Intelligent Web Servers as Agents

Mauro Gaspari **Nicola Dragoni** **Davide Guidi**

Technical Report UBLCS-2004-14

July 14, 2004

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

The University of Bologna Department of Computer Science Research Technical Reports are available in PDF and gzipped PostScript formats via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` or via WWW at URL `http://www.cs.unibo.it/`. Plain-text abstracts organized by year are available in the directory `ABSTRACTS`.

Recent Titles from the UBLCS Technical Report Series

2003-11 *WSSecSpaces: a Secure Data-Driven Coordination Service for Web Services Applications*, Lucchi, R., Zavattaro, G., September 2003.

2003-12 *Integrating Agent Communication Languages in Open Services Architectures*, Dragoni, N., Gaspari, M., October 2003.

2003-13 *Perfect load balancing on anonymous trees*, Margara, L., Pistocchi, A., Vassura, M., October 2003.

2003-14 *Towards Secure Epidemics: Detection and Removal of Malicious Peers in Epidemic-Style Protocols*, Jelasity, M., Montresor, A., Babaoglu, O., November 2003.

2003-15 *Gossip-based Unstructured Overlay Networks: An Experimental Evaluation*, Jelasity, M., Guerraoui, R., Kermarrec, A-M., van Steen, M., December 2003.

2003-16 *Robust Aggregation Protocols for Large-Scale Overlay Networks*, Montresor, A., Jelasity, M., Babaoglu, O., December 2003.

2004-1 *A Reliable Protocol for Synchronous Rendezvous (Note)*, Wischik, L., Wischik, D., February 2004.

2004-2 *Design and evaluation of a migration-based architecture for massively populated Internet Games*, Gardenghi, L., Pifferi, S., D'Angelo, G., March 2004.

2004-3 *Security, Probability and Priority in the tuple-space Coordination Model (Ph.D. Thesis)*, Lucchi, R., March 2004.

2004-4 *A New Graph-theoretic Approach to Clustering, with Applications to Computer Vision (Ph.D Thesis)*, Pavan., M., March 2004.

2004-5 *Knowledge Management of Formal Mathematics and Interactive Theorem Proving (Ph.D. Thesis)*, Sacerdoti Coen, C., March 2004.

2004-6 *An architecture for Content Distribution Internetworking (Ph.D. Thesis)*, Turrini, E., March 2004.

2004-7 T-Man: Fast Gossip-based Construction of Large-Scale Overlay Topologies, Jelasity, M., Babaoglu, O., May 2004.

2004-8 A Robust Protocol for Building Superpeer Overlay Topologies, Montresor, A., May 2004.

2004-9 A Unified Approach to Structured, Semistructured and Unstructured Data, Magnani, M., Montesi, D., May 2004.

2004-10 Exact Methods Based on Node Routing Formulations for Arc Routing Problems, Baldacci, R., Maniezzo, V., June 2004.

2004-11 Mapping XQuery to Algebraic Expressions, Magnani, M., Montesi, D., June 2004.

2004-12 VDE: Virtual Distributed Ethernet, Davoli, R., June 2004.

2004-13 SchemaPath: Extending XML Schema for Co-Constraints, Marinelli, P., Sacerdoti Coen, C., Vitali, F., June 2004.

Intelligent Web Servers as Agents

Mauro Gaspari[1] **Nicola Dragoni**[1] **Davide Guidi**[1]

Technical Report UBLCS-2004-14

July 14, 2004

Abstract

Software agents have been recognized as one of the main building blocks of the emerging infrastructure for the Semantic Web, but their relationship with more standard components, such as Web servers and clients, is still not clear. At the server side, a possible role for agents is to enhance the capabilities of servers using their intelligence to provide more complex services and behaviors. In this paper we explore the role of agents at the server side presenting an Open Service Architecture (OSA) which extends the centralized Internet Reasoning System (IRS-II) to a distributed scenario. The architecture uses a distributed facilitation protocol which integrates Web Services with agent communication languages. Finally we present an implementation which extends Tomcat with these features.

