Chapter VI

Mobile Agents Meet Web Services

Cristian Mateos, Universidad Nacional de Centro, Argentina

Alejandro Zunion, Universidad Nacional de Centro, Argentina

Marcelo Campo, Universidad Nacional de Centro, Argentina

Abstract

Web services standards provide the basis for interoperability, discovery and integration of distributed applications. Web services will enable mobile agents to better use and exploit Web accessible applications and resources. However, there is a lack of tools for integrating mobile agents and Web services. This chapter presents MoviLog, a novel programming language for enabling mobile agents to consume Web services. The most interesting aspect of the language is its reactive mobility by failure mechanism that allows programmers to develop mobile agents without explicitly providing code for handling mobility or Web services invocations.

Introduction

Many researchers envision the Web of the future as a global community where people and intelligent agents interact and collaborate (Hendler, 2001). Unfortunately, today's Web has been designed for human interpretation and use (McIlraith et al., 2001), generally for reading and browsing HTML pages and online form filling. However, there is a need for automating the interoperability of B2B (business-to-business) and e-commerce applications. Until now, this interoperation has been handled by using programs that interact with Web accessible services to obtain and then parse HTML content for extracting data. This approach is very weak since it depends on the format of the HTML pages, and the interfaces for accessing services (e.g., CGI or RMI). In order to achieve a truly automatic interoperability between programs and Web accessible resources, new technologies aim at creating a *Semantic Web* (Berners-Lee et al., 2001), where information and services offered by any site are described in a nonambiguous and computer-understandable way.

In the scenario of the Web consisting of sites with highly dynamic content, mobile users, unreliable links and small portable devices such as personal digital assistants (PDAs) and cellular phones, mobile agents will play a fundamental role (Hendler, 2001). A mobile agent is a computer program that represents a user in a computer network and is able to migrate autonomously from site to site to perform task on behalf of the user (Tripathi et al., 2002). This feature is particularly interesting when an agent makes sporadic use of a valuable shared resource located at a remote site. In addition, efficiency can be improved by moving agents to a host to query large repositories and then return with the results, thus avoiding multiple interactions with the data over network links subjected to delays or interruptions of services.

Mobile agents exhibit a number of properties that make them suitable for exploiting the potential of the Web, because they add mobility—the capacity to migrate across sites of a network (Fuggetta et al., 1998)—to common capacities of ordinary intelligent agents such as reaction, perception, deliberation and autonomy. Some of the most significant advantages of mobile agents are their support for disconnected operations, heterogeneous systems integration, robustness and fault-tolerance (Lange & Oshima, 1999; Milojicic et al., 1999).

Despite the number of applications that can be benefited from the usage of mobile agents (Kotz & Gray, 1999), this technology has shown difficulties when used for interacting with Web content (Hendler, 2001). Agents' inability to understand concepts required for invoking and using Web accessible services and resources requires the creation of a Semantic Web, where content is described according to precise semantics. In this sense, we claim that there is a need for a mobile agent development tool for solving these problems which preserve, at the same time, the key benefits of mobile agent technology.

A step towards the widespread adoption of mobile agents is MoviLog (Zunino et al., 2002). MoviLog is a platform for building Prolog-based (Bramer, 2005) intelligent mobile agents for the Web that provides a novel mechanism for handling mobility named reactive mobility by failure (RMF). This mechanism allows the programmer to exploit the advantages of mobility without explicitly programming mobile code.

In order to take advantage of the features of mobile agents for building Web applications, we have extended MoviLog to invoke Web services. This offers a great opportunity for building distributed applications based on intelligent mobile agents that access Web information and resources in an automatized form. For example, it will be possible to automate classic e-commerce applications such as e-shops, e-malls and e-actions, allowing automatic interaction between participating entities at both sides of each transaction, along with a minimal programming effort.

This chapter is structured as follows: The next section introduces Web services and the Semantic Web. Next, we describe the most relevant work. Then, MoviLog is briefly introduced. After that, we explain our approach for integrating MoviLog and Web services. Then, an agent implemented with MoviLog is described. Finally, conclusions and future work are presented.

Web Services and the Semantic Web

Unlike the current Web, Web services (Vaughan-Nichols, 2002) (i.e., Web accessible programs and devices) can be seen as a set of programs interacting in a network without human intervention. In order to enable programs to interchange data, it is necessary to define communication protocols, formats for data transfers, and specific points on which the communication will be established. Such definitions must be made in a rigorous way, preferably by using a computer-understandable language with well defined semantics.

Web services are a natural consequence of the evolution of the Web into a more open medium that facilitates complex and systematic interactions between applications (Curbera et al., 2001). A fundamental goal of Web services is to provide a common representation of the applications that use different communication protocols and interaction models. A natural approach to cope with this is to decouple the abstract descriptions of application functionality from the interaction model involved, and then representing such descriptions in a common language that can be interpreted by every single application.

The technological backbone of Web services is based on standardization efforts centered on extensible markup language (XML) (W3C Consortium, 2000). XML is a markup language for handling structured data that extends and formalizes HTML. Additionally, the W3C Consortium have developed simple object access protocol

(SOAP) (W3C Consortium, 2003a), a communication protocol entirely based on XML. Nowadays, SOAP has become the most ubiquitous protocol for applications that interact with Web services.

One of the most-used languages for Web service representation is WSDL (W3C Consortium, 2003b). WSDL is an XML-based language that allows developers to create Web service descriptions as a set of functions that operate with SOAP messages. From a WSDL specification, a program can autonomously determine the services of a Web site and how to invoke and use these services. As a complement to WSDL, universal description, discovery and integration (UDDI) (OASIS Consortium, 2004) has been developed. UDDI provides mechanisms for publishing and searching service descriptions written in WSDL. The weak point of this infrastructure is that it does not take into account the semantics of each service. Some languages for solving these problems are resource description framework (RDF) (W3C Consortium, 2004) and ontology Web language (OWL) (Horrocks, 2005). RDF can be used to bind attributes to Web accessible resources and to link these resources between them. OWL extends RDF support for high-level resource descriptions. A step towards the creation of a standard service ontology is OWL-S (DAML Coalition, 2006).

