Towards Trusted Semantic Service Computing

Michel Deriaz

University of Geneva, Switzerland

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 functional 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 to bind them together. Today our service manager can deal with two languages: regular expressions, which is probably the most powerful syntactic-only description language; and Prolog, which is purely semantic. This implementation is made, since its beginning, with evolution in mind, i.e. to easily support integration of new additional formal languages and to provide support for non-functional properties of services. This paper proposes also a trust-based extension of our architecture in order to deploy effectively these self-describing services in an uncertain environment.

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 Web Service of a hotel in order to book a room, then the Web Service of an aircraft company in order to buy a flight ticket, and finally the Web Service of a car rental service in order to book a car.

Popular and well known SOAs, like the architecture supporting Web Services based on SOAP, UDDI and WSDL, publish an API of their capabilities. A client entity has therefore to choose the different services it wants to access at design 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, services need therefore to be chosen at runtime and are not necessarily identified already at design time.

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 binds them to the appropriate services. From the user's point of view, services are accomplished anonymously. Indeed, the same request sent twice over the time can be fulfilled by two different services.

2 A semantic SOA

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 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. In case of several matching services, a non-deterministic choice is performed.

3 Implementation

Our implementation is written in Java and uses sockets to transmit data (specification files, parameters, results) between the different entities. The use of this prototype is very simple. The unit that wants to act as a service manager just has to launch the *ServiceManager* class. This class, which acts as a server, waits for client connections.

To provide a new service, a programmer will have to extend the *Service* class. This class defines two primitives for registering and executing a service, through the *register* and the abstract *execute* methods respectively. The *register* method is used for actually registering the service to the service manager. The programmer redefines the *execute* method, which is called each time the service is required. The *Service* class then serves as a wrapper of a specific service. It becomes then very easy to transform an existing functionality into a service that can be registered in the service manager.

A program that wants to use a specific service just has to call the static *Entity.execute* method and to give as parameter the name of the specification file, the address of the service manager, and the parameters. The system is then responsible to find a matching service, to execute it and to return the result to the requester.

4 Towards trusted services

Web services are currently mainly thought for business. Our vision is that everybody should be able to use remote services. Finding or providing services on the Internet should be as easy as finding or publishing a web page; or as easy as localizing or making available some files in a peer-to-peer file-sharing system. Let's take the example of a user who wants to convert a *PS* file into a *PDF* file. The usual way consists in searching a program on the Internet, downloading it, installing it, and perhaps paying for it. Our vision is that such kind of services should be available online. The user describes what he wants to do and the system does it. There are already a few services like that to convert files [2], but not without some drawbacks. First, they are accompanied with advertisement and commercial links. This means that theses services are meant to be used by humans and there is no interest for the provider to make them accessible directly by programs or even in an anonymous manner. Second, they are very seldom and can disappear anytime. And third, how can we trust such a service? Is the original file that the client sends really deleted after processing? Can the resulting file host a Trojan or a virus?

The following scenario should be supported by our future architecture. A user who wants to convert a file submits its request written in a formal way to the system. Basically, in this case, the specification file precises that the operator is *convert*, that the input is a *PS* file and that the output is a *PDF* file. The system then returns a list containing all the matching services and their trust value, indicating how reliable they are. The user, which can be a human or a machine, chooses one or several services to process its request and rates them according to the quality of the result.

We are currently working on the possibility of adding non-functional properties such as trust and rewarding information in the specification. The trust value indicates how reliable a peer is, and is continuously modified according to the ratings made by the client entities. A rewarding system will give to good peers a better access to other services. A good peer is one that has a high trust value, or a good availability, or any other criteria that can be combined together. The rewarding system seems very important to us. Web pages are read by humans, so they can be used for advertisement. Peer-to-peer systems like Kazaa reward good peers by giving better access to those that share lots of files. We are currently studying different approaches that could motivate people to publish high quality services, that will stay online and possibly autonomously interoperate with each other.

5 Discussion

In addition to combining our current implemented architecture with a trust and rewarding system, we envisage to make some other improvements to our architecture.

The first improvement could be to reconsider our architecture, which is currently centralized. The main advantage of a centralized architecture is simplicity. Trust values can be stored by the service manager, in a similar way than eBay [1]. A simple rewarding system would be to give higher priority for accessing services to peers that have a high trust value. But a centralized architecture has also some drawbacks, essentially scalability problems and the fact that there is a single point of failure. Therefore we consider also the possibility of decentralizing our system. The trust value of each peer is then hold by the peer itself, and varies according to the ratings of other peers. Algorithms like EigenTrust [5] can be used in order to prevent malicious peer modifying their own trust value or cooperating with other malicious peers in order to subvert the system. Algorithms like P-Grid [4, 6], designed with c2c commerce in view, can be used to localize a specific service. Another solution would be to use the Google API [3] to build a powerful service localisation service.

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 to their 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 final 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 contain 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 saw also in section 5 that the specification files can host other kind of information, like trust values or spatial coordinates. These non-functional properties are useful for efficient deployment of such self-describing services.

References

1. eBay website, http://www.ebay.com/.
2. PS to PDF website, http://www.ps2pdf.com/convert/convert.htm.
3. The Google API website, http://www.google.com/apis/.
4. Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Beyond "web of trust": Enabling P2P E-commerce. 2003.

5. Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. 2003.
6. Karl Aberer. P-Grid: A self-organizing access structure for P2P information systems. *Lecture Notes in Computer Science*, 2172, 2001.