

# A Metadata Level for the *tuBiG* Grid-aware Infrastructure

Sabin Buraga      Lenuța Alboaică

Faculty of Computer Science, “A. I. Cuza” University of Iași  
Berthelot, 16 – Iași, Romania  
{busaco,adria}@infoiasi.ro

**Abstract.** The paper proposes the use of a metadata level in the context of designing a Java-based object-oriented system, called *tuBG*, that offers a layered infrastructure to create the adequate framework for complex interactions between Grid components (e.g., agents or Web services). Our proposal can be considered as a first step towards a conceptual model for specifying semantic Grid services, by using Semantic Web actual standards and technologies.

## 1 Preamble

In order to build open, large-scale and inter-operable distributed applications in the context of Grid computing [8, 14], one of the key requirements is the design of an environment in which collaborative distributed applications may be developed in a standardized way [17]. Actually, there are many existing network computing and Grid projects, each of them presenting various (incompatible) architectures and distinct characteristics. The main goal is to design a *generic high-level architecture* able to give support for multiple existing and future protocols, programming languages and standards, in the context of the Semantic Web [4, 10].

The paper investigates the possibility to attach a metadata level to our proposed layered infrastructure – *tuBiG* [1] –, a Java-based object-oriented system. The *tuBiG*'s aim is to create the adequate framework for complex interactions between heterogeneous and geographically distributed components. Using a shared-memory modeled by sets of tuples inspired by Linda [15], the *tuBiG* project presents an abstract communication architecture that can be used to map existing Web-based technologies.

The metadata level will give us the possibility to align *tuBiG* to Semantic Web's current directions, in order to design and deploy semantic Grid services. This first level can be considered the basis for more complex higher levels, such as an ontology level, which will address different problems in modeling of semantic Grid applications and their interactions.

## 2 Background

### 2.1 Grid Technology

The Internet technologies' opportunities have led to the undreamt possibility of using distributed computers as a single, unified computer resource, leading to what is known as Grid computing [8, 14]. Grids enable the sharing, selection, and aggregation of a wide variety of heterogeneous resources, such as supercomputers, storage systems, data sources, specialized devices (e.g., wireless terminals) and others, that are geographically distributed and owned by diverse organizations for solving large-scale computational and data intensive problems in science, engineering and commerce [8].

A three layer model [16, 18] for the Grid infrastructure was adopted by various world-wide research communities. This model views the Grid as an architecture made by:

- the *Computational Grid* – the lower layer concerned with large-scale pooling of computational and data resources that requires significant shared infrastructure to enable the monitoring and control resources in the resulting ensemble;
- the *Information Grid* – this middle layer allows uniform access (via metadata descriptions) to heterogeneous information sources and providing commonly used services running on distributed computational resources; the computational resources can vary, from simple method invocations to complete sophisticated applications;
- the *Knowledge Grid* – the top most layer provides specialised (meta-)services used for data discovery in existing data repositories and for managing information services; the meta-services aggregates many other types of services.

Grid applications are distinguished from traditional Internet applications – mostly based on client/server model – by their simultaneous use of large number of (hardware and software) resources. That implies dynamic resource requirements, multiple administrative domains, complex and reliable communication structures, stringent performance requirements, etc. [8, 14].

There are many actual approaches in building different Grid architectures, at different levels, such as integrated Grid systems (e.g., NetSolve, XtremWeb, Unicore), core middleware (Globus, JXTA, Legion), user-level middleware (e.g., Nimrod-G, Cactus, or GridPort). Also, a number of efforts in building Grid applications can be noticed (for example, European Data Grid, Geodise, Earth System Grid, etc.). More details are available in [8] and [19].

### 2.2 Semantic Web Technology

The actual World-Wide Web space is primary composed on pages (markup documents) with information in the form of natural language text and multimedia

intended for humans to read and understand. Computers are mainly used to render this hypermedia information. Information retrieval has become ubiquitous with the WWW's development and information needs no longer to be intended for human readers only, but also for machine processing, enabling intelligent information services, personalized Web sites, and semantically empowered search engines – this is the seminal idea of the *Semantic Web* [4].

