Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial
ISSN: 1137-3601
revista@aepia.org
Asociación Española para la Inteligencia Artificial
España

Belloni, Edgardo; Campo, Marcelo
BrainLets: Dynamic Inferential Capabilities for Agent-based Web Systems
Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial, vol. 5, núm. 13, 2001, pp. 108-114
Asociación Española para la Inteligencia Artificial
Valencia, España

BrainLets: Dynamic Inferential Capabilities for Agent-based Web Systems

Edgardo Belloni Marcelo Campo

ISISTAN Research Institute - UNICEN University
Campus Universitario - Paraje Arroyo Seco
Tandil (B7001BBO), Buenos Aires, Argentina
{ebelloni, mcampo}@exa.unicen.edu.ar

Abstract

This article presents Brainlets, a new mechanism designed to enhance the functionality of web servers with inferential capabilities. Brainlets are mobile Prolog modules supported by an extension to JavaLog virtual machine that enables a strong mobility model. This support is enabled in web servers trough specialized servlets, called MARlets, which provide the JavaLog inference machine. BrainLets can migrate among different hosts in order to meet other agents, to access to resources provided there, or to provide intelligent services under demand.

Keywords: Intelligent and mobile agents; Multi-paradigm languages; Agent-based web systems.

1. Introduction

The widespread use of computers and their connectivity, particularly the World Wide Web and the Java programming language, have provided a new influx in the research, development, and deployment of agents.

A particular motivation for the use of agents is the huge amount of information available on the Internet. Agents have a significant potential looking for information, filtering it, and extracting it from different sources. The ability to represent and act on behalf of the user represents a crucial capability of agents and provides enormous potential for their deployment.

This potential, however, in the near past has been limited due to the lack of appropriate tools to support the flexible development of agents working on web sites. Particularly, the Java platform, although it is very powerful, it is not enough to develop agents showing an intelligent and flexible behavior.

We had to deal with this situation during the development of a project concerning the design and implementation of a multi-agent system in the socialware[1] domain. This multi-agent system involves a network of web sites placed in different cities and countries, each one representing a virtual community composed by common users and members of diverse organizations. The web sites provide several services supporting the different social activities carried out by the members of these communities. The services are based on personal preferences and their development demands a complex engineering dealing with dynamically changing requirements and involves software agents

[1] *Socialware: multi-agent systems developed in order to assist in various social activities on network communities [Hattori et al. 99].*

with characteristics such as autonomy, social ability, reactiviness and proactiviness.

In this context, we choose a multi-paradigm approach for the development of web sites based on intelligent and mobile agents. The approach integrates both object-oriented and logic paradigms for the design and programming of agents. It is essentially based on *JavaLog* [Amandi et al. 99][Zunino et al. 01], a programming language that integrates both Java and Prolog. Through this support software agents can be developed as having certain mental attitudes, beliefs, desires and intentions, which represent, respectively, their informational, motivational and deliberative states. In this sense, an agent can be completely specified by the events that it can perceive, the actions it may perform, the beliefs it may hold, the goals it may adopt, and the plans that give rise to its intentions.

Additionally, this support integrates mobility services that enable logic-based agents, called *BrainLets*, to migrate to another host in order to meet other agents or to access to services and resources provided there. A BrainLet is a JavaLog program that can autonomously move among different hosts following a strong mobility model. Through the architecture supporting BrainLets a web server can be extended with inference processing capabilities.

Major advantages of this multi-paradigm approach are seen developing agents that require an intelligent and flexible behavior. Reducing expensive global communication costs by moving the computation to the data sources and distributing complex computations onto several hosts, which could be heterogeneous, is other worth mentioning benefit. Additionally, Brainlets can be used to provide web servers with more intelligent services that can be used by common web clients.

The article is structured as follows. The following section analyzes the rationale and motivations for our multi-paradigm approach to develop stationary and mobile agents that support an intelligent behavior. Section 3 introduces BrainLets, mobile logic-based agents in our approach, and the basic software architecture necessary to support them. Section 4 presents the details of the implementation support necessary to extend web servers functionality with logic modules. Finally, in Section 5, some concluding remarks are delineated and options for future work are discussed.

