Analele Ştiinţifice ale Universităţii "Alexandru Ioan Cuza" din Iaşi
Informatică, Tomul XII, 2002, 45–53

Scientific Annals of the "Alexandru Ioan Cuza" University of Iaşi
Computer Science Section, Tome XII, 2002, 45–53

An XML-based Object-Oriented Infrastructure for Developing Software Agents

Sînică ALBOAIE[1] Sabin-Corneliu BURAGA[2] Lenuţa ALBOAIE[3]

Abstract

In this paper, we present an agent-based object-oriented solution for accessing the Web distributed resources. We describe a multi-agent infrastructure – called *Omega* – that can be considered as a hierarchical space of a set of distributed objects that models the Web resources. We propose an XML/RDF-based model that can be used as an universal manner for serialization and metadata description of the objects processed by the agents. We also present a distributed address space for active objects and the way in which we can use them as a foundation for a Web-like distributed system that uses active objects.

1 Introduction

The primary goal of Tim Berners-Lee's vision of the Semantic Web [4, 18] is to enable intelligent queries for knowledge on the Web instead of the conventional mechanisms to access the resources. To do this, computer scientists need to achieve the following:

[1]Institute of Theoretical Computer Science, Romanian Academy, Iaşi branch
abss@iit.iit.tuiasi.ro
[2]Faculty of Computer Science, "Al. I. Cuza" University of Iaşi, Romania
busaco@infoiasi.ro
[3]Faculty of Computer Science, "Al. I. Cuza" University of Iaşi, Romania
adria@infoiasi.ro

- to understand the semantic mechanism of all kinds of queries, and what kind of components the process of questioning the Web formally consists of;
- to rigorously capture, represent or symbolise the knowledge contained on the Web.

To accomplish this goal, we are designing and implementing an infrastructure for agent software development, called *Omega* [2], viewed as a tree-like space of a set of distributed objects that models the Web resources by using XML/RDF constructs. The *Omega* system offers a flexible framework for building agent-oriented distributed applications on the Web – consult section 3 for details.

The *Resource Description Framework* (RDF) [9, 24] may develop into one of the foundations of the Semantic Web by enabling semantic interoperability. RDF is designed for data sharing and processing in an automatically way as well as by humans (see section 2). To assure the Web scalability, independently designed programs – especially Web agents – must be able to exchange and process the meaning of data and metadata in an independent manner. Semantic interoperability can be completed only if different users (agents, tools, other different Web clients, etc.) interpret RDF statements in the same way.

The *Omega* framework offers an addressing space for the Web objects and a mechanism for remotely accessing the Web distributed resources (that can be viewed as objects). To enable the flexible quering and accessing mechanisms about the distributed Web resources, we must offer a facility for serialization – in an independent manner – the data and metadata (objects) processed by the *Omega* agent system. In our first implementation, the object serialization was given as a proprietary data format. In this paper, we investigate some of the possibilities of serialization given by the XML family of markup languages [34]. Some of the drawbacks due of the lack of a description language regarding the objects' properties can be elegantly resolved by XML. Even our approach can be used in the context of Web services discovery and description infrastructures, the paper do not intend to discuss these issues. From the authors' point of view, the serialization of the Web objects (see section 3.4) can be considered as a flexible way to exchange information between software agents. Additionally, for each object we can attach different metadata constructs to specify several semantic properties.

These descriptions are written in RDF and shortly presented in section 3.5.

2 Resource Description Framework (RDF)

2.1 General Presentation

Resource Description Framework (RDF) allows the description of the metadata associated of the Web documents (resources). RDF consists of a model for the representation of the named properties and property values. This is suitable to model objects behaviours. RDF properties may be thought of as attributes of resources and in this sense correspond to traditional attribute-value pairs. RDF properties can also signify relationships between resources and therefore an RDF model can be similar to an entity-relationship diagram. In object-oriented design terminology, resources correspond to objects and properties correspond to instance variables [9, 11, 24].

To facilitate the definition of metadata, RDF is based on *classes*. A collection of classes, typically designed for a specific purpose or domain, is called a *schema* [9]. Through the sharability of schemas, RDF supports the reusability of metadata definitions. The RDF schemas may themselves be expressed in RDF. To syntactically represent the metadata, the RDF model uses the XML (Extensible Markup Language) [8] language.

Figure 1 shows the level where RDF is situated in the context of Web data modeling layers [3].

2.2 RDF Model

The basic model of RDF consists of three object types:

resources All objects being described by RDF expressions are called *resources* and they are always named by *Uniform Resource Identifiers* (*URI*) [5] plus optional anchor identifiers. Using URI schemas (i.e. `http`, `ftp`, or `file` schemas), every type of resource can be identified in the same manner.

properties A *property* is a specific aspect, characteristic, attribute, or relation to describe a resource, as stated in [24]. Each property has a specific meaning, defines its permitted values, the type of resources it can specify, and its relationship with other properties (via RDF Schema).