Besides XML (Extensible Markup Language) [25], the central specification of the Semantic Web is RDF (Resource Description Framework) [3], a standardized basis for processing metadata. The RDF is intended to be used to capture and express the conceptual structure of information offered in the Web.

To effectively define metadata, several XML-based languages were proposed. For example, to associate metadata to published Web resources, DCMI (Dublin Core Metadata Initiative) [22] constructs may be used. Web sites' metadata is specified by RSS (Rich Site Summary) [22] elements. In section 4, we use our defined *XFiles* [6] language to denote metadata regarding properties of the *tuBiG* components.

The metadata languages and the general RDF specification make the *meta-data* level of the Semantic Web.

The architecture of Semantic Web implies another two levels [5, 10]:

- the *schema* level gives the possibility to specify simple ontologies in order to define a hierarchical description of concepts and properties. These ontologies can be expressed by RDF Schema [25];
- the *logical* level offers more complex languages, based on description logics, that can model sophisticated ontologies and can help to build reasoning services for the Semantic Web. A typical example is OWL (Web Ontology Language) [12].

Because in this paper we focus on metadata level only, we do not give here any other details (for more information, consult [4], [5], [10] or [25]).

### 3 Short Presentation of the *tuBiG* System

This section summarizes the material presented in [1].

The *tuBiG* platform gives a *virtual addressing space* to be used by distributed applications. This virtual space is formed by a networked set of heterogeneous hosts that agree to share their local resources to others.

The basic model of the system is not a client/server one, each node could send requests to other nodes, but in the same time can resolve requests received from different nodes. We can view our approach as a *many-to-many* one. As particular situations, we can have an *one-to-one* communication – in this case, the *tuBiG* project offers support for peer-to-peer computing – or an *one-to-many* communication – for multicast, anycast or broadcast ways of communication.

### 3.1 *tuBiG* General Architecture

The *tuBiG* infrastructure is a Java-based object-oriented middleware system and provides a layered architecture composed by three main layers [1]:

- *tuBiG – Core* is the central element of the system and includes the services and the features that can make possible the building applications for the Grid layers, in order to give access to distributed heterogeneous resources and users; this layer consists of several components used to implement low-level services for communication, resource management, resource discovery and security.
- *tuBiG – Peel* contains the global services that can be accessed in a public manner by other superior layers; this part is based on the *Core* layer.
- *tuBiG – Interface* is the layer that provides an API (Application Programmer Interface) used to implement high-level functionalities that could be offered by superior layers. One goal of the *tuBiG* project is to provide at the *Interface* layer certain abstractions that hide specific implementation details of each communication protocol and resource management technique. This makes easier the process of design and implementation of software agents that can populate the Grid and a specific agent communication language [17]. The *Interface* layer will be able to give support for implementation of the (semantic) Web services, too.

The implemented resource management mechanism supervises the requests for certain resources stored by the nodes within *tuBiG* addressing space and assigns the requested resource to the client node that made the request.

The *Core* component offers different object datatypes used for the management of the superior layers of the system. We can mention the following datatypes:

- *tuple-object* can be viewed as an atomic addressable information entity and is denoted by a special tuple. This approach is inspired by Linda [15] programming language.  
By using such of tuples, the *Core* layer will store in a particular way data structures used for the (serialized) representation of information (data, request, response) between the nodes of the Grid.  
A tuple can be considered as a generic object of the environment. For example, a tuple can store references to information about a service, a block of data, an address of a node and its access restrictions, etc. Also, a tuple can be considered as a simple container for metadata – see the example provided in section 4.3.
- simple base datatypes, such as *intElement*, *floatElement*, *stringElement* and *pointerElement* – objects that represent the common datatypes provided by any programming language and they are public available to the programmer. The *pointerElement* object represents global references to tuples and will be used to localise tuples in the *tuBiG* space.

