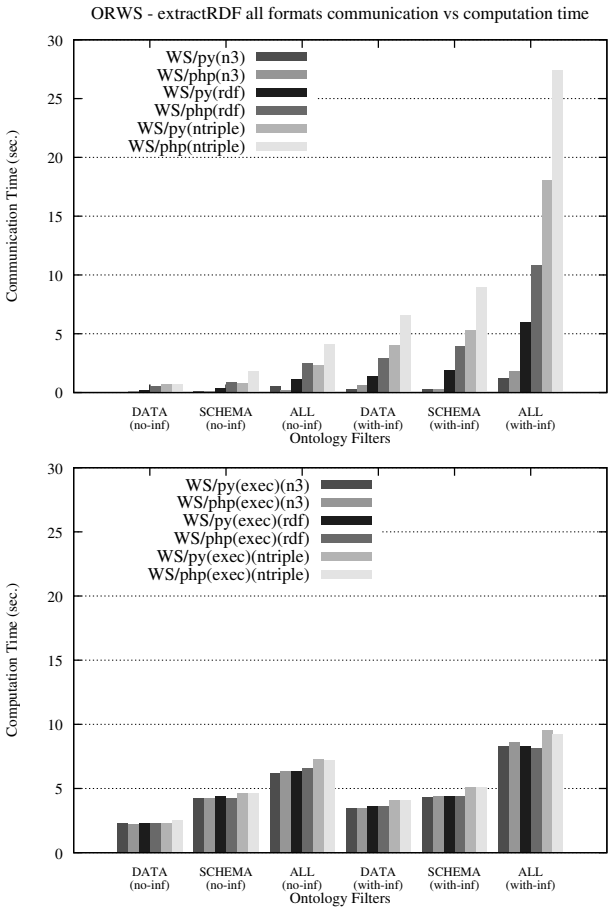

Figure 6: ontology extraction test: WS connector varying the ontology formats

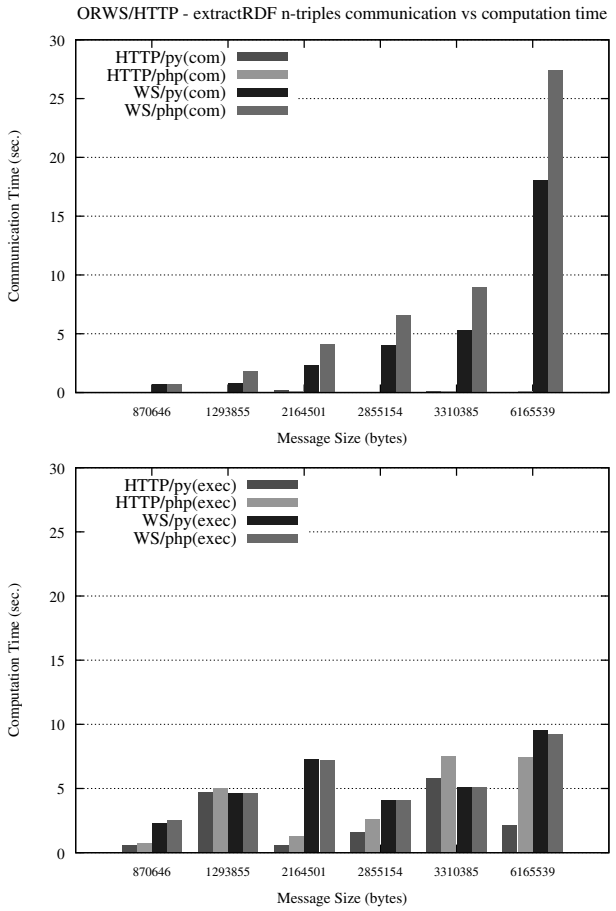

Figure 7: ontology extraction test: WS and HTTP connectors with a fixed ontology format

implementation. This is due to an inefficient implementation of the XML-Schema/Pyhon datatype conversion in the SOAP/WSDL Python library.

In figure 7 we report the communication and computation times of the ontology extraction service in four possible configurations, obtained by coupling the two client programs (PHP and Python) with the two Sesame connectors (HTTP and WS). The ontology format is fixed to N-Triples an we varied the extraction filters. In these cases we reported on the x-axis for both graphs the output message sizes.

In the topmost graph of figure 7 the communication times of the HTTP connector are not visible since they are in the order of few hundred of milliseconds. Also these tests confirm that the HTTP connectors outperforms the WS connectors in terms of communication time. The graph in the bottom part of figure 7 shows the computation times of the two connectors. We already commented that the WS connector takes a time to compute the ontology extraction request that increases with the amount of data to be extracted from the repository. This is not the case of the HTTP connector. This is due to the particular implementation of the Sesame HTTP connector, that uses caching mechanisms to optimize the extraction in two cases: the entire repository dump and the extraction of all individuals. Our WS connector implementation does not use caching since we decided to design the Ontology Server webservices as stateless.

Conclusions

From the experiences in designing and developing Web Services implementations for ontology middlewares and systems we learned that there is an intrinsic communication overhead in this approach: the SOAP message processing is time consuming and the communication costs increase with the message size. The SOAP approach shows scalability problems with respect to the increasing size of messages.