1. Dipartimento di Scienze dell'Informazione, University of Bologna, via Mura Anteo Zamboni 7, 40127 Bologna, Italy.

1 Introduction

Although there are many small-scale examples of implemented systems which provide knowledge services on the Web, several issues concerning the full exploitation of semantic Web technologies in the Internet remain open. A significant challenge to address is the design of distributed reasoning infrastructures tightly integrated with current Internet components and technologies that would allow semantic Web Services [10, 9] to be exploited in the large. While new standards are emerging such as the Ontology Web Language (OWL) [16], and the community is currently addressing many central issues as the design of a Web Service Modelling Ontology (WSMO) [4], there is still not a widely accepted architecture for the underlying reasoning infrastructure.

Agents have been recognized as one of the main building blocks of this emerging 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. When used at the client side agents should provide intelligent support and advanced services to user for example being able to retrieve, compose and execute semantic Web Services (semi)automatically. However another possible role for agents is to enhance the capabilities of servers using their intelligence to provide more complex services and behaviors. For example the Internet Reasoning Service (IRS-II) [12] is a knowledge based server which supports the publication, location, composition and execution of semantic Web Services. The IRS provides facilities to achieve complex tasks, if all the needed services are available. It explicitly distinguishes between tasks (what to do) and methods (how to achieve tasks) and as a result supports capability-driven service invocation, flexible mappings between services and problem specifications and dynamic, knowledge-based service selection.

We argue that the features of the IRS can be generalized and distributed over the Internet extending Web servers with the IRS functionalities. According to this idea a Web server can be seen as an agent which manages knowledge-level descriptions of its capabilities and is able to cooperate with other "intelligent Web servers" in order to perform complex tasks.

In this paper we try to clarify this view of agents at the server side proposing an Open Service Architecture (OSA) which integrates the IRS framework with agent communication languages in a distributed setting.

The paper is organized as follows. In Section 2 we recall the approach and the architecture of IRS-II. Then in Section 3 we propose a Knowledge-Level Open Service Architecture (OSA) which extends Web servers with the IRS functionalities and agent's capabilities providing semantic Web Services. In Section 4 we focus on the architecture of a single node of the proposed OSA. Then in Section 5 we present a Tomcat based implementation of the OSA. The final Section of the paper contains our conclusions and highlights our future work.

2 IRS-II

IRS-II [12] is a framework and implemented infrastructure which is aimed at supporting the publication, location, composition and execution of semantic Web Services. IRS-II has three main classes of features which distinguish it from other work on semantic Web Services. Firstly, it supports *one-click publishing* of standalone software: IRS-II automatically creates the appropriate wrappers, given pointers to the standalone code. Secondly, it explicitly distinguishes between *tasks* (what to do) and *methods* (how to achieve tasks) and as a result supports *capability-driven* service invocation, flexible mappings between services and problem specifications, and dynamic knowledge-based service selection. Finally, IRS-II services are web service compatible: standard web services can be trivially published through the IRS-II and any IRS-II service automatically appears as a standard web service to other web service infrastructures.

2.1 IRS-II Approach

Semantic Web Services are described by means of the UPML framework [7] developed within the IBROW project [1]. Figure 1 shows how a single knowledge based application would be

described in UPML terms. The UPML framework partitions knowledge into four classes of components: **Task Models**, which provide a generic description of the task to be solved, **Problem Solving Methods (PSMs)**, which provide abstract, implementation independent descriptions of reasoning processes that can be applied to solve tasks in specific domains, **Domain models**, which describe the domain of an application and **Bridges**, which specify mappings between the different model components within an application. Each class of component is specified by means of an appropriate **ontology**.

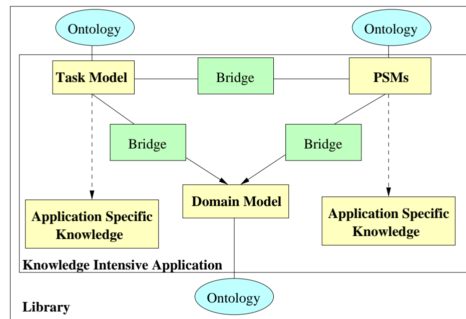


Figure 1. The UPML framework.

