A thesis proposition: Service Oriented Computing in a P2P architecture

POSITION PAPER

June 7, 2005

Michel Deriaz

University of Geneva, Switzerland
{first_name.last_name} [at] cui.unige.ch

Abstract. This position paper is an attempt to describe my future thesis work. It shortly describes the work that has already been done, and then suggests different research directions. Currently we have an architecture in which services register with a service manager and entities queries the service manager in order to find a service that is able to fulfill a specific need. Unlike traditional Web services that use APIs to communicate, our architecture uses specification files allowing powerful syntactic and even semantic descriptions of services or requests. However, in client-server architectures, the clients act independently from the others and this prevents collaboration between them. This thesis proposition suggests a decentralized system, highly dynamic, accessible by humans and machines, and proposes to study issues like semantic interoperability, trust, and rewarding of good peers.

1 Introduction

The Web has been designed for humans. Common syntax rules (like HTML) allow pages to be displayed correctly in browsers. Common semantic rules (like the language that is used) allow humans to understand the content of a page. But this semantics is not included into the pages, and it is therefore difficult for a machine to "understand" a Web page. There are some agents that are designed to crawl the Web and to search specific information. An example is the shopbot, a program that searches the best price for a particular article. It retrieves pages that contain the wanted item, then searches a "$" sign in the neighborhood of the name of the article, and finally shows the best price to the user. Currently such robots are not able to do much more. They are not able to interpret correctly the content of Web pages. It is time for the well-known *syntactic* Web to turn into the *semantic* Web, allowing humans as well as machines to access it.

Web services can be accessed by machines using standard protocols like SOAP, WSDL and UDDI, but these protocols contain no semantics. It means that it must imperatively be a human that binds a requester to a specific service. It is not a problem in the case of a travel agent that uses Web services of aircraft companies to book flights or Web services of hotels to book rooms, because these services are always the same. But it is a problem for a user that just needs once a particular service, like converting a PS file into a PDF file. Traditional Web services are designed more for businesses than for end-users.

There are some Web sites that offer “simple” services, like converting a PS file into a PDF file. An example can be found at [PS2PDF]. Such services are very useful. Instead of searching, installing, using, uninstalling different programs till you find the one that fulfils your need, you simply rely on someone else to provide you the right service. Theses services are very simple to use, but are meant for humans. Often advertisement is showed before you access the service.

We can understand that “simple” service providers like the PS to PDF service described above are not designed to be anonymously accessed by machines. The author wants either to make money by adding ads, or at least to provide information that he wants to be read by human visitors, like information about himself or about his company. We observe the same thing in homepages, where people can “sell” themselves, in services like Google, which are sponsored by commercial links, or in P2P systems, where people accept to share file in order to access other files. And even if the service would have been designed to be accessed by a machine, there is still the problem of finding automatically a specific service and to *trust* it.

The dream would be to write a request like “convert PS (aFile.ps) into PDF”, and have a system that will find the best (more reliable, most trustworthy...) service currently available, and that simply returns the wanted result. It is a dream for the user, but it is also the future for the researcher. The Proposition section will analyze more deeply what challenges have to be faced and gives some research directions.

2 Achieved work

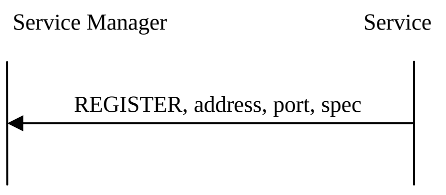
This section describes the work that has been done for a past FNRS project.

2.1 Architecture

We can summarize our current architecture as follows. A service is a unit that is able to accomplish a specific task that is described into a XML file. This file, called a service specification, is then transmitted to a service manager that will store this information with the service's address. An entity is a unit that sends a specific request, called an entity specification, to the service manager. The latter 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.

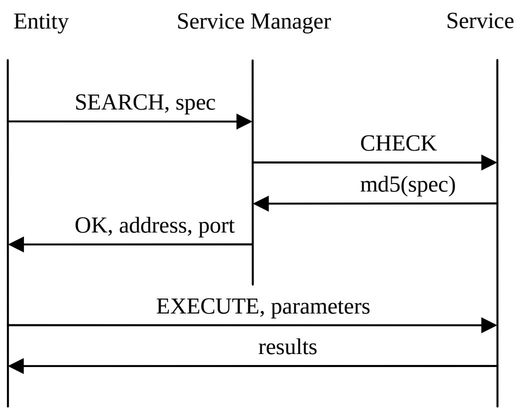
We have two main primitives: REGISTER_A_SERVICE, used by a service that registers itself to a chosen service manager; EXECUTE_A_REQUEST, used by an entity to find an appropriate service and to connect to it in order to execute the client's request.

The REGISTER_A_SERVICE main primitive wraps a smaller one, called REGISTER. Registering to a service manager is done by sending the REGISTER keyword, 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, provided that the requested service is still available and still the same:



2.2 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 Proposition

The Introduction section revealed already the directions to follow for our future researches. This section will give some precisions for each topic.