2. Intelligent and Mobile Agents: A Multi-paradigm Approach

In a broad sense, an agent is any computer program that acts on behalf of a (human) user. In this context, a mobile agent is a computer program, which represents a user in a computer network, and it is capable of migrating autonomously from node to node, to perform some computation on behalf of the user [Karnik and Tripathi 98].

Object-oriented languages have characteristics that partially satisfy the programming requirements of agents. Certainly, Shoham [Shoham 93] has introduced the term agent-oriented programming as a form of object-oriented programming. Concerning this point of view, an agent's state consists of beliefs, capabilities, choices, and similar notions, and the computation consists of interactions, such as informing, offering, accepting, rejecting and competing [Shoham 97]. In this context, an object can model an agent. The object's methods represent the agent's abilities (the agent's behavioral capabilities) and the object's variables of instance represent the agent's mental state (the agent's knowledge).

Multi-agent systems - i.e. systems dealing with stationary cognitive agents distributed on a computer network, which communicate one another in order to pursue a common goal - could benefit from integration with mobility services in many ways. For instance, groups of agents could be composed by a combination of heavyweight, cognitively complex agents dispatching tasks to lightweight agents pursuing only a few goals. These lightweight agents could proactively leave the group, move themselves to other hosts and possibly rejoin the group when they have accomplished their goals. Other scenario would show a relatively complex agent cloning itself in order to accomplish a specific goal on a specific host. Such capability is particularly interesting when an agent makes sporadic use of a valuable shared resource. There would be a number of benefits from such organizations. Efficiency, for instance, can be improved by moving lightweight agents performing queries over a large database to the host of the database itself. Response time and availability would improve when performing interactions over network links subject to long delays or interruptions of service.

Object-oriented languages also satisfy the programming requirements of mobile agents. Currently, Java is the most frequently used programming language for development of mobile agent systems. Aglets [Lange and Oshima 98],

Voyager [ObjectSpace Corp. 98] and Odissey [General Magic Corp. 98] are examples of Java-based mobile agent systems. The multi-platform support and the ubiquity of the Java virtual machine make Java particularly well suited for mobile agents' technology. The networking support of Java includes sockets, URL communication, and a distributed object protocol called remote method invocation (RMI). These features smooth the progress of dissemination of mobile agents throughout the Internet. Furthermore, Java has other features not found in any other language that directly support implementation of mobile agents [Wong et al. 99]. For instance, Java facilitates migration of agent's code and state via its class-loading and object serialization mechanisms.

Although the benefits that object-oriented languages provide for the agent-oriented programming in general, and particularly the Java language for the mobile agents programming, these languages have limitations to develop agents requiring an intelligent and flexible behavior. These limitations are demonstrated at the time of dealing with the agents' mental attitudes. In this case, different algorithms of inference, applied on the instance variables that represent the knowledge of the agent, must be implemented. These algorithms are, in general, hard to implement and they provide little scope of flexibility facing changes [Amandi et al. 99].

It is widely accepted that the logic-programming paradigm represents an appropriate alternative to manage mental attitudes due to its evident support to represent and infer relationships among mental attitudes such as intentions, goals and beliefs. Logic languages enable us to represent mental attitudes in declarative form through logic clauses. Deductive algorithms interpret the clauses that, in the context of the agent-oriented programming, give origin to agents' knowledge dependent reasoning.

Therefore a multi-paradigm approach, integrating both object-oriented and logic programming, represents a superior approach to support the development of intelligent agents systems. It aims to solve the limitations mentioned above, enabling stationary and mobile agents to manage complex mental attitudes.

3. BrainLets: Mobile Logic-based Agents

The fundamental concern of this work is the materialization of generic software architecture to support stationary and mobile agents based on JavaLog, a multi-paradigm programming language.

As was analyzed in section 2, multi-agent systems could benefit from integration with mobility services in many ways. In order to integrate mobility services to the JavaLog language, a mechanism that enables executing JavaLog's logic modules to migrate to another host using a strong mobility model was developed. This mechanism aims to enable an executing unit, in this case the brain component of an multi-paradigm agent, to move as a whole by retaining its executing state - i.e. retaining control and internal data associated to the executing unit, e.g., the program counter or the call stack - across migration. The migration is transparent, in that the brain component resumes its execution on the new host right after the instruction that triggered the migration.