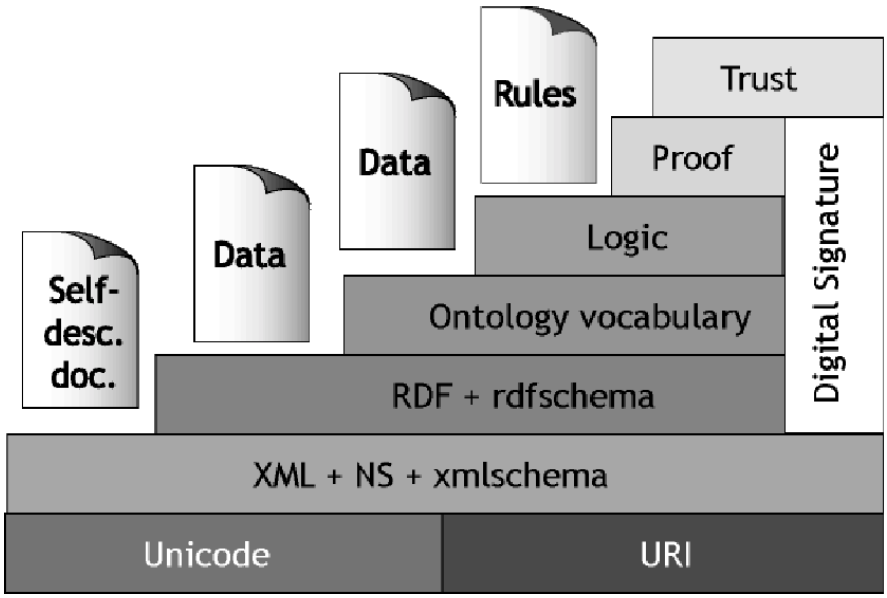

Figure 1: Web data modeling layers

statements A specific resource together with a named property, plus the value of that property for that resource is an RDF *statement*. These three individual parts of a statement are called, respectively, the *subject*, the *predicate*, and the *object*. The object of a statement (e.g., the property value) can be another resource or a literal.

RDF also specifies three types of container objects:

- *Bag* – an unordered list of resources or literals,
- *Sequence* – an ordered list of resources or literals,
- *Alternative* – a list of resources or literals that represent alternatives for the single value of a property.

The collections can be used instead of *Description* elements. The containers may be defined by a URI pattern. RDF can also be used to make statements about other RDF statements (higher-order statements).

The RDF data model provides an abstract, conceptual framework for defining and using metadata. Currently, there are several proposals of model-theoretical semantics for RDF and RDF Schema [17, 23].

2.3 RDF Formal Model

The RDF data model can have three equivalent representations: as 3-uples, as a graph, and in XML. Formally, the RDF data model is defined as follows:

1. There are three sets called *Resources*, *Literals*, and *Statements*.
   There is a subset $Properties \subset Resources$.
   Each element $s \in Statements$ is a triple $s = \{ pred, sub, obj \}$, where $pred \in Properties$, $sub \in Resources$, and $obj \in Literals \cup Resources$.

2. The *reification* of a $\{ pred, sub, obj \} \in Statements$ is a new resource $r \in Resources$ as follows:

$$s_1 : \{\text{type}, [\, r \,], [\, RDF : Statement \,]\}$$
$$s_2 : \{\text{predicate}, [\, r \,], [\, pred \,]\}$$
$$s_3 : \{\text{subject}, [\, r \,], [\, sub \,]\}$$
$$s_4 : \{\text{object}, [\, r \,], [\, obj \,]\}$$

   where s_1, s_2, s_3, $s_4 \in Statements$ and type, predicate, subject, object $\in$ *Properties*.

3. There is an element of *Properties* known as $RDF : type$. Members of *Statements* of the form $\{ RDF : type, sub, obj \}$ must satisfy the following condition: *sub* and *obj* are members of *Resources*.

4. There are three elements of *Resources*, not contained in *Properties*, known as $RDF : Seq$, $RDF : Bag$ and $RDF : Alt$. There is a subset $Ord \subset Properties$ corresponding to the ordinals $(1, 2, 3, \ldots)$. We refer to elements of *Ord* as RDF_i, where $i = 1, 2, 3, \ldots$

More details can be found in [9, 11, 17, 23].

3 *Omega* Agent Framework

3.1 Motivation

We can consider as the fundamental resources that computers expose to the software components (i.e. operating system, applications) or users the following items: the computing capabilities, (volatile or non-volatile) memory, local and remote data (documents), metadata (different descriptions about several properties of the resources: content, structure, layout/interface, dynamics, security issues, etc.).

Of course, there are other modalities to describe these properties without using XML-based assertions, but with the penalty of the platform and software independence. Obviously, these documents (including XML resources) are made to be read and processed in a distributed system (the Web itself). To easily access and obtain the knowledge contained by a specific document, there must exists an universal mechanism/model – based on the XML family – to accomplish that. This is the seminal idea of the Semantic Web [4].

3.1.1 WWW Space as a Distributed Hypermedia System

Also, the World-Wide Web space can be viewed as a distributed hypermedia system that uses Internet technologies (i.e. TCP/IP protocol family), a global system of heterogeneous networked computers. Advances in networking and Web/Internet technology are leading to a network-centric computing model, and the Web and Internet itself, of course, are evolving into the infrastructure for global network computing. By populating this infrastructure with object-based components and combining them in various ways, we can enable the development and deployment of interoperable distributed object systems on the Web [33].

The object model provides the ability to mimic real world processes in a fluid, dynamic and natural manner. The Web allows for objects to be distributed to servers thereby centralizing access, processing, and maintenance, provides a multiplexing interface to distributed objects, and allows thin-clients (e.g., mobile phones or handheld devices). We can safely now state that **Web + Object** integration is a viable reality. This is emphased by different software organizations and companies – especially in the e-business domain [12, 29] – that are using Web-enabled distributed object

technology, in the form of intranets and extranets, to solve their computing problems, and the emergence of an industry that provides Web and object interfaces to distributed object tools.