Ontology Web Services are implementations of software functionalities acting on ontologies files (or data streams) that may be very large. From our experiments we learned that the use of SOAP-based communication for ontology data transmission has to be limited to message sizes less that 1 MB in order to have response times compared to the HTTP-based request-response mechanism. We used for our tests the ontology developed in the ReMuNa project. The ontology extraction tests showed clearly the limits of the

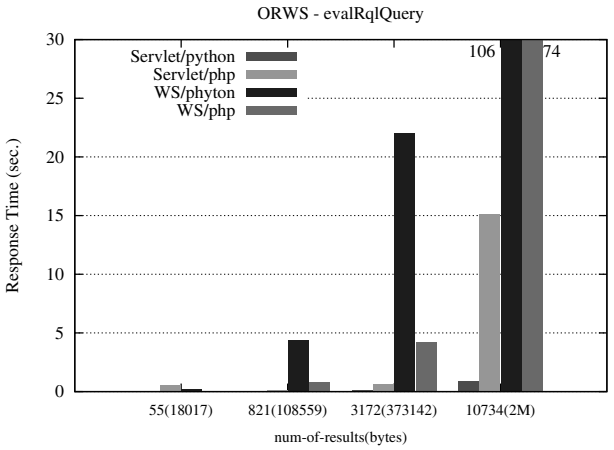

Figure 8: ontology query test

Web Services approach when applied to this ontology.

Sometimes the communication time is not the main aspect to be considered in designing and developing SOA environments. Another important issue may be fast integration of existing software in a new distributed and heterogeneous system architecture.

If this is the case, Web Services technology allows to speedup webservice building from existing software and to automate the generation of proxy modules on the service consumer side that transparently handles all tasks concerning SOAP protocol processing. The same proxy is in charge of translating information streams from the service provider internal datatypes to the ones of the consumer part; this is a fine solution to the typical data conversion problem of communication among heterogeneous software systems.

In our experience we tested several environments and libraries for Web Services programming. We realized that most of them are equivalent with respect to the set of tools and facilities offered to assist and speed up the process of webservice developing from scratch or as wrappers of existing software. What makes the difference is the support for automatic data conversion. Some environments provide support only for automatic data conversion of simple datatypes. Other systems offer data conversion libraries with high overhead (like the one discussed in the previous section). Thus, when webservice invocation involves complex datatype conversion the choice of the Web Services development platform is crucial, and wrong decisions may have bad consequences on the overall system performances.

References

Aiello, A.; Mango Furnari, M.; Massarotti, A.; Brandi, S.; Caputo, V.; and Barone, V. 2006. An experimental ontology server for an information grid environment. *International Journal on Parallel Programming*.

Amann, B.; Beeri, C.; Fundulaki, I.; and Scholl, M. 2002. Ontology-based integration of xml web resources. In Horrocks, I., and Hendler, J., eds., *Proc. of the First Intern. Semantic Web Conference - ISWC 2002*, number 2348 in Lecture Notes in Computer Science, 117–131. LNCS.

Apache Axis Project. 2006. The Apache Software Foundations. http://www.service-architecture.com/.

Apache Tomcat Project. 2006. The Apache Software Foundations. http://tomcat.apache.org.

Dameron, O.; Noy, N.; Knublauch, H.; and Musen, M. 2004. Accessing and manipulating ontologies using web services. In *Proc. of the Third International Semantic Web Conference (ISWC2004) - Semantic Web Services workshop*.

J. Broekstra *et al.* 2002. Sesame: A generic architecture for storing a querying rdf and rdf schema. In *Proc. of the first International Semantic Web Conference (ISWC2002)*, number 2342 in Lecture Notes in Computer Science, 54–68. LNCS.

M. Giordano and M. Mango Furnari. 2006. Web services dell'ontology server di remuna. Technical report, Istituto di Cibernetica E. Caianiello del C.N.R. *IC-CNR Technical Report*.

Mahmoud, Q. 2004. *Middleware for Communications*. Wiley.

Noy, N. F.; Sintek, M.; Decker, S.; Crubezy, M.; Fergerson, R. W.; and Musen, M. A. 2001. Creating semantic web contents with protege-2000. *IEEE Intelligent Systems* 16(2):60–71.

Remuna Project. 2006. ReMuNa: Rete dei Musei napoletani. http.remuna.org. Istituto di Cibernetica CNR - E. Caianiello. http//www.remuna.org.

Volz, R.; Oberle, D.; Staab, S.; and Motik, B. 2003. Kaon server - a semantic web management system. In *Alternate Track Proc. of the Twelfth International World Wide Web Conference (WWW2003)*. Budapest - Hungary: ACM.

W3C Rec. 2003. Soap version 1.2 part 0: Primer. W3C Recomendation 24 june 2003. http://www.w3c.org/TR/2004/Rec-owl-features-20040210/.

W3C Rec. 2006. Web service Description Language (WSDL) Version 2.0 Part 0: Primer. W3C Recomendation 6 january 2006. http://www.w3c.org/TR/2004/Rec-owl-features-20040210/.

W3C Work. Group. 2004. Web sevices architecture. W3C Working Group Note 11 February 2004.

Acknowledgments

We thank Salvatore Brandi who provided the required information and documentation concerning the ReMuNa Ontology Server architecture and implementation details. We also want to thank Aldo Aiello who is one of the main architect and developers of the ReMuNa ontology we used as testbed for our experiments. He gave us several hints about the ontology structure that were very useful for the interpretation of the experimental results discussed in this work.