JavaLog´s mobile logic modules are viewed in our approach as logic-based agents called BrainLets, able to move among different servers in order to perform some computation close to a valuable remote resource or to provide intelligent services under demand.

3.1. A Software Architecture Supporting BrainLets

Mobile agents require protected agent execution environments (or servers) that act like a dock station that accepts agents and provides native resources for them. A mobile agent server is responsible for executing agent code and providing primitive operations to agent programmers, such as those that allow agents to migrate, communicate, access host resources, etc. A logical network of agent servers implements the mobile agent system. Agent servers can be specialized to provide application-specific services.

Figure 2 shows the software architecture developed to implement a server of BrainLets. This is essentially founded on the generic architecture of Java-based mobile agents described in [Wong et al. 99]. The deployment diagram uses UML extensions proposed for mobility properties by Odell et al [Odell et al. 00].

This architecture prescribes that a multi-paradigm agent is a composite JavaLog object that supports mobility and can communicate with other agents, at level of its brain component. The architecture includes the following major components: an agent manager, an inter-agent communications manager, a security manager, an application gateway and a directory manager.

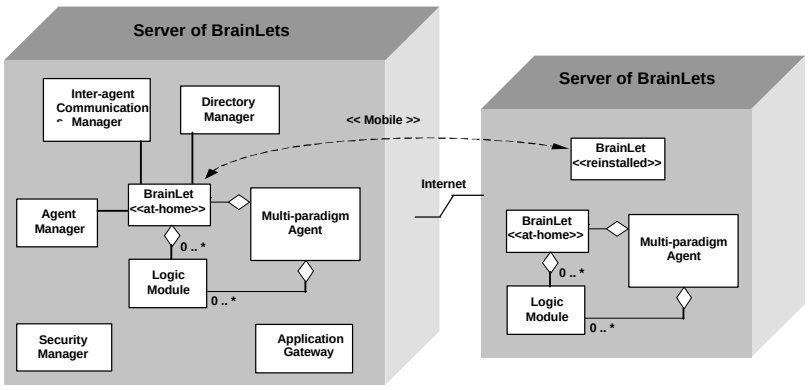

Figure 2 - Software architecture supporting BrainLets

The agent manager receives BrainLets for execution on the local host and sends BrainLets to remote hosts. The security manager authenticates the BrainLet before it is allowed to execute. Thereafter, the Java virtual machine automatically invokes the security manager to authorize any operations using system resources.

BrainLets may use the directory manager to identify the location of an application server and then migrate to the host on which the server is located. The application gateway provides a secure entry point through which agents can interact with application servers. An arriving mobile agent accesses to resident servers such as database servers through this gateway. The inter-agent communication manager facilitates communication among agents spread through the network.

A local proxy hides the remoteness of a BrainLet executing in a different space of address. A proxy is a surrogate for a BrainLet. It serves as a safeguard that protects the BrainLet from direct access to its public methods. The proxy provides location transparency for the BrainLet; that is, it can hide the BrainLet´s real location. If the actual BrainLet resides at a remote host, the proxy forwards the requests to the remote host and returns the result to the local host.

3.2. BrainLet's Life Cycle

As a specialized brain component, a BrainLet supports all the built-in predicates defined for a standard prolog interpreter, as well as those inter-agent communications mechanisms based on logic modules provided by JavaLog. A BrainLet extends a brain component with mobility services and provides support to offer its services to web applications.