3.1.2 From the CGI Approach to a Distributed Object Infrastructure

But the Web didn't start out this way. Network-centric object computing is the result of a logical technological evolution. As originally conceived, it was driven by hypertext documents called Web pages or HTML documents [4, 11]. Initially, Web pages had static content (rich graphics and text at first, multimedia later), and were interlinked. Browser applications running on user PCs or workstations were used to retrieve documents stored on Web servers. Helper applications supplemented the browser, handling other document types such as Word, PostScript, PDF (Portable Document Format) or different graphics, video, and audio formats. Web pages soon begun to include dynamic content as helper applications, called *plug-ins*, were integrated into the browser and CGI (Common Gateway Interface) [15] scripts enabled users to input data to a Web server and access Internet services (i.e. data queries). Finally, programmatic content was added via Java applets, VBScript and JavaScript programs, to provide further interactive functionality and modify content in-place [11]. These languages and techniques enable richer documents (e.g., animation and Web forms generated on-the-fly). Note that programmatic content can also include server-side execution of code such as accessing a remote database service (i.e. SQL queries).

Prior to the addition of programmatic content, the Web was based on a client/server computing model which lacked scalability, common services, security, and a development environment needed to develop and deploy large-scale distributed applications. CGI scripts are not scalable because each requires a separate server-side process to handle each client request, services are limited to accessing database servers via CGI scripts, transaction information (such as credit card information) is not encrypted, and the programming model offered by HTML/HTTP using CGI and a three-tiered system is limiting.

For example, the CGI model presents some disadvantages:

- CGI technology is not object-oriented and offers no type safety,
- dynamic interaction with the user is difficult,

- the interface between the server and database manager is *ad-hoc*,
- the interfaces cannot be combined or extended,
- client data consists of simple strings that the server must explicitly decode,
- there are no standard definitions of service interfaces, and the interfaces are not self-describing.

Also, CGI applications are more error-prone, are harder to maintain, and are not likely to be reusable.

With the advent of Java, and the distributed object infrastructures CORBA/IIOP and OLE/DCOM, the stage was set to evolve the Web from a document management system to a platform for distributed object computing and electronic commerce.

Bringing distributed objects to the Web offers the following advantages (to name only few of them):

- extensibility (e.g., for applications, services, and APIs built from objects, objects can easily be replaced or added),
- cross-platform interoperability,
- independent software development,
- reusable software components,
- componentware,
- network services,
- better utilization of system resources.

Existing legacy applications can even co-exist with distributed objects through the use of object wrappers [33]. The interface could either be the client browser or browser-like with superpositioned distributed object infrastructures.

3.1.3 Mobile Agents

An important step towards *Internet/Web Computing* is represented by the mobile computations. A mobile object, usually called an *agent* when operating on behalf of a user, is a downloadable, executable object that can independently move (code and state) at its will – the mobile agent is not bound to the system in which it began the code execution and can travel from one node (host) on a network to another.

Mobile agents present the following attributes:

1. *reactive* – the ability to respond to changes within agent environment;
2. *autonomous* – the mobile agent is able to exercise control over its own actions (decisions);
3. *goal-oriented* – the agents have a planned itinerary, they do not simply act in response to the environment;
4. *communicative* – the ability to communicate with other agents, by exchanging information (knowledge);
5. *mobile* – the mobile agents can transport themselves from one machine to another.

Mobile agents provide a way to think about solving software problems in a networked environment that fits more naturally with the real world. Mobile agents can be used to access and manage information that is distributed over large areas [7, 21].

The main benefit is that the software components can be integrated into a coherent and consistent software system – e.g. a multi-agent system – in which they work together to better meet the needs of the entire application (utilising autonomy, responsiveness, pro-activeness and social ability).

A mobile agent architecture provides the "framework within which mobile agents can move across distributed environments, integrate with local resources and other mobile agents, and communicate the results of their activities back to the user. This framework can then be used to build mobile agents that perform user-driven tasks to fulfill distributed information management goals" [21].

Taking this notion further, mobile agents could be used to monitor the network activities and provide input to QoS (Quality of Service) and global

optimization mechanisms. They could be used during negotiation (with representative agents) to solve different constraint optimization problems.

One key research area is to provide security against malicious agents (who intend to access local resources or can carry a virus) and malicious hosts (who can alter the agent code/state or read private information) [30].

The current mobile agent systems – available as commercial or open-source applications – are implemented in C++, Java, Tcl, Scheme and Python programming languages.

3.2 Internal Architecture of the *Omega* System

3.2.1 Overview

The *Omega* is an agent-based system that offers an addressing space (viewed as a tree) for the Web objects and different techniques to remotely access the Web distributed resources (viewed as objects) [2].

Each object processed by *Omega* can be considered as a collection of objects included in that one. The links (edges) between the vertices of the tree are given by the aggregation relationship exposed by the object-oriented methodologies [6].

To emphase the aggregation relationship, we attach to each object a name or an index, and in this way we can uniquely refer each object of the tree by its name/index (viewed as an identifier). Each object will have an unique list of the identifiers that represent its "address" in the addressing space used by the *Omega* agents. An identifier can be considered as an `IName` object (at the implementation level, an `IName` object can be viewed as an object-tree path or a list of object identifiers). By using a tree of objects, we can structure more easily the distributed resources for a given local web (such as a cluster or an intranet).