Related Work

Web services are a suitable model for the systematic interaction of Web applications and the integration of legacy platforms and environments. A few years ago, technology for supporting automatic interactions between Web applications has started to emerge, first with the development of automated e-commerce and B2B transactions, and more recently, with the creation of large-scale resource sharing infrastructures for grid computing (Foster & Kesselman, 2003).

Grid computing is mainly centered around the *computational Grid* concept (Foster et al., 2001): A distributed computing infrastructure whose goal is to provide safe and coordinated computational resource sharing between organizations. In this context, Web services play a fundamental role, since they give a satisfactory solution to the problem of heterogeneous systems integration, which is a fundamental requirement of almost every grid-based application.

Another good solution to tackle down heterogeneous systems integration problems in grid applications is to employ mobile agents. This technology, besides being heterogeneous by nature, also makes for efficient use of hardware and software resources, due to location awareness capabilities of mobile agents. Moreover, mobile agent technology represents a powerful paradigm for developing heterogeneous applica-

tions that use Web resources, especially when these applications are designed as a set of agents interacting with the various services offered by the Semantic Web.

Many agent-based tools for programming the Semantic Web can be found in the literature. Some examples relevant to our work are ConGolog (McIlraith et al., 2001), IG-JADE-PKSLib (Martínez & Lespérance, 2004) and CALMA (Chuah et al., 2004). Roughly speaking, these tools aim at simplifying the development of intelligent agents which interact with Web services. However, ConGolog and IG-JADE-PKSLib do not support agent mobility. Furthermore, neither IG-JADE-PKSLib nor CALMA provide facilities for handling ontological information about Web services. In addition, IG-JADE-PKSLib agents present serious performance and scalability problems, as we have shown in (Mateos et al., 2006a).

A lot of work has also been concerned with providing frameworks and platforms for materializing Web services as mobile agents. For instance, Ishikawa et al. (2004) propose a framework for implementing mobile Web services, which run on a JAVA-based mobile agent platform named Bee-Gent. However, Bee-Gent lacks support for common agent requirements such as knowledge representation, reasoning and high-level communication. On the other hand, proposals like (Bellavista et al., 2005) and (Adaçal & Bener, 2006) are more focused on interfacing mobile agents with Web services standard technology. The first one is concerned with achieving interoperability of legacy mobile agent middlewares, whereas the latter one aims at adapting server-side Web services for mobile devices by using mobile agents. In either case, it is not clear to what extent implementation of mobility and service-agent interaction functionality is done automatically by the framework. Finally, another interesting work is the one presented in (Zahreddine & Mahmoud, 2005), where an agent-based approaches to leverage composite mobile Web services to mobile devices is proposed.

As we will explain, the utmost goal of MoviLog is to simplify the construction of Web-aware mobile agents. Unlike previous work, MoviLog exploits the notion of RMF for seamlessly integrating mobile agents with Web resources, while providing scalability, flexibility and ease of programming. The next section takes a closer look at the MoviLog language.

MoviLog

MoviLog (Zunino et al., 2002) is a platform for building intelligent mobile agents, based on a strong mobility model (Fuggetta et al., 1998), where agents' execution state is transferred on migration. MoviLog is an extension of JavaLog (Amandi et al., 2005), a framework for agent-oriented programming.

MoviLog takes advantage of the benefits of both Java and Prolog since it is built as an extension of JavaLog. At one hand, Prolog is an adequate alternative for representing agents' mental states, and building reasoning algorithms (Amandi et al., 2005). On the other hand, Java has good features for supporting low-level code migration, such as platform independence, multi-thread support and object serialization (Wong et al., 1999).

In order to provide mobility across sites, each MoviLog host has to execute a MARlet (mobile agent resource). A MARlet is a Java servlet (Hunter & Crawford, 2001) that encapsulates a Prolog inference engine and provides services to access it. In this way, a MARlet represents an execution environment for mobile agents, or *brainlets* in MoviLog terminology. Additionally, a MARlet is able to provide intelligent services under demand, such as modifying the content of the main inference engine logic database or perform logic queries. In this sense, a MARlet can be used as an inference server for agents and external Web applications.

Besides providing basic strong mobility primitives to Brainlets, the most important aspect of MoviLog is the notion of reactive mobility by failure (RMF) (Zunino et al., 2005), a mechanism not exploited by any other tool for programming mobile agents. This mechanism states that when certain predicates previously declared in the code of a Brainlet fail, MoviLog transparently moves the Brainlet and its execution state to another site that contains definitions for that predicate, thus making local use of those definitions later. For instance, the following code implements a Brainlet whose behavior is programmed by the rules included in the CLAUSES section:

```
PROTOCOLS
  protocol(a, 2)
CLAUSES
  b(Y):- ...
  ?- a(X,Y), b(Y).
```

The section "Protocols" states that every clause whose functor is *a* and arity is 2 will be treated by RMF.[1] In this way, when the evaluation of *a*(*X*,*Y*) fails, the agent will be transferred to a site that contains definitions for the clause. Then, in case of a successful evaluation at the remote site, the algorithm will try to solve *b*(*Y*), according to the standard Prolog evaluation algorithm; otherwise the evaluation of *?-* will fail, due to the failure of *a*(*X*,*Y*).

The next example presents a simple brainlet whose goal is to first collect temperature values generated at different measurement sites, and then calculate the average of these values. Each measurement point is represented by a MoviLog site running a process which store on a regular basis its measure *T* in the local Prolog database, in the form of a *temperature*(*T*) predicate. The code that implements the brainlet is:

```
PROTOCOLS
 protocol(temperature, 1)
CLAUSES
 average(List, Avg):- ...
 getTemp(Curr, List):- temperature(T), thisSite(S),
                M = measure(T, S), not (member(M, Curr)),
                getTemp([M|Curr], List).
 getTemp(Curr, Curr).
 average(Avg):- getTemp([], List), average(List, Avg).
 ?- average(Avg).
```

The idea of the program is to force the brainlet to visit all the measurement sites, asking the temperature to each one of them, and then computing the average of those values. The potential activation point of RMF has been highlighted in the code. As the reader can see, the "Protocols" section defines that *temperature*(*T*) must be evaluated by RMF. As a consequence, if the evaluation of that clause fails at a site S (the brainlet has already obtained the measure) MoviLog will transfer the brainlet to a site that contains another temperature. The evaluation of *getTemp* will end successfully once all sites offering clauses *temperature*(*T*) have been visited.It is worth noting that the example shows two clear limitations of MoviLog. First, it only uses mobility for evaluating non-local clauses. This may cause performance problems and inefficient usage of system resources. Consider for instance the situation of a large brainlet that requires a small Prolog clause. Clearly, moving the clause to the site where the Brainlet executes requires less bandwidth usage than the opposite approach. In addition, protocols let the programmer to specify the points of the code that will be treated by RMF. Nevertheless, it is necessary to extend the current protocol mechanism to instruct RMF to use some remote invocation mechanism for accessing resources in addition to mobility. In the next section, we expose an approach for solving the mentioned problems.