Migration of a BrainLet between different hosts is illustrated in the Figure 3. When a BrainLet autonomously decides to migrate to another host, because of its informational, motivational and deliberative states, it calls the *move_to* built-in

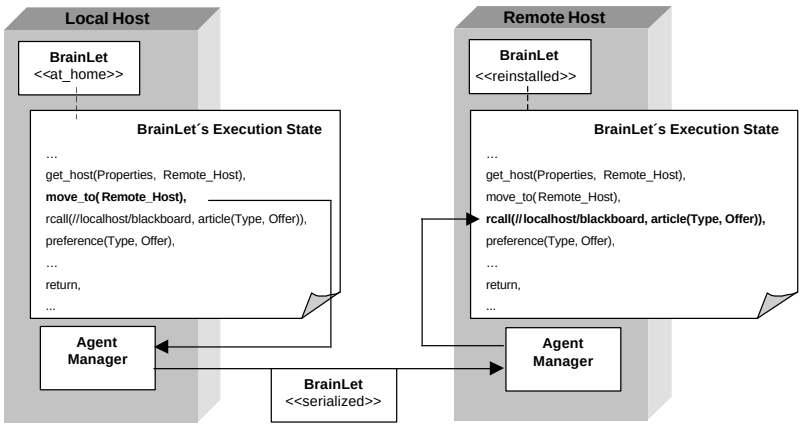

Figure 3 – Migration of a BrainLet between different hosts

predicate. Before transport, the agent manager in the local host serializes the BrainLet and its state - i.e. its knowledge base and code, current goal to satisfy, instantiated variables, backtracking points, etc. Then, the agent manager sends the serialized form to its counterpart on the destination host. Upon receipt of an agent, the agent manager in the remote host reconstructs the BrainLet and the objects it refers to, and then it resumes its execution. Eventually, after performing some computation, the BrainLet could return to the originating host calling the *return* built-in predicate.

4. Extending Web Server Services

One the main objectives of our work is the provision of more intelligent services to web sites via Brainlets. Normally, web servers provide means to extend their functionality through Servlets. A Servlet is a piece of Java code that a web server loads to handle client requests. It represents a pluggable extension to a server that enhances the server´s functionality. Servlets execute within the web server´s process space and they persist between invocations. The Servlet mechanism is the basis for developing extensions that support Brainlets, called MARLets (Mobile Agents Resources).

4.1. MARlets

A MARlet extends a web server with inference processing capabilities. Basically, it extends the Java servlets support encapsulating a specialized brain component and providing services to access it. In this way, a MARlet is able to provide more intelligent services under demand accepting requests, such as adding and deleting logic modules, activating and deactivating logic modules, and performing logic queries. In this sense, a MARlet offers inferential services to web applications or agents. Additionally, a MARlet represents a web dock for Brainlets.

The following example shows the code implementing a simple MARlet. This MARlet defines a brain component as an instance variable. As a specialized servlet, it redefines the service method in order to deal with the requests that receives from clients. Each time the server dispatches a request to a servlet, it invokes the servlet´s *service* method. The service method accepts two parameters: a request object and a response object. The input and output streams, which are manipulated for the MARlet to communicate with the MARlet´s clients, are obtained from these objects. The request object tells the servlet about the client's request, while the response object is used to return a response. After that, this method gets the operation to accomplish from the input stream. A lexical convention is defined in order to name the valid operations. For instance, ¨addCapability¨ defines an operation to add a logic module to the brain component. Then, the corresponding parameters for the operation are read from the input stream. Finally, the corresponding operation is delegated to the brain component.

```
//--- DefaultMARlet.javalog, v 1.0   ---//
import JavaLog.*; import javax.servlet.*;
...

public class DefaultMARlet extends GenericServlet {

    public MobileBrain brain;

    public DefaultMARlet() {
        brain = new MobileBrain();
        initKnowledge();
    }
    public MobileBrain brain() {
        return this.brain;
    }

/* Public methods implementing the services provided by
this simple MARlet. These methods are addCapability,
removeCapability, activeCapability, and answerQuery */

    public void addCapability(String idModule,
                              String knowledge){
        brain().addCapability(idModule, knowledge);
    }
    ...
    public boolean answerQuery(String query) {
        return brain().answerQuery(query);
    }

//---- The service  method ---//
public synchronized void service(ServletRequest req,
ServletResponse res)
.........
    String serviceRequested;
        serviceRequested = req.getParameter("service");

    if( serviceRequested.equals("addCapability") ) {

            String idModule = instream.readLine();
            String knowledge = instream.readLine();
            this.addCapability(idModule, knowledge);

        }else if( serviceRequested.equals("activeCapability") )
{

            String idModule = instream.readLine();
            this.activeCapability(idModule);

             ........
    } // End of service method
...
} //--- End of DefaultMARlet class ---//
```

A MARlet client sends requests to a MARlet in order to require inferential services. The following example partially shows the code of two very simple clients. The example involves a seller and a customer. The seller posts its article offers, and the

customer is able to select and buy different articles based on its preferences.