- *tupleSpace* is an object complex base datatype that denotes a set of tuples. It is used to store the information of the nodes in a space of tuples. There are many types of the generic tuple space datatype (e.g., *proxyTupleSpaces*) and each of them has certain characteristics and functionalities. This datatype can be viewed as a superclass.

These datatypes are used to model a concurrent distributed programming environment through the shared memory mechanisms.

The global references offer the possibility to access any kind of tuples, without any concerns about their localization.

**Tuple Structure** A tuple can contain different elements of the certain types (e.g., *intElement*, *floatElement*, *stringElement* and *pointerElement*).

Additionally, to uniquely identify a tuple, we'll attach a *tuple identifier* (denoted by a `tuple_id` component) and a *time-stamp*. The time-stamp element denotes the moment of time when that tuple was created and is a *floatElement* datatype element. The time-stamp element is private and cannot be directly accessed by the *tuBiG*'s superior layers.

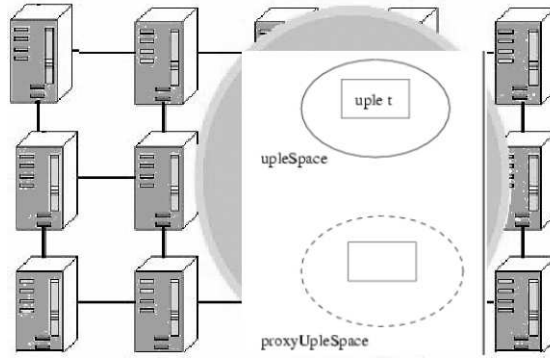
When a tuple needs to be destroyed (its information will be erased), the associated time-stamp will be set to zero. If other remote node (that used in the past this tuple through its reference) tries to have access to the already erased tuple's information, the system will check if the time-stamp is zero. In this case, the reference to the tuple is destroyed, and an event (exception) is generated. Of course, the reference to the tuple will be destroyed if the tuple has a different value for the time-stamp.

The existence of the time-stamp could be used for the implementation of the *tuBiG*'s security mechanism (eventually on the *Interface* layer). Also, the time-stamp component is used in the remote tuple discovery process.

**Node Structure** A node is a tuple subspace that contains a single (“main”) space of tuples (a *tupleSpace* element). This element stores different categories of information provided by the node – a *tupleSpace* element structures all information of a node (see figure 1).

Additionally, a node can contain zero, one or many other “ghost” tuple spaces (of *proxyTupleSpaces* datatype) that will be created at the communication time between this node and other remote nodes of the system. The “ghost” tuple spaces are partial copies of the “main” tuple spaces of the remote nodes involved in the communication process. For each remote tuple, a “ghost” tuple is locally used – this approach is somewhat similar to stub/proxy/skeleton entities of RPC (Remote Procedure Call) or RMI (Remote Method Invocation).

The *tupleSpace* and *proxyTupleSpaces* elements are stored by another space of tuples, called *knownSpaces*. The virtual common space presented below consists of these tuple spaces of the *knownSpaces* type for each existing node of the Grid. Each component of a set of *tupleSpace* and *proxyTupleSpaces* is “aware” of its occurrence in a *knownSpaces* space of tuples.



**Fig. 1.** *tuBiG* Node Structure

**Communication Mechanism** The communication mechanism consists of a set of request/response pairs of tuples.

At the level of *tuBiG Core*, the process of communication is accomplished in an asynchronous manner, because there are requests that do not need an (immediate) response. Internally, the responses to requests consist of different requests which can imply the change of one or many tuples.

A *request tuple* has the following form. First three reserved fields contains the name of the desired service (service type), a handler which will identify the request (request identity), and a state field.