MoviLog and Web Services

MoviLog is based on RMF, a novel mobility model that reduces the effort for developing mobile agents by automating decisions about mobility, such as when and where to migrate (Zunino et al., 2005). Despite the advantages RMF has shown, it is not adequate for developing Web enabled applications where a mix of mobility and remote invocation is required. This section shows an approach to overcome these issues.

Basically, RMF and its runtime support have been adapted to provide integration with the Semantic Web. This support enables MoviLog agents to interact with Web resources to perform their tasks, which makes MoviLog more useful as a mobile agent programming language. Nevertheless, to accomplish this adaptation the following problems were solved:

- **RMF extension:** It was necessary to extend the resource description mechanism used by MoviLog in order to describe Web resources and the way an agent accesses to a certain resource instance. In this sense, proper methods for accessing Web resources (apart from mobility) has been added, such as remote invocation, for the case of Web services, and copy of resources between sites, for the case of Web information retrieval.
- **Automated resource access:** MoviLog should provide an environment for agent execution which automates certain decisions related to resource access and, at the same time, let the programmer define policies for making these decisions. Custom decisions are made based on system metrics, such as network traffic, distance between sites, CPU load or available RAM at a site, among others. In this way, the programmer is able to specify *intelligent* decision mechanisms for accessing resources, thus potentially improving the usage of system resources.
- **Web Service semantics:** To achieve a truly automated interaction of agents and Semantic Web services, each agent has to understand the *meaning* of a Web Service. To do so, we have extended MoviLog to handle ontologies expressed in OWL.

Each one of the items exposed is essential for an effective integration of MoviLog to the Semantic Web. In the next three subsections, we will explain the approaches used in each case.

RMF Extension

In MoviLog, a protocol defines the format of the prolog clauses (i.e., functor and arity) which will trigger the migration of an agent when the site where the agent is currently executing does not contain a definition for any of those clauses. The protocol definition mechanism has been extended to describe more classes of resources that an agent might need to accomplish its goals. Particularly, a single prolog clause can be considered as a resource that agents access on a certain point of their execution. In fact, (Zunino et al., 2005) defines *failure* as the impossibility of an executing agent to obtain some required *resource* at the current site.

A MoviLog resource is composed of a name, and a set of properties, which vary on each resource (for example, functor and arity for a clause-like resource; user, password and database name for a database connection). As we saw, the programmer declares the need for accessing a resource (in a certain point of an agent's code) by adding a protocol into a special section of the program. This protocol contains the mentioned resource name and properties.

The first version of MoviLog (Zunino et al., 2002) proposes to move a brainlet every time an agent requests access to a resource unavailable at the current site. Although mobility is an effective method for accessing resources, performance of agents may suffer if migration is not performed in an intelligent way. For example, consider the case where the size of a brainlet is greater than the size of a requested resource. Clearly, it would be more convenient to get a copy of the resource from the remote site instead of moving the agent to that site. In this case the requested resource is a Web service, the proper access method is to remotely invoke the service, thus transferring only the (potentially small) service arguments and results. Finally, the interaction of an agent with a large remote database can be accomplished only by moving the agent to the remote site, and then locally interacting with the resource. In this case, database access by copy is unacceptable due to the great transfer cost over the network.

The ideas previously discussed show the need for adding extra mechanisms for accessing resources (apart from mobility) to MoviLog. As a consequence, performance can be improved by selecting the most adequate access method for the required resource (see section titled "Automated Resource Access"). Moreover, the methods that can be used for accessing a particular resource may depend on specific characteristics of it, such as its size, permissions, availability, and so forth. To solve this problem, each MoviLog resource is tagged with a type that allows agents to know what access methods can be used to obtain that resource.

Additional Mechanisms for Accessing Resources

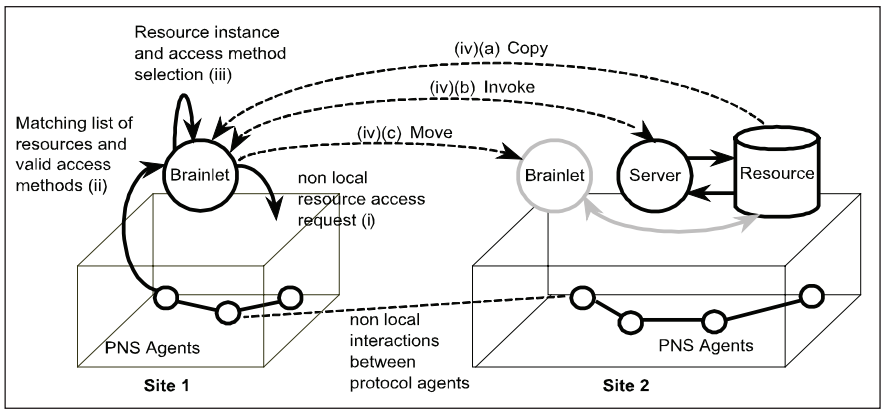
In RMF, every time a resource required by an agent is not present at the current site, RMF transfers the agent and its execution state to any site that contains definitions for that resource. This process is performed iteratively along the agent lifetime.