The simple seller sends a request to the MARlet to accept a logic module, which maintains offers for several electronic devices. The MARlet adds this logic module, called offers, as a new capability for its brain component. The following piece of Java code shows the seller posting offers.

```
//--- SimpleSeller.java, a Java client

public class SimpleSeller {
...
public void postOffers(){
URLConnection connect;
   ...
//--- Posting offers. Adding a capability ----------//
connect = (new URL(
   "http://localhost:8080" +
   "/servlet/DefaultMARlet?service=addCapability")).
         openConnection();
   ...
connect.connect();
outstream = connect.getOutputStream();

outstream.println("offers"); //--- idModule ---//
outstream.println("
  article(tv,[hitachi, 20in,800],
            sendEmailTo(sales@goodies.com)).
  article(radio,[panasonic, h7823,80],
            sendEmailTo(sales@goodies.com)).
  article(tv,[sony, 21in, 1200],
            sendEmailTo(sales@goodies.com)).
"); //--- knowledge ---//
   ...
}...} //--- End of SimpleSeller class
```

In the other hand, the customer sends a request to the MARlet to activate the new capability of the brain component. It aims to enable the brain component to accept queries involving article offers. Then, the customer sends a goal request to the MARlet. The MARlet delegates this goal request to the brain component. In this way, the brain asserts the customer's preferences about cards and TV sets in its knowledge base, and infers an appropriated article according to these preferences. Finally, the customer reads the response from the input stream. The following Java code shows the customer client implementation.

```
//--- SimpleCustomer.java, other Java client

public class SimpleCustomer{
...
public boolean buyArticle(Article anArticle){
   ...
//--- Activating a MARlet's capability ----------//
connect = (new URL("http://localhost:8080" +
      "/servlet/SimpleMARlet?service=activeCapability")).
         openConnection();
   ...
connect.connect();
outstream = connect.getOutputStream();
outstream.println("offers"); //--- idModule ---//
//-- Choosing an offer according to my preferences ----//
connect = (new URL("http://localhost:8080" +
         "/servlet/DefaultMARlet?service=answerQuery")).
            openConnection();
   ...
connect.connect();
outstream = connect.getOutputStream();
instream = connect.getInputStream();
type = anArticle.type;
outstream.println("
   ?- assert(preference(car,[ford, Model, Price]) :-
                  Model > 1998, Price < 60000),
   assert(preference(tv,[sony, Model, Price]) :-
                  Model = 21in, Price < 1500),
   article(" + type + ",[Brand, Model, Price],
                        sendEmailTo(Email)),
   preference(type,[Brand, Model, Price]).
"); //--- Goal Request ---//

success = instream.readBoolean();
response = instream.readHashTable();

if success {
// Buying the article. Sending an e-mail to the provider//
   ...
   emailAddress = response.get("Email").toString();
   this.mailTo(emailAddress, subject, txt);
   ...
}...
}...}//--- End of SimpleCustomer class
```

4.2. Brainlets: Moving Around Sites

The previous example presented a remote evaluation model via Brainlets. That is, the client sends the code that is executed in the server to get a response. If we have many sites offering products with MARlet support installed, we can then use a Brainlet to move around these sites and select the best available offer according the customer preferences. The following code shows a simple Brainlet that implements this mechanism. The example assumes that each site provides a uniform interface.

```
logicModule (customerBrainlet)  :-  {

Sites=[www.buyers.com,www.offers.com,...].

preference(car,[ford, Model, Price]) :-
                  Model > 1998, Price < 60000).
preference(tv,[sony, Model, Price]) :-
                  Model = 21in, Price < 1500).
lookForOffers(A,[ ],_,[ ]).
lookForOffers(A,[S| R], [O|RO], [O| Roff]):-
                        move_to(S),
                        article( A, Offer, Email),
                        O= (S,Offer,Email),
                        lookForOffers(A, R, RO,ROff).
lookForOffers(A,[S| R], [O|RO], [O| Roff]):-
                        lookForOffers(A, R, RO,ROff).

buy(Art):- lookForOffers(Art, Sites,R,Offers),
           selectBest(Offers, (S,O,E)),
           move_to(S),
           buy_article(O,E).
?- buy(#Art).
}
```