The state field indicates the success or the failure of a request and is used to implement the event-based mechanism. The state field can contain different state values about the progress of the request (e.g., “START”, “IN PROGRESS”, “FINISHED” or “ERROR”).

The synchronization could be made on the basis of these state values and of the request identity, using specific synchronization abstractions.

Also, the request tuple can contain other fields of different base datatypes or pointers to other tuples, in order to invoke the desired service. The request tuple is uniquely identified by its time-stamp component and cannot be duplicated.

When a request is made, a request-type object is created. The management of the request-type objects is implemented by means of an internal request queue. The request-type objects are synchronized by using specific synchronization primitives.

On the *Peel* layer, the infrastructure gives the possibility to have two types of requests, in order to implement the *asynchronous* and *synchronous* communication – for more details (including a conclusive example), consult [1]. In both cases, the *tuBiG*'s event mechanism is used to signal the state of the completed request.

The *tuBiG* infrastructure offers a powerful and flexible resource management mechanism that can be used to implement sequential or/and parallel applications.

Using our own *tuBiG*'s capabilities of the remote invocation of the services (via tuples), a node could invoke an operation (e.g., Web service or a method of a remote agent) which its returned result can be placed on any other existing node, if the security mechanism allows this. This feature is useful in the case of a wireless Grid to discover and execute/consume different available services/resources.

Similarly to TSpaces [20] approach, a node can register for *event notifications* – for example, “let me know when a certain tuple (with a desired content) is written/updated to the space”. When an event occurs, the node is notified through a *callback method*. This method is denoted by a reference to another tuple that can contain information about the invocation of another operation.

### 3.2 Implementation

The *tuBiG* project – actually, in the stage of prototype – was developed exclusively in Java, using Java 2 Standard Edition Development Kit 1.4 [24].

For data serialization, we are using Java specific serialization mechanism and our XML-based serialization techniques (for details, see [2]). At the implementation level, the synchronization and event notification mechanisms use the `wait()` and `notify()` primitives.

An XML-based configuration file is available on each node to store the IP and port information about the nodes of the Grid. Also, this file sets certain access restrictions and stores meta-descriptions to each functionality provided by the node. Future implementations will use the configuration file for service naming and discovery.

The prototype was tested on the Microsoft Windows 2000/XP platforms and on the several Linux distributions (such as Redhat 9, Fedore Core 1 and Mandrake 9.x), using a Linux 2.4 kernel.

## 4 A Metadata Level for the *tuBiG* Infrastructure

The section is presenting the use of metadata within *tuBiG* tuples and nodes.

### 4.1 Associating Metadata to Tuples

In order to give support for metadata, we'll associate additional information to certain tuples.

As we seen in section 3.1, the *tuBiG* infrastructure offers the possibility to store metadata within a tuple. In order to associate metadata to tuples, we extend tuple structure with a new component, called *metadata*, which type is the *pointerElement* base datatype. The metadata element refers to a tuple or set of tuples that will represent the associated metadata for the considered tuple.

To gain a maximum of flexibility, we do not restrict metadata to tuples of special datatypes. Thus, at a superior layer, the *tuBiG* infrastructure can have different metadata (in various representations, as we'll see in section 4.3) depending on the purposes of the Grid applications. For example, the metadata can be used for the security mechanism, tuple discovery process or data retrieval.

## 4.2 Associating Metadata to Nodes

Using a similar approach, we can associate metadata to certain nodes. Within the *tupleSpace* element, metadata is stored as a reference to a set of metadata tuples (of *metadataTuplesSpaces* datatype). This metadata is useful to store information regarding node's access privileges, communication types, or the general state of a *tuBiG* node – to give only few examples.

At the superior layers, metadata can be used in different contexts, such as process scheduling, resource monitoring, remote communication, security, etc.