As we mentioned earlier, it is necessary to consider other resource access methods in addition to mobility to avoid performance problems in the execution of brainlets and to make the best usage of network resources. In this sense, the MoviLog evaluation algorithm has been redesigned to support a number of new methods for accessing non-local resources (Figure 1). The goal of this task was to decouple an agent resource access request from the selected method to obtain that resource. The three access methods defined for enhancing MoviLog are:

- **Move:** Moves the agent to the site owning the resource. This implies to serialize the agent execution state and then send it to the remote site. At the remote site, the agent is restored from its serialized form and its execution continues. Let us consider, for instance, the case of an agent that interacts many times with a large remote database. In order to better access the database, it is convenient to migrate the agent to the remote site and use the data locally, thus avoiding wasting valuable bandwidth and time caused by numerous remote interactions with the database.
- **Fetch:** Transfers a resource from a remote site to the local site by copying it to a shared repository accessible by the agent. For example, if an agent needs a sorting algorithm, the code implementing this algorithm can be copied from a remote site to the current location of the agent.
- **Invoke:** Accesses the resource by sending a request to a server agent located at a remote site, then waiting until the results are received, and finally resuming the normal execution flow. This is the proper method to access Web services in MoviLog. This works as follows: The agent sends the name of the service and input parameters to a remote server agent; then this agent locally invokes the service and returns the results back to the client.

Figure 1 shows the steps that MoviLog performs for accessing resources. First, based on a protocol that describes a resource, the algorithm asks the MoviLog platform the list of sites offering that resource. Second, an effective access to the resource must be carried out, first selecting a candidate site from the resulting list of the previous step, and then performing the specific retrieval operation (copy, remote request or agent migration). Third, the agent execution state is updated according to the new conditions after successful access to the resource.

Figure 1. MoviLog non-local resource access mechanism



The brainlet evaluation algorithm works as follows: After a failure, the agent queries the platform for sites offering the required resource, thus obtaining a list of properties (resource type, availability, size, etc.) of the matching resource instances. Taking this list as an input, the algorithm creates a list of pairs $L=<a,b>$, where a represents the information associated to a resource instance and b, the valid methods for accessing that instance. For example, the platform does not consider the *fetch* method to access a large database. Based on the list L, the following actions are performed:

- **Select the specific instance to access:** The algorithm selects from the input list the source site from where the resource will be retrieved. In other words, the algorithm selects an element from the list of instances, leaving the remaining items as backtracking points to ensure completeness.
- **Select the access method:** The algorithm selects the access strategy that best adapts to current execution conditions (site load, available memory, network traffic, etc.), particular characteristics of the application (many or few interactions with the same remote resource) or even custom policies specified by the programmer (see section titled "Automated Resource Access"). Depending on these factors, the platform decides the method for accessing the resource.

Note that, in Figure 1 a new type of agent named protocol name server (PNS) is introduced. Basically, PNSs are stationary agents (i.e., nonmobile) whose goal is to help brainlets to handle failures.

Each host capable of hosting brainlets has one or more PNS agents. PNSs are responsible for managing information about protocols offered at their owner's site, and for returning the list of resource instances matching a given protocol under demand. A site offering resources register with its local PNSs the protocols associated with these resources. As a consequence, PNS agents announce the new protocols by broadcasting this information to other MoviLog-enabled sites of the network.

The next section describes further details of the approach.

Resource Classification

In the previous section we described the way non-local resource access mechanisms operate. When an agent requests access to a non-local resource, the underlying platform builds a list of resource instances that match the agent's needs. This list is a sequence of pairs <a,b>, where a is the information of a matching instance, and b is the set of access methods to obtain a. At this point, the platform filters invalid access methods according to the resource type (Figure 2). This section is concerned with the different types of resources that MoviLog has to take into account in order to select whether to migrate a Brainlet, fetch a resource instance or invoke it.

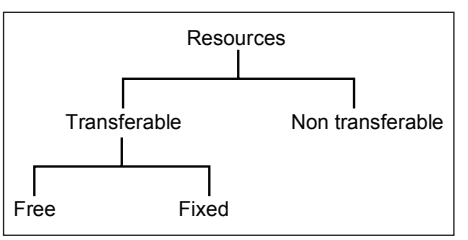
A resource can be classified as either *transferable* or *non-transferable*. A transferable resource can be freely copied from one site to another (e.g., a file or an environment variable), whereas a non-transferable resource remains in a site and cannot be transferred (e.g., a printer or a scanner). There are two kinds of transferable resources: *free* and *fixed*. Free resources can be moved across different sites; fixed resources represent data and devices whose transfer is non-viable (a large database) or undesired (private files, passwords, etc.). Unlike the former case, this last subdivision is done at the application level, and does not depend on the resource type at all.

In MoviLog, a Web service is a resource which represents a service invocation and access support through the Web. This support is composed of Java code that associates an executing Prolog clause requesting access to a Web service with the corresponding low-level SOAP request. In other words, such a Java code maps the service name and input parameters, which are Prolog data types, to their counterparts expressed as SOAP data types. Notice that this invocation support may not be present in every MoviLog site (i.e., every site is not capable of invoking Web services). Moreover, a MoviLog Web service is a non-transferable resource, since the Java code, which performs parameter mapping and low-level SOAP calls, is not copied between sites.

Prolog clauses represent free-transferable resources, since they are basically code that in most cases can be migrated. However, a MoviLog site may deny the transfer of a set of clauses; in this case, these clauses are viewed as fixed-transferable resources. The denial of a transfer could be as a result of one or more of the following reasons:

- If the estimated size of the set of clauses to transfer is too large, its transfer can be inefficient. For example, the transfer of a group of clauses that represents a long Prolog algorithm may not be allowed because of network traffic limitations.
- Some of the clauses have variables that are instantiated with non-serializable objects (i.e., values that are not translatable into a network-transferable format). In this case, the transfer of the resource is impossible.

Figure 2. Classification of resources accessible by mobile entities (Source: Vigna, 1998)



- The remote site has established security restrictions for the transfer of certain resources. Despite the set of clauses is free of the previous problems, there could be security limitations that yield in a denial of fetching-like operations on those clauses.

