Integrating Knowledge-Level Agents in the (Semantic) Web: An Agent-based Open Service Architecture

Nicola Dragoni and **Mauro Gaspari** and **Davide Guidi**
Dipartimento di Scienze dell'Informazione
University of Bologna
via Mura Anteo Zamboni 7
40127 Bologna, ITALY

Abstract

In this paper we present an Agent-based Open Service Architecture (OSA) which integrates geographically distributed agents in the Web. Agents can be realized with traditional AI techniques, but they also provide a set of Web Services to the outside world which constitute their capabilities. The architecture extends the Web with a facilitator level providing agents' specific support. Moreover it provides primitives for Web Services invocation and inter-agent communication based on Agent Communication Languages. We present the design of the architecture and an implementation which extends a common Web Server based on Apache/Tomcat platform.

Introduction

Artificial intelligence is playing an increasingly important role in the Internet which, under the Semantic Web activity, will provide more and more reusable formalized descriptions of data such as ontologies. Although there are many small-scale examples of implemented systems which use this formalized knowledge to achieve intelligent behaviors, for example personal assistants which collect and organize information, several issues concerning the full exploitation of traditional Artificial Intelligence technologies such as Logic and Expert Systems in the Semantic Web remain open.

Agents have been recognized as one of the main building blocks of the Semantic Web infrastructure, but their role is still not completely depicted. For example, their relationship with more standard components such as Web Servers and clients, is still not clear. Most of the examples of agents in the Web are **User Agents** which provide intelligent support and advanced services to users. These agents can be realized with any programming language or traditional AI technologies provided that they are able to access the RDF-based triple-space.

However, whenever an agent provides a set of complex problem solving capabilities, it becomes reasonable to reuse its skill in realizing other intelligent applications. This suggests another possible role for agents: to enhance the functionalities of servers. **Worker Agents**, providing complex problem solving capabilities with respect to a given application domain, can be published and shared on the Web, for example by means of a set of well defined Web Services and an associated ontology. Sophisticated knowledge intensive behaviours can be made available on the Web by developing and publishing libraries of intelligent agents able to interact among them, rather than adding more and more ontologies and semantic annotated data (McIlraith, Son, & Zeng 2001; McIlraith & Martin 2003).

A significant challenge to realize this vision is the design of distributed reasoning infrastructures tightly integrated with current Internet components and technologies that would allow agents to be exploited in the large. While new standards are emerging such as the Ontology Web Language (OWL) (W3C Web-Ontology Working Group 10 February 2004), and the community is currently addressing many central issues as the design of a Web Service Modelling Ontology (WSMO) (Davies, Fensel, & Richardson 2004) and of an Ontology Web Language for Services (OWL-S) (Martin *et al.* 2004), there is still not a widely accepted architecture for integrating agents in a distributed reasoning infrastructure on the Web.

In this paper we present an Agent-based Open Service Architecture (OSA) which integrates geographically distributed agents in the Web supporting both User and Worker agents in a uniform framework. An Open Service Architecture is a software infrastructure that makes a dynamic set of services available to users and agents over the Web. An OSA defines standard mechanisms for creating, naming, discovering and integrating such Web Services. Agents are the main building block of our architecture. They run on **Web Agent Servers**, *i.e.*, Web Servers extended with a facilitator level providing agents' specific support and standard primitives for inter-agent communication based on Agent Communication Languages (ACL). Agents provide a set of Web Services to the outside world which constitute their capabilities. Each agent is able to retrieve, execute and compose Web Services published by other agents in order to achieve its goals or provide more sophisticated services. We present the design of the architecture and an implementation which extends the Apache/Tomcat platform, where agents can be encoded in any programming language including standard AI languages or knowledge representation languages.

An Agent-based Open Service Architecture

A general view of our architecture is depicted in Figure 1. The architecture supports both User Agents and Worker Agents on Web Agent Servers.

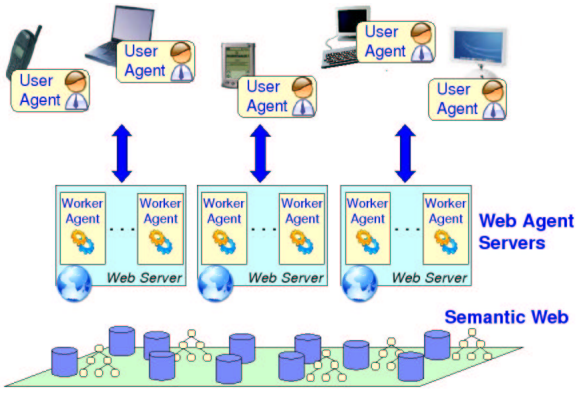

Figure 1: Our Agent-based Open Service Architecture.