3.2.2 Functionality

We choose to use an interpreted environment for our multi-agent model and distributed object structure. Using such an environment, it was easier to implement serialization and various execution control mechanisms [16] which contributed to the implementation of the *Omega* distributed objects system.

Omega offers a distributed object structure, and its initial goal was to determine some good representations of data, types, instructions, functions

and objects of an object-oriented language that can be used as a programming language for mobile agents. The result of this effort is a system written in C++ that is able to unify the notions behind the object-actor duality, namely the duality between passive and active objects.

In this stage, we consider *Omega* as an attempt to offer to active programs what the World-Wide Web space provides by default for presentation of some static entities (documents), namely an infrastructure able to support Web-based distributed applications (e.g., software agents used in clusters or Grid – see [28] for details).

As an example, let us consider the problem of a system in which someone from a location A wish to obtain in real time data from another location B. There is more than one solution, and we present here just two possibilities:

1. In a multi-agent paradigm, we create two agents in A: an agent for the information point, and an agent to be sent at the location B and to obtain the information needed to be communicated to the location A. This approach is used in the design of *Omega* [2].

2. Another way of solving this problem is to create a Web site at B (providing different server-side solutions – i.e. CGI scripts or Java servlets) or a client/server application using a proprietary (TCP/IP-based) protocol [14].

We can observe that the first solution (the multi-agent approach) is more scalable and closer to certain good rules for programming design.

By using the multi-agent paradigm, a system can be easily divided into small entities with control over their interactions. Moreover, we can get a more flexible and adaptable approach (in our example, we can have a more adaptable way of presenting the information at the location A). This flexibility is a part of the client task, as opposed to the Web approach where we require more tasks for the server. In the case of using a client/server solution (with a proprietary communication protocol), some problems come from the high cost of the system design and maintenance.

The solutions that use C++ (networking, DCOM, CORBA) or even Java/C# are quite complicated and they are, in many cases, inappropriate for an open system, as the Internet – and Web, also – is. At the moment, for the generic problem of our example, a client/server solution is more popular

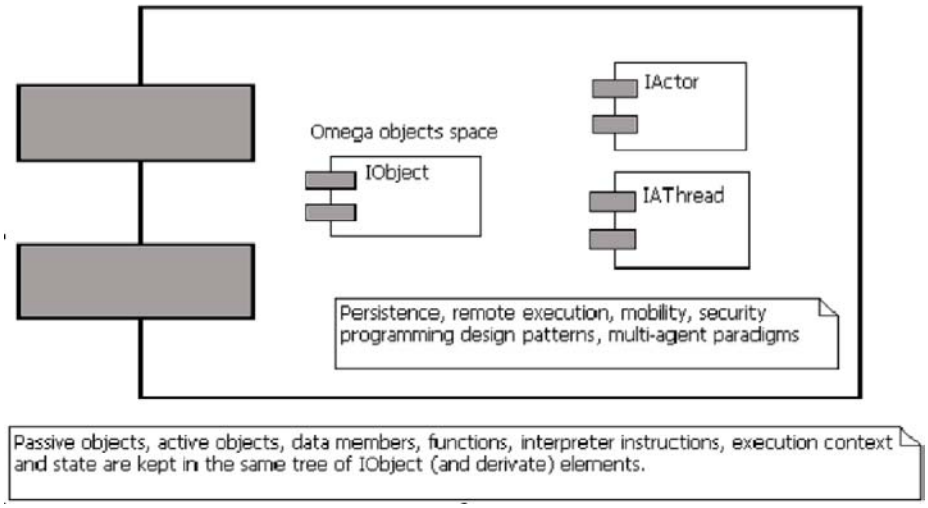

Figure 2: *Omega* objects

in industry (in many cases based on HTML or XML). The later approach is adopted by Web services [28, 34] scenarios, also.

From object-oriented paradigm's perspective, *Omega* can be seen as an object hierarchy that ensures an unitary way of programming, with an implementation of a name-service that is consistent for the resources (objects) that it makes available. The *Omega* system offers serialization mechanisms and garbage collection, also.

For example, the `IObject` class is the base-class for every other class that has memory regions stored within a local system. Every object and function that needs a store space in *Omega* will use `IObject`. In this way, *Omega* assures a space model provided by a common distributed memory. This model is based on the existence of a given node of an `IObject`'s tree, which is easily addressable from the network.

3.2.3 *Omega* Classes

The *Omega* system offers a number of object types which provide functionality to the following classes:

- String class;

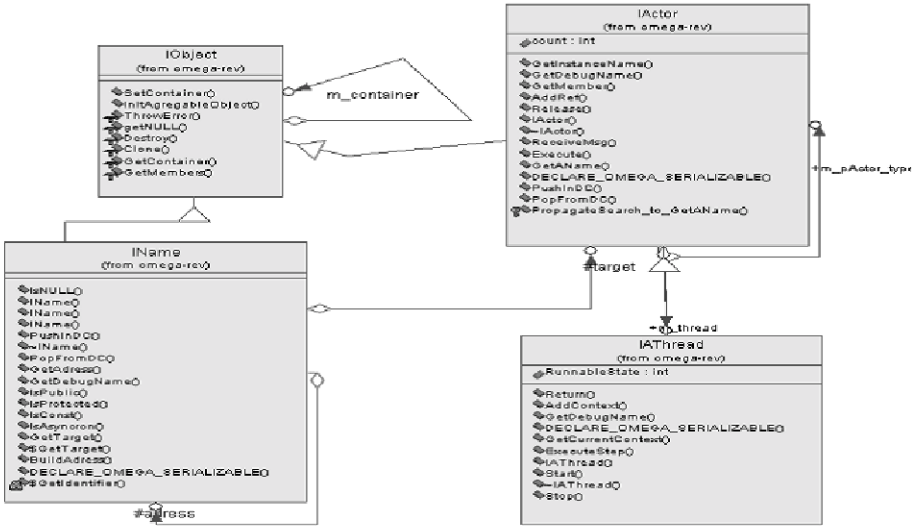

Figure 3: The architecture of the *Omega* classes (fragm.)