3.1 From file-sharing to service-sharing

In the same way as P2P programs like Kazaa or Shareaza allow participants to share files, we propose a P2P system in which participants share services. A complete decentralisation seems important to us. Of course, this avoids the single point of failure problem. But this is not the main point, since cluster techniques can partly solve the problem. The main idea of decentralization is improving the collaboration between the different peers in order to find dynamically new services. That is, in client-server interactions, the different clients do not collaborate with each other. There is no evolution. Without any human interaction, the same requester will always try to contact the same service, even if the service is not available or if a better service appeared on the network. To our view of a dynamic environment, services could appear and disappear all the time, and collaborate in order to provide new services. For instance, a service able to convert PS files into PDF can collaborate with a service able to convert PDF files into Word documents, and therefore be able to solve a request that consists in transforming a PS file directly into a Word document.

3.2 Semantic interoperability

Our current architecture communicates through specification files that support two languages: one based on regular expression; and a second based on Prolog. 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 logical one. We can imagine that machines are able to build a specification file according to 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 [SCL] or Jena [jena]. These two will perhaps bring us a good merge between semantics and ontology, and allow our services to collaborate autonomously.

3.3 Trust

The SECURE project [secure] proposes an interesting approach for the notion of trust. Like by human communities, trust in this project evolves dynamically according to new experiences or recommendation from other participants. Some other work has also been done on handling trust in P2P systems. A good example is the EigenTrust [eigentrust] algorithm in which each peer has a set of mother peers that are responsible to compute and maintain its trust value. But such algorithms are often designed with file-sharing in mind. Then, a single trust value for each peer is sufficient for a client to find the more reliable peer, i.e. the peer that will probably not send a corrupted file hosting a virus or a Trojan.

Handling trust in a service-sharing system is actually quite different. One, trust must be handled in both directions. A server can provide a corrupted service, but a client can also send corrupted data in order to bring the service down. Two, we need a trust value for the "honesty" of the server, but also for the "quality of the service". In a file-sharing system, if you download a song that you do not like, it does not mean that it is poor quality. The only point you are interested in is to not download corrupted files. In a service-sharing system we want of course not have to undergo a Trojan attack, but we want also be able to select the best service, according global reputation, recommendations given by friends, and also according to former experiences that we have had with this peer.

3.4 Rewarding of good peers

What is the interest of publishing for free a service that will be anonymously accessed by a machine? Nothing. The same happens in file-sharing systems; people are usually more interested in getting files than in providing files to the community. It is why file-sharing programs reward good peers (i.e. peers that provide files) by giving them better access to other resources. The challenge here is to be able to measure the global reputation of a service in order to reward the corresponding peer. This can be done according to the different trust values. Of course, security measures must prevent peers from modifying their own trust value and also prevent associations of malevolent peers trying to subvert the system by rating each others in a way that they get a very high global trust value.

3.5 Decentralized architecture

Building a complete P2P system is clearly out of reach for a single researcher and also clearly inefficient. The aim is not to re-invent existing algorithms, but to focus on specific topics, like the trust. Therefore we are not going to continue the development of our current centralized architecture but rather use an existing decentralized one. A possible candidate is provided by the JXTA project [jxta].

4 Related work

This section describes LuckyJ and Web services, two SOAs close to our current architecture.

4.1 LuckyJ

Our current architecture is derived from LuckyJ [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 [WebServices] 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. WSDL (Web Services 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 our current architecture, 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 (this will be different in future work). Therefore we focused our attention to the specification files. Web services describe their functionalities by publishing a structured syntactic description including their API, task that requires human intervention. Our architecture uses formal languages that semantically describe the functional behavior, and these are understandable by machines.

5 Conclusion

This paper presented a thesis proposition. The idea is to share services in a peer-to-peer network in the same way as we share files in systems like Kazaa or Shareaza. To realize this, our attention will be focused on three main domains. One, the semantic interoperability. In order to provide services that can be accessed by humans and machines, and that collaborate with each other in order to provide new services, a common language for expressing semantic information is necessary. Two, the trust. A reputation system should prevent malicious services or malicious clients from subverting the system, and exclude them from the network. Three, the rewarding of good peers. To encourage peers to publish high quality services, a reward like giving better access to resources should be given to reputable peers.

It is clear that it is not a precise description of what is going to be done. It must instead be seen as a general field of interest of the author. Nevertheless, the trust topic seems to be particularly interesting in this proposal, and future work on it should better reflect the challenges that are hidden by this key concept.

6 Bibliography

- [PS2PDF] PS to PDF website, http://www.ps2pdf.com/convert/convert.htm.
- [SCL] David Frankel and al. "Simple Common Logic: A Constraint Language for the ODM"
- [jena] Brian McBride. "Jena: Implementing the RDF Model and Syntax Specification"
- [secure] V. Cahill and al. "Using trust for secure collaboration in uncertain environments." IEEE Pervasive Computing Magazine, special issue Dealing with Uncertainty, 2(3):52–61, 2003.
- [eigentrust] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. "The Eigen-Trust Algorithm for Reputation Management in P2P Networks". 2003.
- [jxta] JXTA website, http://www.jxta.org/
- [LuckyJ] 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.
- [WebServices] Services Web, Login hors-série, september/octobre 2004, pages 8-20.