User Agents act as interfaces between users and the Web, providing a support for discovering and invoking Web Services. Users can configure their User Agents with their preferences. They can be always connected to the OSA or they can disconnect themselves when their users want. A *Web Agent Server* extends a Web Server with agents' functionalities and is the main building block of our OSA. Web Agent Servers are geographically distributed (as Web Servers are) and provide a set of Web Services to the outside world which constitute their capabilities. This set can dynamically change because of new publication of Web Services and/or modification of the existing ones. A Web Agent Server supports both User and Worker Agents providing standard primitives for inter-agent communication based on Agent Communication Languages (ACL). These primitives are realized on top of standard XML based technologies subsuming Web Service invocation-response patterns.

Worker Agents are able to retrieve, execute and compose Web Services to provide more sophisticated services. Contrary to User Agents (which can disconnect themselves from the OSA), Worker Agents are always connected to the OSA and act like a daemon process.

Agents can be realized in any programming language including AI languages or knowledge representation languages.

Fundamental issues in designing Open Service Architectures

Several issues arise in designing an OSA which enables agents to be fully integrated in the Web.

A first issue is related to the *geographically distributed nature* of the nodes of the OSA (User Agents and Web Agent Servers) which are subject to possible **failures or network partitions**. Most of the available knowledge based systems on the Web (such as the IRS-III (Domingue *et al.* 2004)) concentrate their efforts on knowledge modeling issues and there is still a relatively little attention to issues that derive from the distributed nature of these architectures. For example, the IRS-III is a centralized system where all the available services should be published. If the IRS-III server is not reachable there is no way to use the semantic Web Services it provides, even if their standard instances (without the semantic annotations) are reachable. We claim that an OSA should be based on distributed reasoning services, and as a consequence of this should be designed to support distributed reasoning protocols which function in the presence of failures or network partitions. Moreover each provider of Web Services (such as our Web Agent Servers) should support the publication of Web Services which are developed locally in a given site and should provide a protocol to make available these capabilities to other nodes.

Another important issue related to the distribution of the reasoning service is **to establish a standard interface** for these agents. This interface should allow user to invoke Web Services using high level mechanisms, but should also support agent-to-agent interaction. We argue that current standards for invoking Web Services which are based on SOAP over HTTP implementing simple invocation-response patterns are not adequate for this purpose. On the other hand we claim that Agent Communication Languages (ACLs) can be successfully used for this task, especially if they provide support for fault tolerant communication primitives as suggested in (Dragoni & Gaspari 2003; 2004). In our OSA we provide ACL primitives on the top of standard XML and SOAP based technologies. This interface is managed by Web Agent Servers and constitutes a facilitator level.

Main features of our Agent-based OSA

In the following we outline the main features which have guided us in the design of our Agent-based OSA. Then in the next Section we will show in details how the OSA is realized.

Knowledge-level Agents and Communication. We assume *knowledge level* agents (Gaspari 1998), that is, they should concern with the use, request and supply of knowledge without dealing with symbol level issues. Agents access services and communicate each other using a fault tolerant knowledge-level Agent Communication Language (ACL) which provides one-to-one and one-to-many primitives (Dragoni & Gaspari 2003; 2004). The primitives of this ACL can be divided into four categories as shown in Table 1. We assume an asynchronous communication and a reliable message passing, *i.e.*, whenever a message is sent it must be eventually received by the target agent (thus we do not handle communication failures, such as send or receive omission).

Open Architecture. Each Web Agent Server can provide one ore more Web Services to other agents. This set of Web Services can dynamically change because of the creation of new services on the Web Agent Server or the modification of existing ones. Moreover, our OSA allows new Web Agent Servers to dynamically connect themselves to the OSA, providing new Web Services to other agents.

Standard conversation primitives insert($\hat{a}, \hat{b}$, p), ask-one($\hat{a}, \hat{b}$, p), tell($\hat{a}, \hat{b}$, p)
Support for anonymous interaction (one-to-many primitives) ask-everybody($\hat{b}$, p)
Support for anonymous interaction (auxilliary predicates) all-answers(p), register($\hat{b}$, p), unregister($\hat{b}$, p)
Support for creation and termination of agents create($\hat{b}$, w), clone($\hat{b}$), bye

Table 1: Primitives of our ACL. We use $\hat{a}$ and $\hat{b}$ as agent names (recipient and sender respectively), L as a set of agents names, p as the content of a message (expressed as a proposition) and w as a generic knowledge base of an agent.