- Number class;
- List class;
- Control agent-execution class (i.e. support for virtual threads, scripting languages etc.).

As an optimal manner to serialize the *Omega* objects, we choose an XML-based representation. To describe these objects, we will adopt an RDF-based model. The RDF assertions will give the possibility to express semantics of an *Omega* object.

In the *Omega* framework, the data are represented by different classes such as: `IString`, `INumber`, `IOmegaStack`, `IOmegaList`, `IOmegaQueue` that are derived from the `IObject` generic class presented in section 3.2.2.

The *Omega* offers two categories of data types [2]:

- Simple data types – have no components (i.e. `INumber`, `IString`, etc.);
- Compound data types – represent a mix-up of two or more simple types (e.g., `IName`, `IOmegaStack`, `IOmegaList`, `IAThread`).

A compound data type can be considered as an "array" or a "struct" (very similar with the `struct` used in the standard C language).

In our approach, the string data type (`IString`) is not similar to the common concept of the "string" type (present in all modern programming languages). At the implementation level, *Omega* system will use for `IString` another manner to store the content of a string (we do not use XML Schema's `xsl:string`).

3.3 Object Dynamics in *Omega*

3.3.1 *Omega* Name-Service

The *Omega* framework implements a name-service. The main characteristics and rules of this name-service are the following:

- First of all, the space of distributed objects is a tree forest and every node of the *Omega* virtual machine has a tree. The tree nodes are objects derived from the `IObject` generic object (see section 3.4) and have a platform-independent representation. These objects are persistent. Each tree has a root-node: an object that represents the virtual machine. The tree root contains the address (`hostname:port`) of the TCP server that will be used for message transmission through the network and is the root of all names. Every tree node is a possible container for other objects and has a connection with the parent. Each node of the object-tree has an `ObjectID` that identies the child number provided by the parent. An address in the object space is a list that contains the identification of the root as the first list item, and then, in the other items, `ObjectID`s of every object of the tree that is on the path from the root to the node. The address is denoted by the following construct:

$$[hostname : port, id_1; id_2; ...; id_n]$$

- Every object can have one or more associated references (these references are objects themselves); a reference optionally contains the address list (as above), a pointer to the current addressed object, or just its name (or all of these three elements).

- For reasons of persistence, names could not be unique. We use the list of the object-identifiers, but – at execution time – we use the pointer

to the current object (if it is on the same machine), and this is easy to get from the identifiers list.

We use an alternative to the active objects [25] and we intensively use asynchronous communication.

3.3.2 *Omega* Language

For the object system presented above, we provide an active part, namely an execution part, which is the implementation of a small programming language that is using *Omega* objects. In this active part, we try to integrate the object space with notions such as execution thread, function, instruction, data types to be modeled with the help of `IObject` abstraction. In the current implementation, the execution threads represented by an `IAThread` object (actor thread) will have a current execution context in which it can keep the local names and a global name list of the task (a task has more execution threads, some objects have attached execution threads, and they have the same name list from the task they belong to).

To simplify the development of a high-level control language, we are started from a data-type model that had `IString`, `INumber`, `IThread`, and `IObject` as base types and various types derived from `IOmegaActor` (this class is derived from `IActor`). The system is able to initialise and execute `IOmegaActor` objects.

Therefore the *Omega* object environment and the *OmegaKernel* mini-interpreter provide:

- A data model (base type-system, the construction of new objects);
- An address space (every object has its own address consistent at the Internet level);
- Techniques to implement the high-level programming level statements (e.g., `if`, `while`, `for`, or `goto`).

Omega is able to execute small ("scripting") programs. We present below such a program called *test program* – new `IObjects` are created (such as `IContextExecution`, `IName`, `INumber`, `IAThread`, or `IOmegaFunction`).

The main idea is that at runtime, everything is reduced to a creation of new `IObjects` in the distributed space of objects.

```
OmegaTrace ("Test begin")
BeginActor (SimulateDoWhile)
NewINumber i 0
label begin
Inc i
OmegaTrace ("i++ in SimulateDoWhile")
LessThenGoTo i 2 begin
EndActor
SimulateDoWhile ()
OmegaTrace ("Test end")
```

The language provided by the *Omega* framework is similar to an assembler language and may be easily extended with other instructions. The main syntactic construct is similar to a function call. An important step was to create a mechanism for representing data structures, statements and objects under the same abstraction (`IObject`) that is a network shared entity.

3.4 Serialization Mechanism

All classes derived from `IObject` must implement the *serialization* (*marshalling*) and *deserialization* (*unmarshalling*) methods. The process of building of the new data types is based on the fact that an `IObject` has a member of the `IOmegaList` type. That member containts associated links which are instances of the derived classes. In this manner, the serialization of the new types of objects can be automatically accomplished by *Omega* via members' serialization and the call of the overloaded own methods. Of course, for several types of objects (e.g. `IOmegaSockets` used for socket operations) the serialization and deserialization activities can not be viewed as a proper solution.

