Semantic Service Oriented Architecture

Michel Deriaz and Giovanna Di Marzo Serugendo

University of Geneva, Switzerland

e-mails: {Michel.Deriaz, Giovanna.Dimarzo} [at] cui.unige.ch

November 10, 2004

Abstract: This paper describes a new prototype of a semantic Service Oriented Architecture (SOA) called Spec Services. Instead of publishing their API through a protocol like SOAP, as Web Services do, services can register to a service manager a powerful syntactic description or even semantic description of their capabilities. The client entity will then send a syntactic or semantic description of its requirements to the service manager, which will try to find an appropriate formerly registered service and bind them together. Today our service manager can deal with two languages: regular expressions, which is probably the most powerful syntactic-only description language; Prolog, which is only semantic. Nevertheless, this implementation is made, since its beginning, with evolution in mind, i.e. to easily support integration of new additional formal languages.

Keywords: SOA, Specifications, Web Services, LuckyJ, XML

1 Introduction

Service Oriented Architectures (SOA) are more and more fashionable. The idea is to provide a specific service through the Internet and make it accessible from a program. A typical example is a travel agent's booking system that contacts the SOA of a hotel in order to book a room, then the SOA of an aircraft company in order to buy a flight ticket, and finally the SOA of a car rental service in order to book a car.

Popular and well known SOAs, like Web Services, publish an API of their capabilities. It means that a client's program needs inevitably a human interaction to find and use a service for the first time. This is of course not an insurmountable problem in the case of our travel agent, which will almost always use the same services. But if our client is a mobile user, then finding new services implies to know and understand theirs API; task that is currently not possible for a machine.

Our prototype tries to answer this issue by replacing APIs by specifications that describe in a formal way the services. Client entities write theirs requirements in a formal way as well, and the service manager bind them to the appropriate services. In the final user's point of view, services are done anonymously. The same request sent twice over the time can be fulfilled by two different services.

This paper is organized as follow. Section 2 presents the model we used to build our prototype. Section 3 presents a Java implementation of our model. Section 4 enumerates two related works, LuckyJ and Web Services, and makes some comparisons with our prototype. Section 5 presents the future works that we already studied, but that are not implemented by now. And finally, section 6 concludes this paper.

2 Model

2.1 Definitions

To facilitate the reading, we defined the technical words used through this paper:

Regex is a shortcut for regular expressions. It is probably the most powerful syntactic-only description language.

Prolog is an executable logical language that can be used to check theorems.

A **description language** is a language in which we express a service functionality or an entity request. For short we call it simply a **language**. Today our architecture can deal with two such languages: one based on regular expressions, another based on Prolog.

A **service specification** expresses a service functionality in a well defined format with XML. The semantic of the functionality is expressed with description languages.

An **entity specification** expresses an entity request in a well defined format with XML. The semantic of the request is expressed with description languages.

A **specification file**, or shorter said a **specification**, is a XML file that can be either a service specification or an entity specification. A specification file is divided onto subsections, each containing the complete description in a specific formal language.

A **service** is a server software accessible via the Internet, that provides a specific service described in its specification file.

An **entity** is a client software that sends a specification file to a service manager in order to find a service that meets its requirements.

A **service manager** is a public directory with two roles. The first is to manage specification files transmitted by services registering to it. The second is to bind entities to services according the specification files of both of them.

A **unit** can be a service, an entity or a service manager.

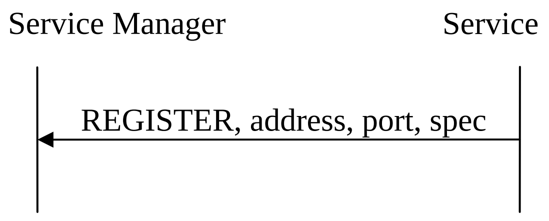
MD5 is a hashing algorithm that transforms any sequence of bytes into a fix-length (128 bits) sequence of bits. Because this function is meant to be quite robust against collisions, it is used in our prototype to track modifications in specification files. Every service is able to send a hash of its specification on request. A service manager uses this functionality to check that a service provides still the same functionality: it compares the hash computed when the service registered itself with the hash provided on request, each time that it searches an appropriate service to fulfill a client's request.