Fault Tolerant Communication. We assume agents are subject to possible crash failures. A crashed agent stops prematurely and does nothing from that point on. Before stopping, however, it behaves correctly[1]. An OSA should deal with such failures preventing agents to wait answers from crashed agents forever. As we show in the next Section, our OSA is integrated with a *distributed failure detector service* (Chandra & Toueg 1996) which allows to add fault tolerance to the primitives of the ACL.

Architecture of the Agent-based OSA

To describe in details the architecture of our Agent-based OSA we first refine the general architecture discussed in the previous Section (Figure 1) and then we focus on a single node of the architecture (the Web Agent Server) describing its main components.

The architecture we propose is presented in Figure 2. It extends a Web Server with two main components: a **facilitator service** and a set of **Agents**. Agents can be both Worker Agents, which always runs on the server, or User Agents which, if necessary, can be downloaded and activated in mobile devices.

Worker Agents implement the Web Services which the Web Agent Servers provide to User Agents and other Worker Agents. They are always active and are able to perform complex problem solving operations interacting with other Agents. Therefore each Worker Agent is reactive (it reacts to requests of services) but it can also have a proactive function to solve complex tasks. An example of Worker Agent can be a simple News Service which forwards to connected User Agents some news according to User Agents preferences. Or it can implement a more

[1]Note that more severe types of failures can occur in these architectures. A well known classification of process failures in distributed systems can be found in (Mullender 1993).

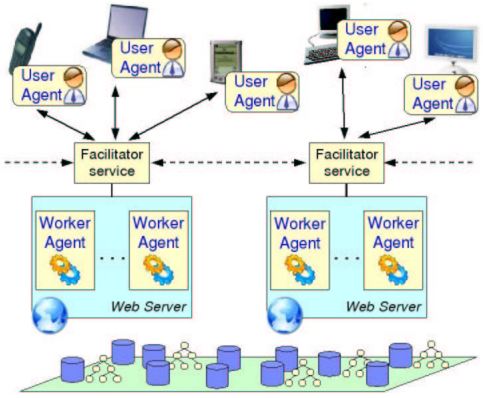

Figure 2: Architecture of the Agent-based OSA.

complex service which requires the interaction with other Worker Agents, such as a *Book-Travel-Agent* which is able to organize a travel according to user's preferences. Agents operate at the knowledge-level (Gaspari 1998) and can be coded in any language provided that it supports the communication primitives of our fault tolerant ACL[2]. Each agent can register its competences (that is, the services it provides) in the associated facilitator service using the ACL primitive *register*. To undo this operation the agent can use the ACL primitive *unregister*. For the sake of simplicity here we assume that agents' capabilities are expressed as a set of propositions (for example, propositions based on a shared ontology), one for each service. However in the last Section of the paper we will show how it is easy to overcome this assumption extending our architecture with more complex description of agents' capabilities.

The facilitator service allows agents (User and Worker Agents) to communicate each others at the knowledge-level by means of the ACL primitives (Table 1). Moreover it provides fault tolerant support to inter-agent communication.

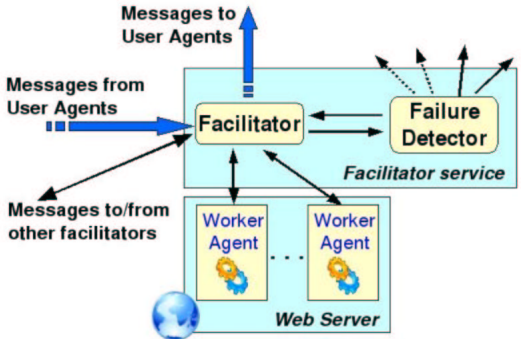

Figure 3: Architectural view of a single Web Agent Server.

[2]From a practical point of view this means that the implementation language should be able to communicate over TCP sockets with other devices. Communication primitives are then realized forwarding adequate SOAP messages to the facilitator component.