In this example the Brainlet has a goal of buying an article received as a parameter. The *buy* clause looks for offers available in the different sites, selects the best and calls a generic predicate to buy the article (this process is not relevant here). The *lookForOffers* predicate implements the process of moving around the defined sites looking for the available offers for the article (we assume that we get the first offer). If there is no offer in the current site, the Brainlet goes to the next one in the list. The *SimpleCustomer* class defined previously must publish this Brainlet.

5. Conclusions and Future Work

In this article, a multi-paradigm approach for the development of intelligent agents has been introduced. This approach integrates both logic and object-oriented programming paradigms to support the development of stationary and mobile agents capable to manage complex mental attitudes. The approach is materialized by a software architecture based on the JavaLog programming language.

Major advantages of Brainlets are seen developing agents that require an intelligent and flexible behavior. Reducing expensive global communication costs by moving the computation to the data sources and distributing complex computations onto several hosts, which could be heterogeneous, is other worth mentioning benefit.

The MARlets support provides a simple way to extend any web server supporting servlets with mobile inferential capabilities. Although, the efficiency of the inferential capabilities can be low, considering that JavaLog is implemented in Java. This limitation, however, is not relevant when complex inferential behaviors are needed to provide a service.

The proposed approach represents one of the current lines into an ongoing research and development project, at ISISTAN Research Institute. This project aims to develop intelligent agents in the socialware domain.

There are several open topics that require further research. Backtracking among different hosts is one of the most interesting; it is due to the different semantics that can be adopted.

6. References

[Amandi et al. 99] A. Amandi, A. Zunino and R. Iturregui. Multi-paradigm Languages Supporting Multi-agent Development. In Multi-Agent System Engineering. F. J. Garijo and M. Boman (Eds.). Lecture Notes in Computer Science, Vol. 1647, pp. 128–139. Springer-Verlag, Berlin - Heidelberg - New York, 1999.

[General Magic Corp. 98] Odissey White Paper. General Magic Corp. Cupertino, Calif., 1998.

[Hattori et al. 99] F. Hattori, T. Ohguro, M. Yokoo, S. Matsubara and S. Yoshida. Socialware: Multiagent Systems for Supporting Network Communities. Communications of the ACM. Vol. 42, No. 3, pp. 55-61. March 1999.

[Karnik and Tripathi 98] N. Karnik and A. R. Tripathi. Design Issues in Mobile Agent Programming Systems. IEEE Concurrency, Vol. 6, No. 3, July-September 1998.

[Lange and Oshima 98] D. Lange and M. Oshima. Programming and Deploying Java Mobile Agents with Aglets. Addisson-Wesley Longman, Reading Mass., 1998.

[ObjectSpace Corp. 98] Voyager White Paper. ObjectSpace Corp. Dallas, Texas, 1998.

[Odell et al. 00] James Odell, H. Van Dyke Parunak and Bernhard Bauer. Extending UML for Agents. AOIS Workshop at AAAI 2000, Austin TX, 2000

[Shoham 93] Y. Shoham. Agent-Oriented Programming. Artificial Intelligence, 60(1), pp. 51-92, March 1993.

[Shoham 97] Y. Shoham. An Overview of Agent-Oriented Programming. In Software Agents, J. M. Bradshaw (Ed.), pp. 271-290. MIT Press, 1997.

[Wong et al. 99] D. Wong, N. Paciorek and D. Moore. Java-based Mobile Agents. Communications of the ACM. Vol. 42 - No. 3, pp. 92–102, March 1999.

[Zunino et al. 01] A. Zunino, L. Berdún and A. Amandi. JavaLog: un Lenguaje para la Programación de Agentes. Revista Iberoamericana de Inteligencia Artificial. En este número. No.13 (2001). ISSN: 1137-3601. AEPIA (http://www.aepia.dsic.upv.es/).