2.2 Architecture

We can summarize our architecture as follow. A service is a unit that is able to accomplish a specific task that is described in a XML file. This file, called a service specification, is then transmitted to a service manager that will register this service (name, address, specification ...). An entity is a unit that sends a specific request, called an entity specification, to the service manager. The later tries then to find an appropriate service according to the two specification files. If it finds it, it returns to the client the address of the matching service.

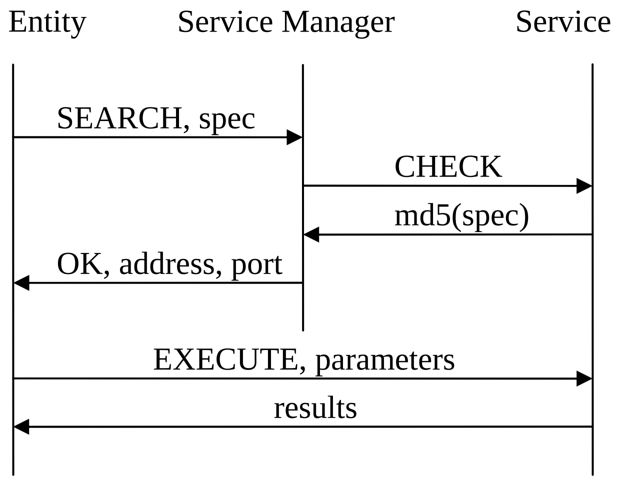
We have two main primitives: REGISTER_A_SERVICE is used by a service that registers itself to a chosen service manager; EXECUTE_A_REQUEST is used by an entity to find an appropriate service and to connect to it in order to execute the client's request.

The REGISTER_A_SERVICE main primitive wraps a smaller one, called REGISTER. Registering to a service manager is done by sending the REGISTER key word, followed by the IP address and port number, and of course the specification file that describes the service functionalities.



We consider that our architecture evolves in a ubiquitous and dynamic world. This means that services can appear and disappear at any time. Our EXECUTE_A_REQUEST main primitive is therefore composed of smaller ones. When an entity connects to the service manager in order to find a specific service, it uses first the SEARCH primitive. The service manager then checks its database in order to find a matching service. If it finds it, the service manager want to be sure that the service it still available. It then uses the CHECK primitive which sends a specific message to the service. The later answers with a fingerprint (md5 hash function) of its specification, so that the service manager can check that the service is still here and still provides the same functionalities. If not, the service manager updates its database and informs the client (KO primitive) that the service is not available anymore. And if the returned fingerprint equals the expected one, the service manager sends the service location (IP address and port number) to the client (OK primitive). The client uses then it's EXECUTE primitive, which sends to the service the different parameters of the request, and waits for an answer.

The EXECUTE_A_REQUEST main primitive can be summarized with the following diagram, supposing that the requested service is still available and still the same:



2.3 Specification files

A specification file, or shorter said, a specification, is a XML file divided into subsections. Each subsection corresponds to a particular language. Each subsection has to be self-contained: it describes completely a service or a requirement. A specification file is structured as follow:

```
<?xml version="1.0" encoding="UTF-8"?>
<specs>
  <regex active="true">
    ...
  </regex>
  <prolog active="false">
    ...
  </prolog>
</specs>
```

During an entity request, the service manager will try to match the entity specification with the service specification for all languages that are active. In our example, we see that two languages are defined (regex and prolog) but only one is active (regex). It means that only regular expressions will be taken into consideration. XML allows us to define a different structure for each language. For example in the case regex, we have four tags: <name> which denotes the name of the service, <params> which describes the expected parameters, <result> which defines the structure of the result, and <comment>, which contains optionally additional information. The following is an example of a sorting service defined by regular expressions:

```
  <regex active="true">
    <name>(?i)\w*sort\w*</name>
    <params>String\*</params>
    <result>String*</result>
    <comments />
  </regex>
```

The regular expression describing the name (`(?i)\w*sort\w*`) accepts all the words that contains the word sort, like quicksort, sorting, or sort. `(?i)` sets the matching case insensitive. The parameters are expressed be the `String\*` regular expression, which means that we expect a list of n Strings. The star indicates 0, 1, or more. If we would expect exactly three Strings (for example), we would write `String String String`. The result tag indicates that this service returns a list of Strings as well. Note that it is of course only a trivial example; the power of regular expressions allows us to express a service much more precisely.