The facilitator service is realized by means of two components (Figure 3):

- **Facilitator**: this component is associated to a Web Server (and therefore to each agent of the Web Server) and it manages the communication with other agents. The facilitator provides anonymous (contents based) facilities to retrieve and request services. It registers the competences of associated agents and forwards these competences to other facilitators executing a distributed protocol. Moreover, it is able to perform a generic matching operation that matches a request (expressed as a proposition) with agents'capabilities.
- **Failure detector**: this component implements a distributed failure detector mechanism (Chandra & Toueg 1996). It monitors all the agents associated with the facilitator service and communicates to the facilitator those that it currently suspects to have crashed. The failure detector is *unreliable*, meaning that it can make mistakes suspecting agents which are not crashed. This unreliability comes from the impossibility results for asynchronous systems to determining whether a process has actually crashed or is only "very slow".

A complete specification of the facilitator and failure detector components is out of the scope of this paper. Readers interested in a complete and formal specification of these components can found it in (Dragoni & Gaspari 2003; 2004).

Discussion

In principle several Worker Agents with the associated facilitator service can be accommodated in a single Web Server. The facilitator and the failure detector are though to be tightly integrated with the Web Server, while the aim of Worker Agents is to provide independent components. Leaving all technical communication details to facilitator means that all the computational resources of the processor running the Worker Agent could be used for itself, enabling these agents to implement simple Web Services or more complex reasoning services. This feature of our architecture is valid also for User Agents, enabling their implementation also on small (e.g. handheld) devices. In this case only the display is sent to the mobile device while the agents runs on the server.

Although all the emerging standards for the Semantic Web use formalisms based on XML, we have chosen to support the integration of any knowledge representation language in our agents. Indeed, in our proposal all the agents have no constraints on the implementation language or knowledge representation formalisms they adopt, but they react to a well defined protocol based on the standard primitives of our ACL. On the contrary, most of AI systems are still being developed using specific AI technologies and languages which usually are not compliant with Web standards, they usually provide powerful engines and a rich set of libraries. From a practical point of view it is not feasible to translate all these technologies in XML based formalism. Therefore we claim that an adequate mechanism should be designed for integrating agents, as cgi and more recently servlets have been developed to access standard application.

In our opinion, an advantage of our approach is that it successfully integrates different issues, such as high-level inter-agent communication and fault tolerance, in an OSA maintaining a clean design of the architecture and a knowledge-level characterization. Despite our ACL only provides a small set of primitives, they can be successfully exploited in several well known scenarios for Web Services usages, as those described in (He, Haas, & Orchard 2004) by the W3C Working Group, to come up to original solutions. For example the *tell* primitive is an example of a *fire-and-forget to a single receiver* scenario, while *ask-one* and *tell* can realize a general asynchronous messaging scenario or more complex conversational message exchanges (as the W3C usage scenarios *request/response* and *request with acknowledgment*). Another important feature of our ACL is that it supports an *anonymous interaction protocol* which has been developed for OSAs and which is integrated with standard agent-to-agent primitives (Table 1). This allows an agent to perform an asynchronous request of services based on contents without knowing the name of the recipient agents (ACL primitive *ask-everybody*). If required they can also continue the cooperation using agent-to-agent communication primitives. In terms of W3C Web Services usage scenarios, this is a case of *registry based discovery* where the registry is distributed in all the facilitators. Also the *third party intermediary* W3C usage scenario can be easily realized by means of our ACL primitives *ask-everybody*, *ask-one* and *tell*. Moreover, the discovery facility is then integrated with fault tolerant primitives to manage multiple (non serialized) asynchronous responses. In (Dragoni & Gaspari 2004) these primitives are formally specified and a formal verification of their fault tolerant properties is also provided.

Another feature that our ACL provides to OSAs is a support for agent creation and cloning. Thus new Worker Agents which implement new services can be created dynamically and become part of the problem solving activity.

Implementation of the Agent-based OSA

In our vision the implementation process is composed by two different steps. As first step we have built a prototype of the Agent Web Server: the User and Worker Agent and the facilitator. As second step we are working to integrate the facilitator in the JXTA architecture (Gong 2001). Agents remain independent objects, that communicate always at the knowledge-level.

Our prototype is an extension of the Apache/Tomcat platform, and represents an Agent Web Server, as depicted in figure 3. The facilitator and the failure detector (written in Java) are an extension of the Web Server, while the User Agents can be coded in any language that supports SOAP communication over standard TCP socket. We distinguish between 2 types of communication:

- from an Agent to its facilitator and vice versa. Messages are first translated into SOAP messages, and then sent to the facilitator via TCP sockets. This choice allows communication even for agents running on handheld devices

with little computational power and enable programmers to write agents in virtually every programming language.