For each access to a sharable object, a *proxy-object* is created, using the RPC (Remote Procedure Call) mechanism [14]. This proxy-object is placed in the same tree of the target object. In the tree of the accessed object, a *stub-object* is created, too. The stub will contain meta-descriptions about the sharable object and will be derived from `IObject`. The stub-object will be a member of the sharable object. This approach allows us to remotely access the stub. In this way, the system will be able to keep updated versions of the different object trees. To obtain the serialized form of an object, the RPC-like mechanism is able to transmit the URI (Uniform Resource Identifier) [5]

of the object. As a response, the system will get the serialized forms of the object and of the proxy-object as well, if it is possible. The *Omega* system is responsible to regularly update the proxy-objects.

The object serialization do not imply the serialization of the whole subtree that has as root the object in cause. For an object, only the serialization of the object itself and of the `IName` list of its children.

3.4.1 XML-based Serialization

The process of serialization uses XML-based constructs. For this, we use the `xsi` namespace available at the following address:

```
http://www.w3.org/1999/XMLSchema-instance
```

This XML namespace is defined by the XML Schema specification [19]. Also, we use the `xsd` namespace, available at this Web address:

```
http://www.w3.org/1999/XMLSchema
```

These namespaces are used to retain the primary types of the data exchanged by agents in the serialization and deserialization processes.

An example (we are using an `IString` object):

```
<IString>
  <name xsi:type="xsd:string">
     Hello OMEGA
  </name>
</IString>
```

The *Omega* encoding style is based on the usual data types defined by the XML Schema data types specification [19]. All data types used within the *Omega* system of agents must either be taken directly from the XML Schema or derived from Omega data types (see section 3.2.3).

The XML Schema specification (see *Datatypes* section from [19]) does not offer the possibility to express data types as XML elements, but only as attributes. To address this, the *Omega* framework declares an schema, called `OMEGA-ENC`, used to define an XML element for each data type (see the example below).

```
<OMEGA-ENC:int id="int1">
  33
</OMEGA-ENC:int>
```

3.4.2 Example

A complete example of *Omega* object serialization follows:

```
<element name="local_address_type" type="...">
  <!-- an element of a simple type -->
  <simpleType name="local_address_type"
              base="xsd:string">
     <enumeration value="tree_id" />
     <enumeration value="unique_name" />
  </simpleType>
</element>
<element name="local_address" type="..." />
  <!-- an element of a complex type -->
  <complexType name="local_address">
    <element name="la_type" type="local_address_type" />
    <element name="la_value" type="xsd:string" />
  </complexType>
</element>

<IName>
  <IOmegaDomain> ... </IOmegaDomain>
  <local_address>
     <la_type> tree_id </la_type>
     <la_value> 1 </la_value>
  </local_address>
  <local_address>
     <la_type> unique_name </la_type>
     <la_value> member_name </la_value>
  </local_address>
  <!-- more other similar constructs... -->
</IName>
```

These XML elements could be used by the programmer to extend the functionality of the *Omega* system with new data types. The *Omega* system only proposes the presented XML-based manner of object serialization, but does not interdict other mechanisms to be adopted for serialization.

3.4.3 SOAP Object Serialization

SOAP – or other protocols that use the RPC over XML approach (e.g., XML-RPC) – will be used to transport the serialized data. SOAP looks to be the right solution because of the great support it gets from different companies and organisations (i.e. Google, Microsoft, or Sun).

SOAP (*Simple Object Access Protocol*) [20, 34] is a simple lightweight protocol used for structured and strong-type information exchange in a decentralized, distributed environment. The protocol is based on XML and consists of three parts:

1. An envelope that describes the contents of the message and how to use it;
2. A set of rules for serializing data exchanged between applications;
3. A procedure to represent remote procedure calls, that is, the way in which queries and the resulting responses to the procedure are represented.

Similar to object distribution models (e.g., IIOP and DCOM), SOAP can call methods, services, components, and objects on remote servers. However, unlike these protocols, which use binary formats for the calls, SOAP uses text format (Unicode), with the help of XML, to structure the nature of the exchanges.

SOAP can generally operate with several protocols, such as FTP (File Transfer Protocol) or SMTP (Simple Mail Transfer Protocol), but it is particularly well-suited for the HTTP (HyperText Transfer Protocol) [15, 34]. It defines a reduced set of parameters that are specified in the HTTP header, making it easier topass through proxies and firewalls. Using SOAP over HTTP also enables resources already present on the Web to be unified by using the natural request/response model of HTTP protocol. The only constraint is that a SOAP message via HTTP must use the MIME (Multipurpose Internet Mail Extensions) [11] type `text/xml`.

We use an existing tool named *gSoap* [32], which is able to generate the code for serialization from a user-defined specification. The *gSOAP* compiler tools provide an unique SOAP/XML-to-C/C++ language binding to ease the development of SOAP/XML Web services and clients in C and/or C++ languages.

Most toolkits for C++ Web services adopt a SOAP-centric view and offer APIs for C++ that require the use of class libraries for SOAP-specific data structures. This often forces a user to adapt the application logic to these libraries. In contrast, *gSOAP* provides a C/C++ transparent SOAP API through the use of compiler technology that hides irrelevant SOAP-specific details from the programmer. The *gSOAP* stub and skeleton compiler automatically maps native and user-defined C and C++ data types to semantically equivalent SOAP data types and vice-versa.