The MoviLog runtime system has been extended with built-in *classification policies* for dynamic categorization of resources, which are configured statically on each server. Thanks to this support, it is possible to specialize as much as is needed for the policy that categorizes a specific resource. At present, the basic MoviLog support for categorizing resources includes fixed policies, where the resource type is configured statically and do not vary along time, and variable policies, where the resource type is computed dynamically. In this sense, a MoviLog Web service resource has an associated fixed policy (i.e., the resource is always non-transferable), whereas a Prolog clause associated policy will determine the type dynamically (e.g., by running a code size estimation algorithm).

Automated Resource Access

The introduction of resource access policies to MoviLog evaluation algorithm bring some benefits that help automating agent execution. First, MoviLog can establish default policies for accessing resources in an efficient way. In addition, a flexible support for programming policies allows the agent developer to declare his own access decisions based on the requirements and characteristics of his application. The way MoviLog defines default access policies is similar to the way a programmer does (i.e., by picking the *best* resource from a list of candidate instances). The main difference is that programmers need some standard directives or commands to indicate to the platform what to do when accessing resources.

The basic elements upon which complex access rules are built are system metrics. MoviLog offers the programmer a number of prolog predicates that return the current value for a certain metric (CPU load, available memory, network traffic, proximity between sites, etc.). For example, the invocation to *freeMemory*(*Site*, *R*) executes instead a prolog predicate which instantiates *R* with the amount of free memory at *Site*. Based upon this predicate, the programmer is able to create more complex policies. This is the case, for example, of an access rule which retrieves a required resource from the site with the highest amount of available RAM.

MoviLog's approach for access policies specification has two major benefits. On one hand, Prolog's declarative nature represents a great flexibility for declaring rule-based policies. On the other hand, it is possible for an agent to modify its access policies in a dynamic and intelligent way, since each policy is in fact a Prolog rule contained in the agent's private knowledge base.

Profiling

Having measures about different aspects of system performance suggests the need that every site on the MoviLog network must be able to manage this information. In this sense, each MoviLog site is responsible for getting and maintaining up-to-date values of system metrics, and also providing a profiling service interface according to programmers' needs. In short, each site provides precise measuring mechanisms for the following metrics:

- **CPU load:** It means the percentage of CPU utilization at some site. It is a useful measure under certain circumstances. For example, if a given resource can be obtained by both *fetch* and *move* methods, and the local CPU load is twice as the CPU load of the site of the resource, move method could be used, thus providing a simple strategy for balancing load across sites.
- **Remote CPU load:** It represents the CPU load at some remote site. Clearly, it is very difficult to have up-to-date values of CPU utilization from other sites, due to the highly dynamic nature of this metric and the potential delays on information transfer over the network. To solve this problem, every MoviLog site broadcast, on a regular basis, average information about CPU local utilization.
- **Free RAM:** It represents the percentage of RAM available for allocation at some site. Furthermore, remote free RAM metric (i.e., available memory at a remote site) is computed similarly to remote CPU load.
- **Transfer rate:** This metric means the current speed of the outcoming communication links, and can be obtained by using typical operating system commands (e.g., Unix *ping*). Having measures about network transfer speed is crucial to decide whether migrating agents or fetching resources, or even invoking services remotely. In most cases, this latter will generate the least amount of data to be transferred, consequently improving agent execution performance.
- **Communication reliability:** It represents the percentage of the information lost during transfer through the network.
- **Proximity between sites:** It is a metric mostly related to network topology, and computes the distance (measured in number of hops) from the local host to some node of the network. Basically, this metric gives an approximated value of the transfer delay of information over the network.
- **Number of executing agents:** It is closely related to CPU load, especially when each agent is doing CPU intensive work (i.e., it is neither blocked nor waiting for some notification or signal). This value, which can be obtained by asking the local agent execution engine, gives an approximate idea of the CPU

and memory utilization. Also, the number of executing agents at some site can be estimated in a similar way in which remote CPU load is computed.

- **Agent size:** This metric represents the estimated size in bytes of the allocated memory space for an executing agent (i.e., the allocated RAM for the agent code and execution state). It is a useful metric to help making decisions on whether or not to migrate an agent, depending of its size and the current network transfer speed. Similarly, this metric can be applied to estimate the size of a resource. In this way, complex access policies that decide whether to migrate an agent or fetch a resource based on each one's size can be declared.

Table 1 summarizes the metrics described previously. The programmer is allowed to use predefined predicates, which compute the metrics in order to declare complex decision rules for accessing resources. These rules have to be included in a special section of the Brainlet code with a unique name and the decision behaviour concerning what instance to select, given a candidate list, and what access method to apply to get that instance. In order to activate a rule, one or more declared protocols have to be associated with the rule name. Then, when the access to the protocol fails, MoviLog searches for similar resources and picks one of them according to the decision made by the rule configured for the protocol.

Web Services Semantics

The effective use of Web services requires both agents and applications to be integrated with legacy Web infrastructure. Nevertheless, the implementation of this integration would be impossible, unless those services are represented in an agent-understandable semantic language (Kagal et al., 2003).

Table 1. System metrics supported by MoviLog

Category	Metric name
Processor and memory	CPU load
	Remote CPU load
	Free RAM
	Remote free RAM
Network	Transfer rate
	Communication reliability
	Proximity between sites
Agent-related	Number of executing agents
	Agent size

A solution to the problem of interoperation between intelligent agents and Web services is to use structured descriptions of the concepts included in a service. These descriptions specify *what* functionality the service provides but not *how* to do it. In this way, agents interact with Web services at the application level, by understanding abstract descriptions that are enough to express services functionality and intended purpose.

The use of Web content description languages offers some advantages for functionality-based discovery of Web services, which permits automatic interaction between agents and services. Service discovery is an inherently semantic problem, because it must abstract the superficial differences between the offered services and the requested ones, so semantic similarities of these two can be recognized. To cope with this problem, utilization of OWL-S over UDDI has recently been proposed, thus creating a powerful infrastructure for Web Service discovery based on the functionality they provide (Srinivasan et al., 2004; Srinivasan et al., 2006).

In order to provide a truly automated interaction with the Semantic Web, each MoviLog agent has to understand the exact meaning of a Web service. To make this possible, we have extended MoviLog to handle ontologies written in OWL Lite, a dialect of OWL for metadata annotation. The most interesting aspect of OWL Lite for our work is that it is easily translatable to Prolog, since it has description logic equivalent semantics, which is a decidable fragment of first-order logic (Baader et al., 2003). Consequently, it is possible to make automatic inferences from a translated OWL Lite ontology and using traditional theorem provers. The reader should remember here that MoviLog agents identify resources through protocols, which are Prolog rules, thus inferences with respect to an OWL Lite ontology expressed as prolog rules can be done.