- from an Agent $\hat{a}$ to another Agent $\hat{b}$ (or to a set of other Agents): messages are always sent to the local facilitator, that takes care about the communication. The facilitator of User Agent $\hat{a}$ receives the message, analyzes it and then forwards it to the facilitator of the Agent $\hat{a}$ (or to the facilitators of the agents in the set). Note that $\hat{a}$ and $\hat{b}$ can share the same facilitator.

The facilitator is designed to be integrated as a soap service and to achieve this goal we have used the latest Apache SOAP library. With this library, soap services can be easily published and used as rpc commands. The procedure that implements the call of these methods already integrates a failure mechanism, so the program that asks for a soap service waits for an acknowledge message. We make some modifications to this library in order to supply an extra function used to achieve a totally asynchronous call: the new method is called `invoke-async` and it is a function that allows the sender to not wait for a response, so that failures are always handled by the failure detector system.

The methods published by the facilitator are a superset of the agent primitives found in table 1. They contains all the primitives functions used by agents like `insert`, `ask-one`, `tell`, etc. and a new one, named `agents-list` used only at the facilitator level to retrieve names and related capabilities of known agents. Almost all of these primitives contain two different code sections that handle the two different situations: when the message is received from the local agent or when it is received from another agent in the network. At the facilitator level some primitives, like `ask-one` and `tell`, have an extra parameter used to identify the right answer of the message, as specified in (Dragoni & Gaspari 2003).

We are now working to integrate the facilitator in the Project JXTA 2.0 Super-Peer Virtual Network (Traversat *et al.* 2003). Project JXTA is an industry leading peer-to-peer platform, originally conceived by Sun Microsystems Inc. and released as open source software. In our vision the JXTA network acts as a physical layer, while the facilitator allows communication at Knowledge Level. Binding an Agent's facilitator to a JXTA peer means that all the physical communication between agents are managed entirely by the JXTA network. Furthermore, the integration with the Project JXTA network supports our architecture with facilities such as:

- the creation of self-organized protected virtual domains, a set of peers interested in a specific topics;
- encrypted connections, where messages are automatically encrypted using TSL.

Finally, because JXTA resources are described in plain XML, we plan to extend some JXTA concepts with metadata in order to attach a semantic description to resources.

Extending the Agent-based OSA with Semantic Web Services

Semantic Web Services (McIlraith, Son, & Zeng 2001; McIlraith & Martin 2003) is an ongoing research area which aims to bring Web Services to their full potential by means of Semantic Web technology. The overall approach is that by augmenting Web Services with rich formal descriptions of their competence many aspects of their management (such as Web Service discovery, invocation and composition) will become automatic. To realize this vision many open problems need still to be solved. In our opinion, the fundamental ones are:

1. Provide a language to *semantically express* the capabilities of Web Services (or service advertisements) and the service requests. Main ongoing works in this direction are WSMO (WSMO Working Group 2004), IRS-III (Domingue *et al.* 2004) and OWL-S (Martin *et al.* 2004).
2. Provide an infrastructure which supports the creation of Semantic Web Services. The infrastructure must clarify *who realize Web Services* and *where the semantic descriptions of Web Services are stored* (in a centralized or distributed repository).
3. Enable *automatic discovery and invocation* of Web Services, that is, enable agents to discover and invoke Web Services on the basis of the capabilities that they provide. The discovery problem is also known as "Semantic Matching problem" (Paolucci *et al.* 2002).

We briefly discuss how our architecture can easily support Semantic Web Services with respects to the above fundamental issues.

1. For the sake of simplicity, in our proposal we have assumed that Web Service capabilities are expressed as a set of propositions, one for each service. However, to support Semantic Web Services we allow OWL-S descriptions as content of ACL messages. Therefore, Worker Agents can advertise their competences by means of OWL-S descriptions and in a similar way User Agents (or Worker Agents) can ask for services by means of OWL-S requests.
2. In our architecture, Worker Agents realize Web Services. The architecture is open and dynamic, meaning that it allows the creation of new services (creating new Worker Agents) and the modification of existing ones. Moreover, the Web Service descriptions are stored in facilitators, realizing a distributed repository of capabilities. Worker Agents can register their capabilities by means on the dedicated ACL primitive *register*.
3. Automatic discovery can be realized by the anonymous interaction protocol of our ACL, which allows an agent to perform a request of a service without knowing the name of recipient agents. The matching operation between agent requests and Web Service capabilities is then performed by facilitators which execute a matching algorithm (for example, the one described in (Paolucci *et al.* 2002)). Then the selected Web Service can be invoked by means of the ACL primitive *ask-one*.