The application of the UPML framework to semantic Web Services provides a number of advantages. In particular, the explicit separation between tasks and methods provides [12]:

- A task-based mechanism for aggregating services. It is possible to specify service types (i.e. tasks) independently from specific service providers.
- A basic model for dealing with ontology mismatches. UPML assumes that the mapping between methods and tasks may be mediated by bridges. In practice this means that if task T is specified in ontology A and a method M is specified in ontology B, which can be used to solve T, it is still possible to use M to solve T, provided the appropriate bridge is defined.
- Capability-driven service discovery and invocation (find me a service that can solve problem X).

2.2 IRS-II Architecture

The overall architecture of the IRS-II is shown in Figure 2. The main components are the **IRS Server**, the **IRS Publisher** and the **IRS Client**, which communicate through a SOAP-based protocol [3].

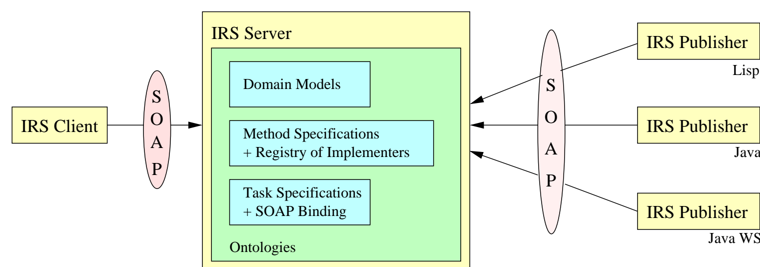


Figure 2. The IRS-II architecture.

The **IRS server** holds descriptions of semantic web services at two different levels. A knowledge level description is stored using the UPML framework of tasks, PSMs and domain models

represented internally in OCML [11]. Additionally, two sets of mappings are used to connect the knowledge level descriptions to a specific web service. The **IRS Publisher** plays two roles in the IRS-II framework. Firstly, it links web services to PSMs within the IRS server. Secondly, it automatically generates a set of wrappers which turn standalone Lisp or Java code into a web service linked to a PSM. A key feature of IRS-II is that web service invocation is capability driven. **The IRS Client** supports this by providing functionalities, which are task centric. An IRS-II user simply asks for a task to be achieved and the server locates an appropriate PSM and then invokes the corresponding web service.

3 A Knowledge Level Open Service Architecture

An Open Service Architecture (OSA) 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. The nodes of current Web Services architecture [14] are Web servers which use Web Service Description Language (WSDL) and the Universal Description, Discovery and Integration (UDDI) registry to provide the above facilities [3, 15].

We describe here a Knowledge-Level extension of this architecture which extends Web servers with the IRS functionalities and agents' capabilities providing semantic Web Services [10, 9]. Agents becomes the main building block of this architecture. They are geographically distributed (as Web servers are) and can provide a set of semantic 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. Moreover each agent provides the IRS features: it supports one-click publishing of "standard" programming code and it is based on knowledge modeling research on reusable components for knowledge-based systems [6, 11].

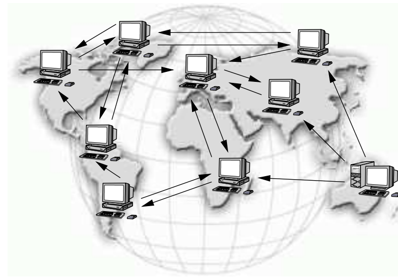


Figure 3. In an Open Service Architecture agents are geographically distributed and provide a set of semantic Web Services to the outside world.

Semantic Web Services allow agents to perform complex problem solving operations and dynamic reconfiguration of available services. However several issues arise in designing an OSA which will enable agents to fully exploit these potentialities. We address some of them in the following, and later, in the conclusions, we outline those we have not taken into consideration.

A first issue is related to the geographically distributed nature of the nodes of the OSA which are subject to possible failures or network partitions. Most of the available knowledge based systems on the Web (such as the IRS-II) 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-II is a centralized system where all the available services should be published. If the IRS-II 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 a KL-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 node should support the publication of Web Services which are developed locally in a given site and should provide a protocol to broadcast 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 semantic Web Services using high level mechanisms, but should also support agent-to-agent interaction. We argue that the 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 can be successfully used for this task, especially if they provide support for fault tolerant communication primitives as suggested in [5].