From this line of research we have already obtained encouraging results. Particularly, we have integrated MoviLog with Apollo (Mateos et al., 2006b), an infrastructure for semantic matching and discovery of Web services. Apollo includes a prolog-based reasoner implemented as a set of rules for computing semantic likeness between OWL Lite-annotated services. This support is used by MoviLog agents in order to determine the set of Web services which best suit their service request.

An Example

In this section we describe in detail an application coded in MoviLog. The application consists of a travel agent whose responsibility is to arrange an itinerary across a number of cities, making the necessary bookings for hotel rooms and airplane tickets to complete the overall trip. The application scenario is situated at a tourism company that sells different tourist packages and manages all these sellings with a

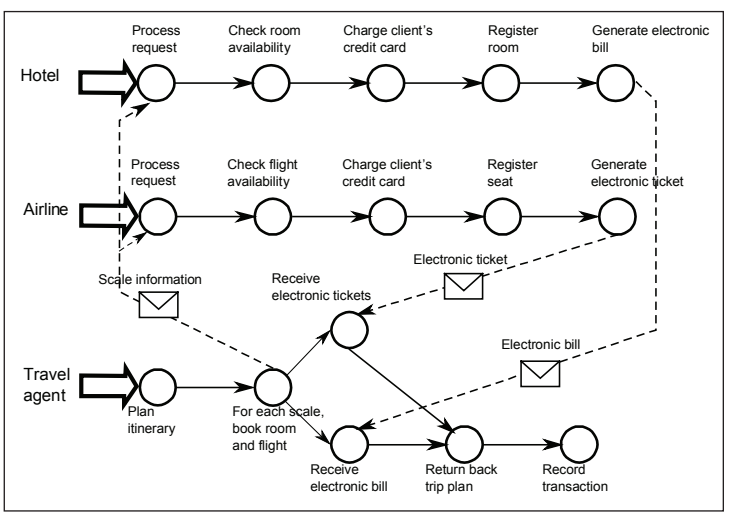
mobile agent-based system. Every time a client wishes to buy a package, an agent is asked to plan an adequate itinerary between the requested origin and destination, along with the corresponding hotel rooms and airplane tickets reservations. It is assumed that both hotel and airline companies involved in the process provide support for Web services for booking rooms and tickets, respectively.

A high level view of the tasks performed by each actor involved in the selling process is shown in Figure 3. Upon receipt of a new request, the travel agent construct an itinerary based on the client preferences such as the desired intermediate cities and the number of days he plans to stay at each one of them. For every stopover of the resulting itinerary, the agent books a flight on any airline according to planned dates in advance. Similarly, the agent books the necessary hotel rooms. Finally, the entire schedule and reservations are returned back to the client, and the transaction is recorded in a database.

From Figure 3 we can see that the travel agent interacts with two kind of resources along its lifetime. First, the agent books rooms and tickets by invoking *Web services* published by hotel and airline companies. Once the request has been successfully served, the agent must register the transaction in a specific company database, so it will need to establish a *connection* to the mentioned database and then store the data. The words in italic represent resources that the agent needs to achieve its goal. In others words, when an executing agent fails to access a *Web Service* or a *database connection* at the current site, RMF will automatically handle this failure, trying to locate and access similar resources owned by other sites.

The code implementing the travel agent is shown in the "protocols" portion. For simplicity, date and time-related information are not taken into account during the

Figure 3. A tourist package construction process



reservation process. For the same reason, the program does not handle exceptions that might be thrown by a Web service, such as insufficient credit card balance, unavailability of rooms or tickets, and so on.

PROTOCOLS

```
protocol('data-base-connection,' [dbName('sellings')], 'dbPolicy').
protocol('web-service,' [name('bookFlight')], 'wsPolicy').
protocol('web-service,' [name('bookHotelRoom')], 'wsPolicy').
```

POLICIES

```
accessWith('dbPolicy,' [ResourceID, Site], 'move', _, 'move'):-
  CPULoad(Site, Load), Load <= 50.
sourceFrom('wsPolicy', [ID1,Site1], [ID2,Site2], Result):-
  leastCPUUsage(Site1, Site2, Result).
```

CLAUSES

```
leastCPUUsage(Site1, Site2, Site1):-
  CPULoad(Site1, Load1),
  CPULoad(Site2, Load2),
  Load1 <= Load2, ! .
leastCPUUsage(_, Site2, Site2).
scheduleCircuit(Origin, Destination, Cities, Schedule):- ...
bookFlightsAndHotelRooms([Destination], [], 0).
bookFlightsAndHotelRooms([C1Info,C2Info|Cities],[SchInfo|Sch],Cost):-
  C1Info = city-info(City1, DaysAtCity1),
  C2Info = city-info(City2, DaysAtCity2),
  web-service([name('bookFlight'), input([from(City1),to(City2)])], FlightInfo),
  web-service([name('bookHotelRoom'),
                input([location(City2),days(DaysAtCity2)])], HotelInfo),
  FlightInfo = [cost(TicketCost), ticket(Ticket), airline(Airline)],
  HotelInfo = [cost(RoomCost), number(RoomNumber), hotel(Hotel)],
  ScheduleInfo = [FlightInfo, HotelInfo],
  bookFlightsAndHotelRooms([City2Info—Cities], Schedule, SubCost),
  TempCost is SubCost + TicketCost,
  Cost is TempCost + RoomCost.
storeTransaction(Schedule, Cost):-
  data-base-connection([dbName('sellings')], Connection),
  storeTransaction(Schedule, Cost, Connection),
  closeDBConnection(Connection).
storeTransaction(Schedule, Cost, Connection):- …
```

```
closeDBConnection(Connection):- ...
? -doService(Origin, Dest, Cities, Schedule, Cost):-
  scheduleCircuit(Origin, Dest, Cities, Circuit),
  bookFlightsAndHotelRooms(Circuit, Sch, Cost),
  storeTransaction(Sch, Cost).
```

The code is composed of three sections. The section "Protocols" declares the agent protocols (i.e., descriptions) of those resources whose failure may trigger RMF. Three protocols have been declared: A connection to the company's database (whose name is *sellings*), a service for flight booking (*bookFlight*) and a service for hotel booking (*bookHotelRoom*). Notice that the last two protocols are declared in a way that they are independent from the particular airline or hotel providing those services.