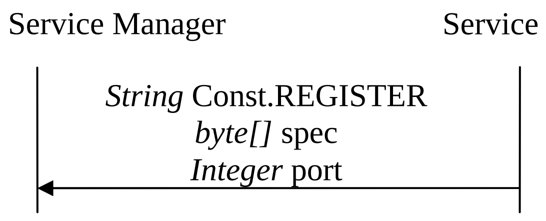
New tags can be added in the future. Another language can have a completely different structure. These two last points justify the use of such an extensible language as XML.

3 Implementation

Our prototype is implemented in Java. Communications between units are made through Java sockets, and the messages can be any kind of Java objects. Unlike Web Services that communicate only via standard protocols like HTTP, all units of our system need to be written in Java. It is not a real drawback, since the main point of our prototype is to prove the possibility of describing and requesting services through specification files.

3.1 *Registration of a service*

A service that wants to register will connect itself to the standard port number (12300) of a service manager and send it a String that defines the action (REGISTER), an array of bytes containing the specification file, and an integer defining the port number on which the service wants to be available.



3.2 *Execution of a service*

An entity that wants to access to a specific service will launch the following operations:

(1) The entity connects to the service manager (always on port 123000) and sends a String describing the operation (SEARCH) followed by the specification that describes its requirements. Note that entity specification files are similar to service specification files.

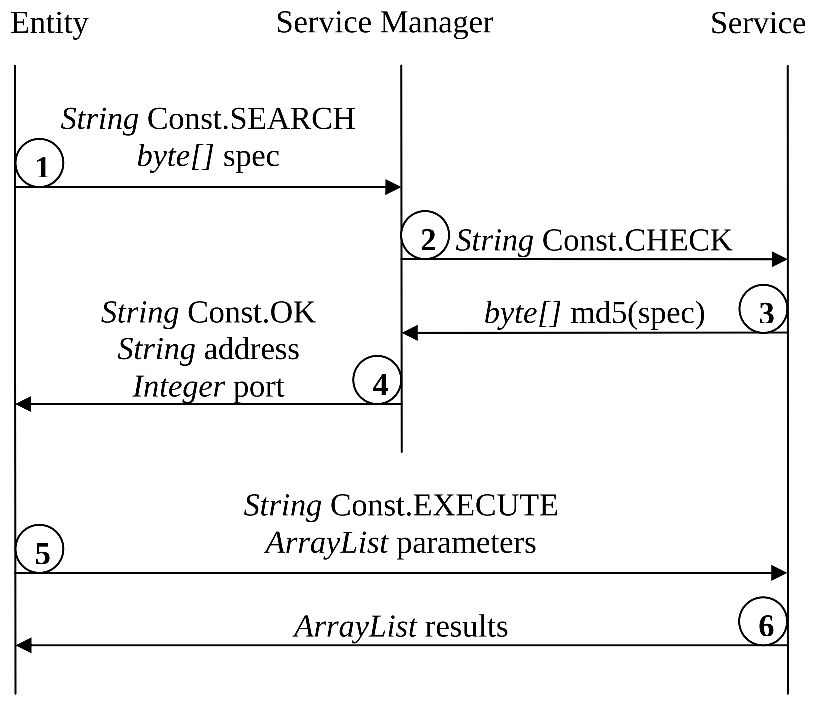
(2) The service manager searches a service specification that matches the entity specification. It then sends the CHECK message to the matching service in order to check that the functionalities are still available and still the same.

(3) The service returns an md5 hash of the specification.

(4) If the hash is the same than the one computed when the service registered, the service manager informs the entity and transmits to it the IP address and the port number of the service that matches the requirements. If the service is not available anymore or if it has changed, the service manager removes it from its database and informs the entity with a "KO" message, meaning "not OK".

(5) Once the entity knows the service that can fulfill its requirement, it connect directly to it with the EXECUTE message and sends the parameters in an ArrayList.

(6) The service executes the request and returns the result also in an ArrayList.



Parameters and results are always transmitted in ArrayLists. It is a standard which allows the transmission of any number of any kinds of objects. Specification files describe the content of these ArrayLists.

3.3 Example of a service code and of an entity code

Our architecture aims to be very simple to use. This section presents the effort required to build a sorting service and an entity that will use it.