In the following we outline the main features of our Knowledge Level OSA. We assume *knowledge level* agents [8], that is, they should concern with the use, request and supply of knowledge without dealing with symbol level issues. As in IRS-II, our OSA holds descriptions of semantic Web Services at a knowledge level, that is, knowledge level descriptions of Web Services are stored by means of the UPML framework of tasks, PSMs and domain models.

Open architecture. New Web Services can be activated creating new agents which deal with them, publishing new services, or augmenting available Web Services with semantic descriptions as in IRS-II.

Agent Communication. Agents access services and communicate each other using a fault tolerant Agent Communication Language (ACL) which provides one-to-one and one-to-many primitives [5]. 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). The asynchrony of the system implies that there is no bound on message delay, clock drift or the time necessary to execute a step (so we omit all timing-based failures).

Competences of Agents. Each agent can provide a set of semantic Web Services which constitutes its capabilities. We assume that capabilities of each agent are known by all the other agents. That is, when an agent updates its Knowledge Base (KB) with a new capability (for example adding a task $\mathbb{T}_k$), it forwards this information to all the other agents, for example by means of an agent primitive *register(myself,* $\mathbb{T}_k$*)* as in [5]. We also assume that agents communicate only their capabilities, not their PSMs. Therefore each agent knows all the problems which can be solved by the OSA, but it knows *how* to solve a task only if it has in its KB a PSM which can solve the task.

Crash Failures. 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[2]. An OSA should deal with such failures preventing agents to wait answers from crashed agents. To this purpose our OSA is integrated with a *distributed failure detector* [2] which allows agents to detect crashed agents. As we show in next Section, the distributed failure detector allows also to add fault tolerance to the primitives of the ACL.

Agents are able to find, execute and compose available Web Services providing more sophisticated instances of them, as shown in Figure 4. In that Figure, two scenarios of Web Service requests are illustrated. In Scenario 1 the agent $\mathbb{A}$ asks an agent $\mathbb{P}$ for a Web Service S. $\mathbb{P}$ is able to provide S and it replies to $\mathbb{A}$. Scenario 2 is more complex: to execute the Web Service S, $\mathbb{P}$ needs to execute other services. If $\mathbb{P}$ is not able to solve these tasks by itself, it requests the services to

2. 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 [13].

PSfrag replacements

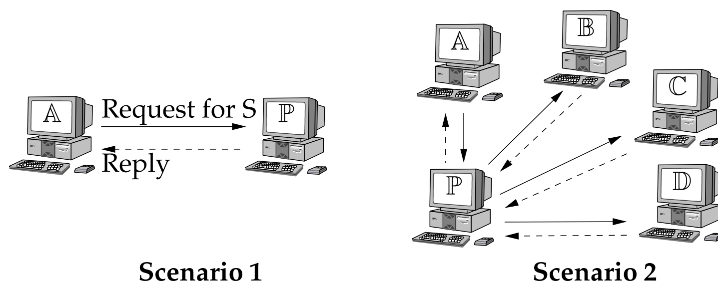


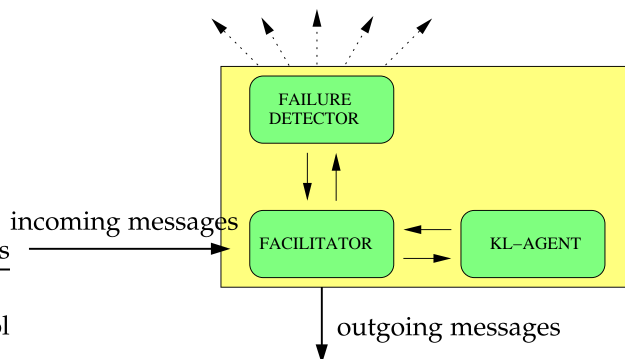
Figure 4. Examples of Web Service requests using an ACL

other agents which are able to solve them. When it receives all the replies it executes the service S and then it replies to $\mathbb{A}$.

4 The IRS Agent

In this section we describe in details the architecture of a single node of our knowledge level OSA. A node is a Web server which integrates an agent providing the IRS functionalities. 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, provided that they satisfy a set of well defined requirements which are illustrated in the next section. In fact 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.