As a result, full SOAP interoperability is achieved with a simple API relieving the user from the burden of SOAP details and enables him or her to concentrate on the application-essential logic. The compiler enables the integration of (legacy) C/C++ and Fortran codes (through a Fortran to C interface), embedded systems, and real-time software in SOAP applications that share computational resources and information with other SOAP applications, possibly across different platforms, language environments, and disparate organizations located behind firewalls.

3.5 Using RDF Metadata Constructs to Capture the State of the *Omega* Objects

For each object of the *Omega* system, we can attach different metadata collections. These meta-descriptions will assure the control versioning (the version and the last verification time-stamp for each object of the tree) and the author of the created objects. For security purposes, the object metadata will retain the list of the associated object's permissions. This approach is inspired from our RDF-based model used for accessing resources of the distributed file systems presented in [10].

The system keeps these description as RDF assertions that can be transported to other objects during the information updating activities (object replication).

In the stub object of any shared objects, certain metadata is available to inform other objects about the object's author and about the permission list to access this object. The system verifies the information – found on the path to this object in the objects tree, by checking the URI from `IName` data-member – to grant or to deny the access to considered object. The meta-descriptions regarding the permissions and the identity of the user who want to access *Omega* objects must provide a certain cryptographic support.

The temporal relationships between objects are stored by RDF constructs, too. This approach is very similar to the model used to retain temporal relations established between (fragments of) Web sites [13].

4 Related Work

Althrough there is not a formal framework for multi-agent systems development, due to dependence on application domains, it has been that the construction of these systems requires a different approach from that of conventional software systems development [26].

We are aware of multiple platforms developed both in academia and software industry companies [27]. This confirms that many computer scientists are considering the agent-oriented software as a possible paradigm, designed and implemented especially in very dynamic environments (such as World-Wide Web space). We can give different examples of frameworks and tools used to develop multi-agent systems (for more details, see [27]), some of them using the Internet open standards:

- *Tryllian's ADK (Agent Development Kit)* – an agent-based business integration platform, designed and built in Java, XML and JXTA with a modular architecture and a unique mobile component approach;
- *Toshiba's Bee-gent (Bonding and Encapsulation Enhancement Agents)* – a CORBA-based communication framework intended to provide cooperative processing in the advanced network society;
- *FIPA-OS* – a Java component-based layered toolkit enabling rapid development of FIPA (Foundation for Intelligent Physical Agents) compliant agents;
- *Grasshoper* – an open-source CORBA-based platform that allows software agents to move between different fixed and wireless computing systems and to execute various tasks in the process; this platform provides support for MASIF (Mobile Agent System Interoperability Facility) – a standard specification developed by the Object Management Group (OMG);

- *JADE (Java Agent DEvelopment Framework)* – a widely used agent platform that can be distributed across heterogeneous machines and that can be configured via a remote GUI (Graphical User Interface);

- *Xraptor* – a simulation environment for continuos virtual multi-agent systems written in C++ for Unix platforms that allows studying the behavior of agents in different 2- or 3-dimensional worlds.

However, the existing implementations have not convinced the whole community or do not cover or provide some facilities desired by programmers or final users. Some proprietary solutions, though well developed, are not built as open systems and can not be easily extended or modified. On the other hand, we were not impressed by the available open-source platforms. Therefore, from the authors' point of view, it was more useful and interesting to design and implement new systems, hoping that they will cover and combine better features.

The existing multi-agent platforms use different approaches for communication between agents, by using low-level communication protocols (TCP/IP, SMTP and HTTP) or standard high-level languages – such as KQML (Knowledge Query Manipulation Language) [7].

The *Omega* system presents an advantage, by adopting an XML-based platform-independent approach in serialization and exchanging information between agents. The SOAP model is more flexible and easy to use than CORBA or DCOM solutions. Some of the *Omega*'s facilities could be also integrated in the *MAIS (Mobile Agents Information System)* – a platform for creating dynamic clusters [22].

5 Conclusion and Futher Work

In the software engineering process of creation of the *Omega* framework, we have used the design principles of the distributed systems to develop our own software platforms and ideas related to the multi-agent paradigm and actor spaces [1, 16]. From this point of view, *Omega* represents an infrastructure able to support the agent-oriented programming. By this approach, we emphasize a trend which is shaping the evolution of the software development techniques for open distributed applications.

Although an interesting experience, the design and the implementation of any new programming language that comes with new concepts and

ideas requires much time. *Omega*'s programming language has remained at a prototype level. The experience with *Omega* can be used to understand or/and evaluate the existing and up-coming agent platforms, as well as in design and implementation of future agent-based systems or applications. We intend to experiment an XML-based version of the *Omega* language that can be used to exchange mobile code of the software agents coded within the *Omega* framework. Another possible future approach is to consider an object-oriented language, inspired by MAML (Multi-Agent Modeling Language) – an easy-to-learn Objective-C-like language that can be used to create agent-based models [31].

Presently, we are continuing to extend the ideas used in the *Omega* system by designing and developing a new agent-oriented application, named *AgKA* (*Agent as Knowledge Agent*). This system is intended to be a framework for intelligent Web agents that will act as personal agents for the user. The *AgKA* system will provide a space of distributed objects and a name service for accesing these objects, also. Instead of a tree of objects, we will use a graph that can act as a semantic network and can store the knowledge of the agents. *AgKA* will model the functionality of a peer-to-peer (P2P) system or of a cluster, following the ideas stated in [28].

An important issue in our work is to contribute to the developing of the new programming paradigms like aspect oriented programming, generic programming and, of course, agent oriented programming.