3.3.1 SortS, a sorting service

The following code consists in a sort service:

```
import kernel.*;
import java.util.*;

public class SortS extends Service {

  public static void main(String[] args) {
    new SortS().register("localhost");
  }

  public ArrayList execute(ArrayList list) {
    Collections.sort(list);
    return list;
  }
}
```

This class extends the Service class, available in the kernel package of our architecture:

```
package kernel;

public abstract class Service implements Runnable {
  ...

  /**
   * Executes the service itself. This method has to be overridden be any
   * class that extends the Service class.
   * @param list the parameters to transmit to the service
   * @return an ArrayList that contains the response of the service
   */
  public abstract ArrayList execute(ArrayList list);

  /**
   * Registers the service to the service manager running at the specified
   * address. The location and the name of the specification file has to be
   * defined by the <code>Const.DEFAULT_SPEC_FILENAME</code> constant; by
   * default this value is set to <code>spec.xml</code> (in the current
   * directory).
   * @param address the address of the service manager
   * @return a boolean indicating whether the service registered
   * successfully
   */
  public boolean register(String address) {
    return register(address, Const.DEFAULT_SPEC_FILENAME);
  }

  /**
   * Registers the service to the service manager running at the specified
   * address, with the specified specification file.
   * @param address the address of the service manager
   * @param specFile the path to the specification file
   * @return a boolean indicating whether the service registered
   * successfully
   */
  public boolean register(String address, String specFile) {
    ...
  }

  ...
}
```

The `main(...)` method is used to register the service at a service manager available at `localhost`, from a console. By default, the specification file must be stored in the same directory and be named spec.xml.

Every service extends the `Service` class and has to redefine the abstract `execute(...)` method, which is the only method that will be indirectly invoked by a client. The parameters are transmitted in an ArrayList and the result is returned in an ArrayList as well.

3.3.2 SortE, a sorting entity

An entity that wants to use the sorting service can do it with a single line of code:

```
import kernel.*;

public class SortE extends JFrame implements ActionListener {
  ...

  private void sort() {
    ...
    result = Entity.execute(SM_ADDRESS, parameters);
    ...
  }
}
```

`result` and `parameters` are ArrayLists and `SM_ADDRESS` is a String defining the service manager address. We assume that the entity specification file is available in the same directory and is named spec.xml. To execute a request, an entity uses the Entity class, available in the kernel package of our architecture:

```
package kernel;

public class Entity {
  ...

  /**
   * Executes a request. This method sends its request to the service
   * manager, which is responsible to find a matching service. If a
   * matching service is found, the parameters are sent to the
   * <code>execute</code> method of this service. The location and the name
   * of the specification file has to be defined by the
   * <code>Const.DEFAULT_SPEC_FILENAME</code> constant; by default this
   * value is set to <code>spec.xml</code> (in the current directory).
   * @param smAddress the IP address of the service manager
   * @param list the parameters to transmit to the service
   * @return an ArrayList that contains the response of the service, or
   * null if the service can not be found
   */
  public static ArrayList execute(String smAddress, ArrayList list) {
    return execute(smAddress, Const.DEFAULT_SPEC_FILENAME, list);
  }

  /**
   * Executes the request described in the specified specification file.
   * This method sends its request to the service manager, which is
   * responsible to find a matching service. If a matching service is
   * found, the parameters are sent to the <code>execute</code> method of
   * this service.
   * @param smAddress the IP address of the service manager
   * @param specFile the path to the specification file
   * @param list the parameters to transmit to the service
   * @return an ArrayList that contains the response of the service, or
   * null if the service can not be found
   */
  public static ArrayList execute(String smAddress, String specFile,
ArrayList list) {
    ...
  }

  ...
}
```

4 Related work

This section describes LuckyJ and Web Service, two SOA close to our architecture.

4.1 LuckyJ

Our architecture is derived from LuckyJ, a platform allowing run-time evolution of applications. This is particularly useful for two kinds of applications: those that manage safety critical systems, such as nuclear power plants, and those that offer 24/7 services, like mail accounts servers.

The different services register to the service manager by describing their functionalities using well defined keywords. The client entities can then contact the service manager and transmit the request of the desired service using shared keywords. The service manager is then responsible to find the most appropriate service, to transmit him the request, and finally to communicate the answer to the client.