PSfrag replacements



which need control

Figure 5. The architecture of a single node of the distributed IRS.

The architecture we propose is presented in Figure 5; it extends a Web server with three components:

- **KL-agent**: operates at the knowledge level, provides a set of semantic Web Services and it is able to perform complex problem solving operations interacting with other agents. A KL-agent is reactive (it reacts to requests of services) but it can also have a proactive function to solve complex tasks.

- **facilitator**: this component is associated to a KL-agent and implements a distributed facilitator mechanism which provides anonymous (contents based) facilities to retrieve and request services. It registers the competences of a KL-agent and forwards these competences to other facilitators executing a distributed protocol. The facilitator should use semantic web standards such as OWL-s to represents the competence of agents.
- **failure detector**: this component implements a distributed failure detector mechanism. It monitors all the nodes in the system and communicates to the facilitator those that it currently suspects to have crashed.

In principle several KL-agents with the associated facilitator 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 the agent is to provide an independent component. The agents has no constraints on the implementation language or knowledge representation formalisms it adopts, but it reacts to a well defined protocol based on the standard primitives of an agent communication language. The primitives of this ACL [5] can be divided into four categories as shown in Table 1. Contents based services requests are realized as one-to-many primitives: whenever an agent needs a given service which solve a task T it can execute these multicast primitives. The language supports an **anonymous interaction protocol** which has been developed for Open Service Architectures and which is integrated with standard agent-to-agent primitives. This allows an agent to perform a request of services based on contents without knowing the name of the recipient agent. If required they can also continue the cooperation using agent-to-agent communication primitives. Another feature that our ACL provides to Open Services Architectures is a support for agent creation and cloning. Thus new agents and new services can be created dynamically and become part of the problem solving activity.

5 An implementation based on Tomcat

A sketch representing how a generic node is implemented can be found in figure 6. We have chosen to build our OSA on a stable framework: the Apache Web Server together with the Tomcat Servlet/JSP container.

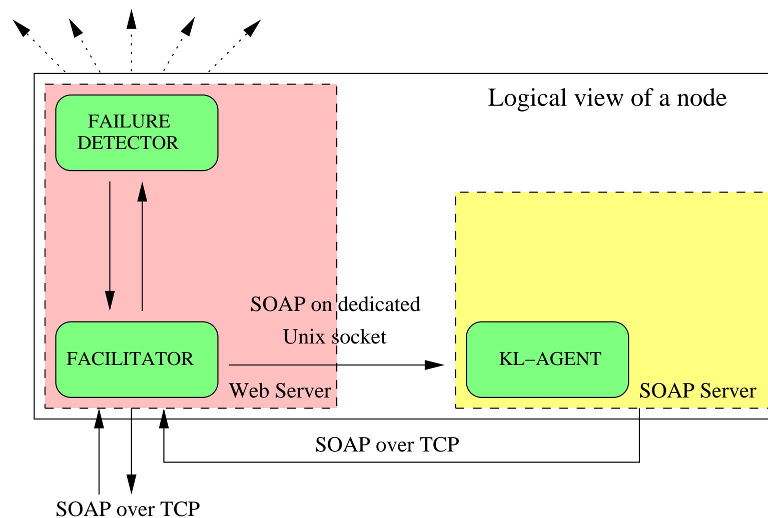


Figure 6. Implemented agent architecture.

In our implementation the facilitator and the failure detector are integrated into the Web server, while the agent can be coded in any language that supports SOAP communication. Both