Conclusions

We have presented the design of an Agent-based OSA which integrates knowledge level agents on the Semantic Web. The nodes of our OSA are agents which can use standard knowledge modelling technologies and communicate using a fault-tolerant ACL. The ACL based interface has been designed to support coordination and cooperation among agents, thus can be used in all the application domains which require these functionalities. At the moment we have not addressed the problem of realizing specific primitives for applications with real time requirements.

We have implemented the first prototype of our OSA, but for the lack of time some problems remain open. The prototype provides a simple authentication protocol which enables agents to join the OSA. The failure detection system is not already developed at this time, although we don't see particular implementation problems. Giving agents the possibility to reply to a message without specify the referred message offers an abstraction layer that must be carefully handled. There are a variety of methods that could be used: we have chosen to integrate a callback function in the primitives that do interactive communication, but we need more intensive testing. The `clone` is another unimplemented feature in the prototype. We are studying if the implementation should be limited on local machine or, otherwise, which enhancement could lead the possibility to do a `clone` primitive on remote machines.

Our future work will concern:

- The semantic specification of a Web Service. We are investigating on the integration of a description model for Semantic Web Services, either OWL-S or WSMO.
- The test of our OSA with a set of application scenarios, among them an *intelligent news server* and a *travel organizer*.
- The completion of the agent authentication protocol. We plan to fully integrate in our OSA the JXTA authentication protocol, currently under development.

References

Chandra, T. D., and Toueg, S. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 43(2):225–267.

Davies, N.; Fensel, D.; and Richardson, M. 2004. The future of Web Services. *BT Technology Journal* 22(1).

Domingue, J.; Cabral, L.; Hakimpour, F.; Sell, D.; and Motta, E. 2004. IRS-III: A Platform and Infrastructure for Creating WSMO-based Semantic Web Services. In *Proceedings of the WIW 2004 Workshop on WSMO Implementations*.

Dragoni, N., and Gaspari, M. 2003. Integrating Agent Communication Languages in Open Services Architectures. Technical Report UBLCS-2003-12, Department of Computer Science, University of Bologna, ITALY.

Dragoni, N., and Gaspari, M. 2004. An Object Based Algebra for Specifying A Fault Tolerant Software Architecture. Accepted for publication in JLAP (Journal of Logic and Algebraic Programming) special issue on "Process algebra and system architecture".

Gaspari, M. 1998. Concurrency and Knowledge-Level Communication in Agent Languages. *Artificial Intelligence* 105(1-2):1–45.

Gong, L. 2001. JXTA: A Network Programming Environment. *IEEE Internet Computing* 5:88–95.

Gruber, T. 1993. A Translation Approach to Portable Ontologies. *Knowledge Acquisition* 5(2):199–220.

He, H.; Haas, H.; and Orchard, D. 2004. Web Services Architecture Usage Scenarios. Technical Report NOTE-ws-arch-scenarios-20040211/, W3C. http://www.w3.org/TR/2004/NOTE-ws-arch-scenarios-20040211/.

Martin, D.; Paolucci, M.; McIlraith, S.; Burstein, M.; McDermott, D.; McGuinness, D.; Parsia, B.; Payne, T.; Sabou, M.; Solanki, M.; Srinivasan, N.; and Sycara, K. 2004. Bringing Semantics to Web Services: The OWL-S Approach. In *First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*.

McIlraith, S., and Martin, D. 2003. Bringing Semantics to Web Services. *IEEE Intelligent Systems* 18(1):90–93.

McIlraith, S.; Son, T.; and Zeng, H. 2001. Semantic web services. *IEEE Intelligent Systems, Special Issue on the Semantic Web* 16(2):46–53.

Mullender, S. 1993. *Distributed Systems*. ADDISON-WESLEY.

Paolucci, M.; Kawmura, T.; Payne, T.; and Sycara, K. 2002. Semantic Matching of Web Services Capabilities. In *Proceedings of the first International Semantic Web Conference (ISWC)*.

Traversat, B.; Arora, A.; Abdelaziz, M.; Duigou, M.; Haywood, C.; Hugly, J.-C.; Pouyoul, E.; and Yeager, B. 2003. Project jxta 2.0 super-peer virtual network. Available online: `http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf`.

W3C Web-Ontology Working Group. 10 February 2004. *OWL Web Ontology Language Guide*. W3C Recommendation.

WSMO Working Group. 2004. Web Service Modeling Ontology (WSMO). http://www.wsmo.org/2004/d2/. WSMO Working Draft D2v1.1.