The "Policies" section defines the strategies for accessing the resources. Two policies for accessing database connections and booking services, namely *dbPolicy* and *wsPolicy*, have been declared. The first one states that an agent will move to any site *S* providing database connections only if the CPU load at *S* is less or equal than 50%, otherwise any other alternative method for accessing the resource will be used. On the other hand, *wsPolicy* adds restrictions about the source to be contacted for invoking Web services. In this case, given two different sites with Web services invocation support, the one with the least current CPU usage will be chosen. It is worth mentioning that *CPULoad* is a profiling predicate that returns the current CPU usage at some site. Both policies could have been implemented by using other profiling predicates such as those related to the performance and transfer rates of the outcoming communication links.

Finally, the section "Clauses" implements the travel agent's behaviour as a set of Prolog rules. The rule *?-doService* constructs an itinerary across different cities (code not shown), then makes the reservations, and finally stores the results. The lines of code in bold are potential activation points of RMF. When the execution of the code reaches a clause whose functor and first argument match any declared protocol, that clause is not further evaluated according to the standard prolog evaluation algorithm. In such case, the clause is interpreted as a request to access some resource that is described by the mentioned protocol.

At this point, the reader may be wondering how MoviLog can access a resource from just a simple Prolog call. In other words, how MoviLog maps a call of the form *resource-name*([$property_1$, $property_2$, ..., $property_n$], arg_1, arg_2, ..., arg_m) to the desired resource instance. Every protocol published by a MoviLog site is composed of a name, a unique identifier, a list of properties and a local Prolog clause, the last of which implements the resource access functionality. In the case of a Web service resource, this clause performs the service invocation itself and then returns the results back, whereas for a database connection resource its associated clause will create an object that represents that connection (e.g., a JDBC object), check

if the agents have permissions, and finally return the connection. In other words, each one of these kinds of clauses implements the access tasks that depend on the type of the specific resource instance described by the associated protocol. Also, each protocol defines the way a runtime instantiated Prolog clause maps to its resource access clause. Therefore, effective resource access is fully transparent to the programmer, due to the fact that the MoviLog platform binds at runtime each argument arg_i of the current executing clause with the results given by the specific resource access clause.

Conclusion and Future Work

Web services enable the construction of new types of applications characterized by their ability to interact with Web-accessible services through standard protocols. The extensions of Web services with semantics aim at realizing a dream where programs autonomously use the vast amounts of resources present on the Web. In this complex, rich computational environment, intelligent mobile agents will have a fundamental role due to their capacity to infer, learn, act and move.

Our research aims at providing tools for building intelligent agents that autonomously interact and live within the WWW. JavaLog is a programming language that supports the basic bricks for constructing intelligent agents. MoviLog adds flexible and usable support for reactive mobility to JavaLog. We have described in this chapter an approach for integrating MoviLog with Web services.

The main difference between MoviLog and others platforms for mobile agents is twofold. First, its support for reactive mobility by failure dramatically reduces development effort by automatizing agent and resource mobility decisions. Second, it permits to transparently invoke Semantic Web services, which enables the construction of Semantic Web-aware mobile agents with little coding effort.

We are currently working on improving the way MoviLog sites manage protocol and profiling information in order to achieve better scalability. Until now, we have experienced some problems when the number of sites or even the amount of interchanged information increase. A step towards addressing this issue is GMAC (Gotthelf et al., 2005), a multicast-based communication protocol specially designed for MoviLog. It is important to note that this idea is still being explored.

Finally, although the description of the RMF support for agent and resource mobility were circumscribed to MoviLog, the mechanism can be further applied to other programming languages and contexts. In fact, we have implemented a prototype RMF-based mobile agent platform for the Java language. The middle-term goal from this line of research is to isolate as much as possible RMF from the programming

language involved, so to provide mobility at the middleware level in other contexts besides the Semantic Web, such as those related to grid computing.

References

Adaçal, M., & Bener, A. B. (2006). Mobile Web services: A new agent-based framework. *IEEE Internet Computing*, *10*(3), 58-65.

Amandi, A., Campo, M., & Zunino, A. (2005). JavaLog: A framework-based integration of java and prolog for agent-oriented programming. *Computer Languages, Systems and Structures*, *31*(1),17-33.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., & Patel-Schneider, P. (2003). *The description logic handbook: theory, implementation, and applications.* Cambridge University Press.

Bellavista, P., Corradi, A., & Monti, S. (2005). Integrating Web services and mobile agent systems. In *Proceedings of the 1st International Workshop on Services and Infrastructure for the Ubiquitous and Mobile Internet (SIUMI) (ICDCSW 2005)* (Vol. 3, pp. 283-290).

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The Semantic Web. *Scientific American*, *284*(5), 34-43.

Bramer, M. (2005). *Logic Programming with Prolog*. New York: Springer-Verlag.

Chuah, S. H, Loke, S. W., Krishnaswamy S., & Sumartono, A. (2004, July 20). CALMA: Context-aware lightweight mobile BDI agents for ubiquitous computing. In *Proceedings of the Workshop on Agents for Ubiquitous Computing (UbiAgents 2004)*. NY: Morgan-Kaufmann Publishers.

Curbera, F., Nagy, W. A., & Weerawarana, S. (2001). Web services: Why and how. In *Proceedings of the Workshop on Object-Oriented Web Services (OOPSLA 2001)*, Tampa, FL. ACM Press.

DAML Coalition. (2006). *OWL-S 1.2 pre-release.* Retrieved from http://www.daml.org/services/owl-s

Foster, I., Kesselman, C., & Tuecke, S. (2001). The anatomy of the grid: Enabling scalable virtual organization. *The International Journal of High Performance Computing Applications*, *15*(3), 200-222.