Table 1. **Knowledge-level specification of ACL primitives**

Agent primitives	Knowledge-level behaviour
Standard Conversation primitives	
insert($\mathbb{A}$, $\mathbb{B}$, p)	Agent $\mathbb{B}$ asks agent $\mathbb{A}$ to insert a proposition p in its KB.
ask-one($\mathbb{A}$, $\mathbb{B}$, T)	Agent $\mathbb{B}$ asks agent $\mathbb{A}$ for a service which solves the task T.
tell($\mathbb{A}$, $\mathbb{B}$, T)	Agent $\mathbb{B}$ sends agent $\mathbb{A}$ an instantiation of T.
One-to-many primitives (Contents based services requests)	
ask-everybody($\mathbb{B}$, T)	Agent $\mathbb{B}$ asks all agents which are able to solve T for a solution of T.
ask-first($\mathbb{B}$, T)	Agent $\mathbb{B}$ asks all agents which are able to solve T for a solution of T and it gets the first one.
ask-best(L, $\mathbb{B}$, T)	Agent $\mathbb{B}$ asks the first agent in the list L despite failures, say $\mathbb{C}$, for a solution of T.
all-answers(T)	This predicate succeeds when an agent receives all the answers relative to a given task request T despite failures.
Registration of new services	
register($\mathbb{B}$, T)	An agent $\mathbb{B}$ tells other agents that is able to solve the task T.
unregister($\mathbb{B}$, T)	An agent $\mathbb{B}$ tells other agents that is no longer able to solve the task T.
Support for open architectures	
hello($\mathbb{A}$, $\mathbb{B}$)	An external agent $\mathbb{B}$ sends agent $\mathbb{A}$ a request to be added to the system.
create($\mathbb{B}$, w)	Creates a new agent (*i.e.,* a facilitator and KL-Agent on the same Web server) with a new name $\mathbb{B}$ and a new knowledge base w. The name of the new agent is known only by the agent that creates it.
clone($\mathbb{B}$)	When an agent executes this primitive it creates a new agent (*i.e.,* a facilitator and KL-Agent on the same Web server) with a new name $\mathbb{B}$ which has the same competences of the creator (if the agent $\mathbb{A}$ was registered in the system for the task T_k and $\mathbb{A}$ clones itself creating the agent $\mathbb{B}$, then $\mathbb{B}$ will also be registered for T_k);
bye	Terminates the execution of the IRS-agent which performs the primitive. This information is sent to all the IRS-agents in the system.

the facilitator and the failure detector can be seen as the infrastructure that manages the communication.

Messages from the facilitator to the local agent are first translated into SOAP messages, and then sent using a standard Unix socket (activated at initialization time). Messages from local agent to other agents are always sent to the local facilitator, and then processed. The local agent uses SOAP over TCP protocol to send the message, the same techniques used by facilitators to talk with other facilitators. From the facilitator point of view, the only way to distinguish a message sent by the local agent from another message sent by another agent is to check the

sender field in the message itself.

The facilitator is designed to be integrated as a SOAP service. 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. We made some modifications to this library in order to achieve asynchronous communication. In fact, the original procedure that implements the call of these methods integrates a failure mechanism, so the program that asks for a SOAP service waits for an acknowledge message. This was not correct for our implementation because we delegate communication errors to the failure detection component. So we made a little hack 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 doesn't return any value. In this way the sender does not wait for a response and failures are always handle by the failure detector system.

Using the Apache SOAP library, the facilitator exports its methods to the 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 [5]. To deeply understand how the communication is implemented, suppose that agent A would communicate with agent B. Suppose that the selected communication primitive is the `ask-one(B, A, p)`. Agent B will receive the message and then reply with the primitive `tell(A, B, r)`. The communication protocol we have implemented is not related to how the messages p and r are coded, and allows agent B to reply with the tell primitive without any specific reference to the previous `ask-one` message. This goal is achieved implementing the `ask-one` primitive as a callback function. The communication process from agent A to agent B is the following:

- agent A sends an `ask-one(B, A, p)` message to its facilitator.
- the facilitator of the agent A receives the message and starts a callback function that sends the message to the facilitator of the agent B and waits for the related reply.
- the facilitator of the agent B forwards the message to agent B

Facilitators play a fundamental role in this communication process. When agent A sends the message to its facilitator, some operations starts:

- the facilitator starts a new thread containing a callback function;
- a tag that identifies the new thread is generated;
- the message p, together with the tag, is sent to the facilitator of agent B;
- the new thread waits for the associated reply. It will recognize the correct message because the related `tell` primitive will have the same tag associated to the message r. Once the reply is retrieved, the tag is deleted from the message r, and then the correct primitive `tell(A, B, r)` is sent back to agent A.