References

[1] G. Agha, C. Callsen, *Actor Spaces: An Open Distributed Programming Paradigm*, Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming, ACM Press, 1993

[2] S. Alboaie, G. Ciobanu, *Designing and Developing Multi-Agent Systems*, in International Symposium on Parallel and Distributed Computing (ISPDC) Proceedings, Scientific Annals of the "A.I. Cuza" University, Computer Science section, Tome XI, "A.I. Cuza" University Press, Iaşi, 2002

[3] D. Beckett, *The Design and Implementation of the Redland RDF Application Framework*, in Proceedings of the 10th World-Wide Web Conference, Hong Kong, ACM Press, 2001

[4] T. Berners-Lee, *Weaving the Web*, Orion Business Books, London, 1999

[5] T. Berners-Lee *et al.* (eds.), *Uniform Resource Identifiers (URI): General Syntax*, Internet Standard, RFC 2396, IETF, 1998

[6] G. Booch, *Object-Oriented Analysis and Design*, Addison-Wesley, 1994

[7] J. Bradshow, *Software Agents*, AAAI Press, 1997

[8] T. Bray *et al.* (eds.), *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation, Boston, 2000: `http://www.w3.org/TR/REC-xml`

[9] D. Brickley, R. V. Guha, *Resource Description Framework (RDF) Schema Specification 1.0*, W3C Candidate Recommendation, Boston, 2000: `http://www.w3.org/TR/2000/REC-xml-20001006`

[10] S. Buraga, *A Model for Accessing Resources of the Distributed File Systems*, in Advanced Environments, Tools and Applications for Cluster Computing Proceedings of the NATO ARW Mangalia, Romania, 1-6 September 2001, D. Grigoraş *et al.* (eds.), Lecture Notes in Computer Science – LNCS 2326, Springer-Verlag, 2002

[11] S. Buraga, *Web Technologies* (in Romanian), Matrix Rom, Bucharest, 2001

[12] S. Buraga, *Web Site Design* (in Romanian), Polirom, Iaşi, 2002

[13] S. Buraga, G. Ciobanu, *A RDF-based Model for Expressing Spatio-Temporal Relations Between Web Sites*, in Proceedings of the 3rd International Conference on Web Information Systems Engineering (WISE 2002), 12-14 December 2002, Singapore, IEEE Press, 2002

[14] S. Buraga, G. Ciobanu, *Programming Workshop in Computer Networks* (in Romanian), Polirom, Iaşi, 2001

[15] S. Buraga, V. Tarhon-Onu, Ş. Tanasă, *Web Programming in bash and Perl* (in Romanian), Polirom, Iaşi, 2002

[16] C. Callsen, *Open Distributed Heterogeneous Computing*, PhD Thesis, University of Illinois at Urbana-Champaign, 1997

[17] W. Conen, R. Klapsing, *A Logical Interpretation of RDF*, in Linköping Electronic Articles in Computer and Information Science, 5, 2000: `http://www.ida.liu.se/ext/epa/cis/2000/013/tcover.html`

[18] S. Decker *et al.*, *Knowledge Representation on the Web*, in F. Baader (ed.), International Workshop on Description Logic (DL'00): `http://www.cs.vu.nl/~frankn/abstracts/DL00.html`

[19] D. Fallside (ed.), *XML Schema*, W3C Recommendation, Boston, 2001: `http://www.w3.org/TR/xmlschema-0/`

[20] C. Gorman, *Programming Web Services with SOAP*, O'Reilly and Associates, 2001

[21] S. Green, F. Somers, *Software Agents: A Review*: `http://www.cs.tcd.ie/research_groups/aig/iag/iag.html`

[22] D. Grigoraş *et al.*, *MAIS – The Mobile Agents Information System Support for Creating Dynamic Clusters*, in Proceedings of ICA3PP, Beijing, 2002

[23] P. Hayes (ed.), *RDF Model Theory*, W3C Working Draft, Boston, 2002: `http://www.w3.org/TR/rdf-mt/`

[24] O. Lassila, R. Swick (eds.), *RDF Model and Syntax Specification*, W3C Recommendation, Boston, 1999: `http://www.w3.org/TR/REC-rdf-syntax/`

[25] R. Lavender, D. Schmidt, *Active Object: An Object Behavioral Pattern for Concurrent Programming in Pattern Languages of Program Design*, Addison-Wesley, 1996

[26] P. Maes (ed.), *Designing Autonomous Agents – Theory and Practice from Biology to Engineering and Back*, MIT Press, 1990

[27] E. Mangina, *Review of Software Products for Multi-Agent Systems*, AgentLink.org, 2002: `http://www.agentlink.org`

[28] L. Moreau, *Agents for the Grid: A Comparison with Web Services (Part I: the transport layer)*, in IEEE International Symposium on Cluster Computing and the Grid Proceedings, Berlin, Germany, May 2002

[29] G. W. Treese, L. Stewart, *Designing Systems for Internet Commerce*, Addison-Wesley, 1998

[30] * * *, *AgentWeb*: `http://www.cs.umbc.edu/agents`

[31] * * *, *Multi-Agent Modeling Language*: `http://www.maml.hu`

[32] * * *, *SOAPware*: `http://www.soapware.org/`

[33] * * *, *Web Object Integration*: `http://www.objs.com/survey/web-object-integration.htm`

[34] * * *, *World Wide Consortium's Technical Reports*, Boston, 2002: `http://www.w3.org/TR/`