Foster, I. & Kesselman, C. (2003). *The grid 2: Blueprint for a new computing infrastructure*. Morgan-Kaufmann Publishers.

Fuggetta, A., Picco, G. P., & Vigna, G. (1998). Understanding code mobility. *IEEE Transactions on Software Engineering*, *24*(5), 342-361.

Gotthelf, P., Mendoza, M., Zunino, A., & Mateos, C. (2005, September 1-2). GMAC: An overlay multicast network for mobile agents. In *The 34th JAIIO Proceedings of the VI Argentine Symposium on Computing Technology (AST)*, Rosario, Santa Fé, Argentina.

Hendler, J. (2001). Agents and the Semantic Web. *IEEE Intelligent Systems*, *16*(2), 30-36.

Hunter, J., & Crawford, W. (2001). *Java servlet programming*. O'Reilly & Associates, Inc.

Horrocks, I. (2005). OWL: A description logic based ontology language. In P. Beek (Ed.), *Principles and Practice of Constraint Programming (CP 2005)* (LNCS 3709, pp. 5-8). Springer.

Ishikawa, F., Yoshioka, N., Tahara Y., & Honiden, S. (2004). Behavior descriptions of mobile agents for Web services integration. In *Proceedings of the IEEE International Conference on Web Services* (pp. 342-349).

Kagal, L., Perich, F., Chen, H., Tolia, S., Zou, Y., Finin, T., et. al. (2003). Agents making sense of the Semantic Web. In W. Truszkowski, C. Rouff, & M. G. Hinchey (Eds.), *Innovative concepts for agent-based systems* (LNCS 2564, pp. 417-433). Springer.

Kotz, D., & Gray, R. S. (1999). Mobile agents and the future of the Internet. *ACM Operating Systems Review*, *33*(3), 7-13.

Lange, D. B., & Oshima, M. (1999). Seven good reasons for mobile agents. *Communications of the ACM*, *42*(3),88-89.

Martínez, E., & Lespérance, Y. (2004, July 19-23). IG-JADE-PKSlib: An agent-based framework for advanced Web service composition and provisioning. In *Proceedings of the Workshop on Web Services and Agent-Based Engineering* (pp. 2-10). NY: Morgan-Kaufmann Publishers.

Mateos, C., Zunino, A., & Campo, M. (2006a). Extending MoviLog for supporting Web services. *Computer Languages, Systems and Structures* (in press). Elsevier Science.

Mateos, C., Crasso, M., Zunino, A., & Campo, M. (2006b). Adding Semantic Web services matching and discovery support to the MoviLog platform. In M. Bramer (Ed.), *Artificial intelligence in theory and practice, IFIP International Federation for Information Processing*. Springer.

McIlraith, S., Son, T. C., & Zeng, H. (2001). Semantic Web services. *IEEE Intelligent Systems, Special Issue on the Semantic Web*, *16*(2),46-53.

Milojicic, D., Douglis, F., & Wheeler, R.(1999). *Mobility: Processes, computers, and agents*. Reading, MA: Addison-Wesley.

OASIS Consortium. (2004). *UDDI, Version 3.0.2*. Retrieved from http://uddi.org/pubs/uddi_v3.htm

Srinivasan, N., Paolucci M., & Sycara, K. (2004). An efficient algorithm for OWL-S based semantic search in UDDI. In J. Cardoso & A. Sheth (Eds.), *Semantic Web Services and Web Process Composition: Proceedings of the First International Workshop (SWSWPC 2004)*, San Diego, CA (LNCS 3387, pp. 96-110). Springer.

Srinivasan, N., Paolucci M., & Sycara, K. (2006). Semantic Web service discovery in the OWL-S IDE. In *Proceedings of 39th Hawaii International Conference on Systems Science* (Vol. 6, p. 109b).

Tripathi, A. R., Karnik, N. M., Ahmed, T., Singh, R. D., Prakash, A., Kakani, V., et al. (2002). Design of the ajanta system for mobile agent programming. *Journal of Systems and Software*, *62*(2), 123-140. Elsevier Science.

Vaughan-Nichols, S. J. (2002). Web services: Beyond the hype. *Computer*, *35*(2), 18-21.

Vigna, G. (1998). *Mobile code technologies, paradigms, and applications*. PhD thesis, Politecnico di Milano, Milano, Italy.

W3C Consortium. (2000). *Extensible markup language (XML), Version 1.0* (W3C recommendation, 2nd ed.). Retrieved from http://www.w3.org/TR/2000/REC-xml-20001006

W3C Consortium. (2003a). *SOAP: Primer, Version 1.2, Part 0* (W3C recommendation). Retrieved from http://www.w3.org/TR/2003/REC-soap12-part0-20030624/

W3C Consortium. (2003b). *Web services description language (WSDL), Version 1.2* (W3C working draft). Retrieved from http://www.w3.org/TR/2003/WD-wsdl12-20030303/

W3C Consortium. (2004). *RDF primer* (W3C recommendation). Retrieved from http://www.w3.org/TR/rdf-primer

Wong, D., Paciorek, N., & Moore, D. (1999). Java-based mobile agents. *Communications of the ACM*, *42*(3), 92–102.

Zahreddine, W., & Mahmoud, Q. (2005). An agent-based approach to composite mobile Web services. In *Proceedings of the 19th International Conference on Advanced Information Networking and Application* (pp. 189-192). IEEE Computer Society.

Zunino, A., Campo, M., & Mateos, C. (2002). Simplifying mobile agent development through reactive mobility by failure. In G. Bittencourt & G. Ramalho (Eds.), *Advances in Artificial Intelligence: Proceedings of the 16th Brazilian Symposium on Artificial Intelligence (SBIA 2002)*, Brazil (LNCS 2507, pp. 163-174). Springer.

Zunino, A., Campo, M., & Mateos, C. (2005). Reactive mobility by failure: When fail means move. In G. Tayi & S. S. Ravi (Eds.) *Information Systems Frontiers, Special Issue on Mobile Computing and Communications: Systems, Models and Applications* (pp. 141-154). Kluwer Academic Publishers

Endnote

[1] For a comprehensive introduction to Prolog see http://www.coli.uni-saarland.de/~kris/learn-prolog-now