Two meaningful primitives are the `init` and the `hello`. The facilitator code provides an initialization function, `init`, that should be called before anything else. This function provides some variable initialization, starting the log subsystem and initializes the communication with the local agent via a standard Unix socket. The `init` method requires one parameter that specifies the port number that must be used for the communication within the agent. In our vision the name of the agent is composed by two parts: the hostname of the machine that hosts the service and the ObjectURI, that represents the id of the service. However in the prototype that we have realized we refer to an agent with just the hostname, so the hostname identifies the agent itself.

The problem of integrating agents from disconnected networks is addressed by the `hello` primitive (and the related `agents-list` method). The `hello` method requires two parameters: the name of the sender and the name of the receiver. The receiver agent replies with its list of all known agents and their capabilities (using the `agents-list` method) and then stores the sender in its list of known agents, if it was not there before. The list of the agents and their capabilities is then retrieved and stored by the facilitator.

5.1 Integrating the IRS functionalities

In the current prototype we have only integrated a subset of the IRS system. The IRS is integrated realizing an agent which accept SOAP messages implemented in OCML. This is achieved transforming the IRS-II server which was based on a lisp HTTP server into a more simple SOAP server. The IRS client and the IRS publisher are realized as Web pages, which contains the standard invocations to the IRS operations for publishing and achieving tasks. The glue code generated for accessing the published services is similar to one generated in the centralized version of the IRS-II, but their competences are registered in the facilitator.

6 Conclusions

We have presented the design of a KL OSA which integrates the IRS-II system with a standard Web server in a distributed scenario. The nodes of our OSA are agents which use knowledge modelling technologies and communicate using a fault-tolerant ACL. We have implemented the first prototype of our OSA, but for the lack of time some problems remain open. 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 choosen 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 OSA is still independent about the way the communication messages between agents are internally realized. In the prototype we have built only simple forms of messages, with a syntax like Prolog or Lisp. But other complex specification of the service can be used. We're investigating on the integration of a description model for semantic web services, either OWL-S or WSMO. WSMO also propose an interesting architecture for the orchestration of Web services which could be integrated in this setting. Finally, some work should be done refining the startup of an agent and its facilitator and to support the integration with all the functionalities provided by the IRS-II system.

References

[1] V. R. Benjamins, E. Plaza, E. Motta, D. Fensel, R. Studer, B. Wielinga, G. Schreiber, and Z. Zdrahal. An Intelligent Brokering Service for Knowledge-Component Reuse on the World-Wide-Web. In *Proceedings of 11th Workshop on Knowledge Acquisition, Modeling and Management*, Alberta, Canada, April 1998.

[2] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[3] F. Curbera, M. Dutler, R. Khalaf, N. Mukhi, W. Nagy, and S. Weerawarana. Unraveling the Web Services Web - An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

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

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

[6] D. Fensel and E. Motta. Structured Development of Problem Solving Methods. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):913–932, 2001.

[7] Dieter Fensel, Enrico Motta, V. Richard Benjamins, Monica Crubezy, Stefan Decker, Mauro Gaspari, Rix Groenboom, William Grosso, Frank van Harmelen, Mark Musen, Enric Plaza, Guus Schreiber, Rudi Studer, and Bob Wielinga. The Unified Problem-solving Method Development Language UPML. *Knowledge and Information Systems*, 5(1):83–131, 2003.

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

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

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

[11] E. Motta. *Reusable Components for Knowledge Modelling*. IOS Press, 1999.

[12] E. Motta, J. Domingue, L. Cabral, and M. Gaspari. IRS-II: A Framework and Infrastructure for Semantic Web Services. In *Proceedings of the International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 306–318, Sanibel Island, FL, USA, October 2003. Springer Verlag.

[13] S. Mullender. *Distributed Systems*. ADDISON-WESLEY, 1993.

[14] E. Newcomer. *Understanding Web Services*. ADDISON-WESLEY, 2002.

[15] A. Tsalgatidou and T. Pilioura. An Overview of Standards and Related Technology in Web Services. *Distributed Parallel Databases*, 12(2-3):135–162, 2002.

[16] W3C Web-Ontology Working Group. *OWL Web Ontology Language Guide*, 10 February 2004. W3C Recommendation.