The interesting point is that these operations are done asynchronously. It is therefore possible to dynamically add a new version of a service. During a short lap of time we will have the old version of the service that finishes to serve requests made before the start of the update process, while newly made requests will be answered by the new version of the service. The old version of the service can be removed as soon as all its pending requests are answered.

We notice three main differences between LuckyJ and our prototype. Firstly we use specification files to describe or request a service. Expressed in different formal languages, it is possible to let machines to find themselves an appropriate service, without any human intervention. This is not possible in LuckyJ, where the language is only sufficient to express the API. Secondly, parameters and results are transmitted between entities and services into ArrayLists, which can contain any kind of objects. In LuckyJ communication are restricted to Java primitive types. Finally we notice that our implementation allows us to transform any existing program into a service; we just need to call the appropriate method from the `execute` method in the `Service` class.

4.2 Web Services

Web Services is probably the most popular Service Oriented Architecture (SOA). A Web service is a logical component of an organization available on the internet. To access it, widespread protocols like HTTP are used. The difference between Web Services and other formerly used technologies, like DCOM and CORBA, is that Web Services are articulated around XML. Everything, from data exchanges to protocols, are made in XML, eliminating therefore all links with a software or materiel architecture. This technology allows as well exchanges of documents as remote procedure calls, in a synchronous or in an asynchronous manner.

This offer is available through a set of standards protocols: SOAP (Simple Object Access Protocol) supports exchanges of documents and Remote Procedure Calls (RPC). HTTP, FTP and SMTP are used to transport the information. WDSL (Web Service Description Language) describes in a standard way the public available services. UDDI (Universal Description, Discovery and Integration) is used to find a specific service in a directory.

Unlike Spec Services, Web Services can be written in different languages. This is clearly a huge advantage for such a widespread system. Nevertheless the main goal of our prototype was to communicate through specification files and not to deal with interoperability issues. Therefore we focused our attention to the specification files. Web Services describe their functionalities by publishing their API, task that requires human intervention. Spec Services use formal languages, understandable by machines.

5 Future works

The first improvement we would like to do is to implement a better logging system. Today, debugging is a very difficult task that becomes quite impossible when the bug is not detected immediately. We would like to log the services activities in files so that we can trace back easily in case of a crash or misbehavior of the system.

The second improvement would like to add new languages. As mentioned earlier in this document, this prototype was build with evolution in mind. XML allows us to easily extend the languages we use and of course to add new ones. The language based on regular expressions is very powerful, more expressive than the one used in Web Services, but is only syntactic. Human participation, even if strongly reduced, is therefore still necessarily to express the specification in this language. The language based on Prolog solves partly the former problem in the sense that it is a semantic one. We can imagine that machines are able to build a specification file according theirs needs, but we saw also that writing a specification file in Prolog is far from easy. And we think that a common ontology is essential, at least for a particular set of problems. We are therefore looking for other languages like Simplified Common Logic or Jena. These two will perhaps bring us a good merge between semantics and ontology.

The third improvement consists in transforming our implementation into a local based service. The idea is to make a service available only if certain requirements are met. For example a printing service can be available only if the entity is physically close to the printer and only at certain hours of a day. The University of Geneva deployed a geo-localized system that provides course material to mobile computers that are in specific areas. Technically, the system computes the position of the user according to the strength of different Wi-Fi signals received from different well known placed antennas. We think that we could use this technology in our prototype and integrate position information in our specification files.

6 Conclusion

In this paper, we have presented a semantic SOA (Service Oriented Architecture). Unlike most SOA that publish an API, Spec Services is a prototype that allows publishing semantic description of functional behavior (no API). Today our specification files can contains two different languages, the first based on regular expressions, and the second based on Prolog. But we saw that theses files, written in XML, allow us to add new languages or extend existing ones very easily. We presented also a Java implementation of a sorting service, and the single line of code to insert in an existing program in order to access this service.

7 Bibliography

[1] M. Oriol, G. Di Marzo Serugendo, "*A Disconnected Service Architecture for Unanticipated Run-time Evolution of Code*", IEE Proceedings-Software, Special Issue on Unanticipated Software Evolution, Susan Eisenbach (Ed), 2004.

[2] Services Web, Login hors-série, september/octobre 2004, pages 8-20.

[3] W3 Schools. http://www.w3schools.com/

[4] Loosely coupled. http://looselycoupled.com/glossary/