## 4.3 Representing Metadata

From the previous sections, we can note that the lower layers of the *tuBiG* architecture include metadata as references to special tuples. At the *Interface* layer, we need a common representation of associated metadata. To accomplish this goal, we propose the use of XML-based languages, such as RDF (Resource Description Framework) [3, 25], DCMI (Dublin Core Metadata Initiative) [22] or *XFiles* [5, 6].

Using these languages, it is easy to map existing metadata tuples to XML constructs. Because *XFiles* is suitable to store metadata regarding distributed file systems and other information concerning files, this format can be mainly used to maintain *tuBiG* node's metadata.

Also, the RDF/XML-based languages can be considered in exchanging information (including metadata) processes between certain components of the *tuBiG* architecture, following the approach described in [7].

**Example** In [1] we give an example of searching digital pictures stored on a number of nodes that form a Grid. The physical localization of the persons that hold these pictures could be distributed on the entire planet. The nodes of the Grid will run *tuBiG* software.

At the *Core* layer, we need to discover all tuples that have as data elements three fields (of *stringElement* base datatype) contained the type of desired content (“image”), its format (“JPEG” file) and associated metadata (e.g., “taken with a digital camera by Dana Petcu”). The first component is the tuple identifier, followed by other three elements that are the service type (“match-tuple”), the request identifier (e.g., 133), and the state field (i.e. “IN PROGRESS”). We omit the timestamp information.

```
{
  tuple_id = 23412,
  service_type = match-tuple,
  req_id = 133,
  state = IN PROGRESS,
  content = image,
  type = JPEG,
  metadata = *25674
}
```

To simplify, we consider metadata stored within another tuple, identified by an unique number, and we note with “\*” the pointer to this tuple. The metadata tuple will store the information about the digital camera and the person which taken photos:

```
{
  tuple_id = 25674,
  service_type = get_metadata,
  keyword = metadata,
  value = Digital Camera Dana Petcu
}
```

At the *Interface* layer, the metadata will be stored by XML documents. For example, we can have the following RDF statements that include *XFiles* elements:

```
<rdf:Description rdf:about="#request">
  <xf:Properties>
    <!-- metadata regarding the node
         that will process the request -->
    <xf:Owner>
      <rdf:Description rdf:about="#tuBiG">
        <!-- host localization -->
        <xf:Location xf:dns="thor.infoiasi.ro">
          193.231.30.225
        </xf:Location>
        <!-- security info -->
        <xf:Permissions>
          ...
        </xf:Permissions>
      </rdf:Description>
    </xf:Owner>
    <!-- information regarding the desired image -->
    <xf:Type xf:mime="image/jpeg">
      ordinary
    </xf:Type>
    <xf:Meta xf:regexp="no">
      Digital Camera
    </xf:Meta>
    <xf:Meta xf:regexp="no">
      Dana Petcu
    </xf:Meta>
  </xf:Properties>
</rdf:Description>
```

The *xf* is the namespace used by *XFiles* constructs and *rdf* denotes RDF elements and attributes.

We are extending *XFiles* language with a new *Meta* element that will store certain metadata (here, simple strings). To gain flexibility, we offer the possibility of using Perl-style regular expressions. Of course, the given example can be refined to better express and structure associated information.

## 5 Advantages and Related Work

The *tuBiG* system can be considered as a software infrastructure and a test-bed solution for any layer of the Grid, by providing support for building complex solutions to the dynamic services required by each of these levels.

The applications – e.g., peer-to-peer software, cluster and Grid components, wireless network services, portals, etc. – built on the *tuBiG Interface* layer can use different computing and communication technologies at the Internet scale. Using both synchronous and asynchronous types of communication via request/response tuples, the system is flexible enough to use multiple communication paradigms (using agents, Web services or/and a mixture of these). From this point of view, our proposal is similar with the Globus [23] and OGSA (Open Grid Services Architecture) [11] – that is based on the assumptions that Grid architectures should provide basic services, but not impose particular programming models or higher-level computing architectures and Grid applications require services beyond those provided by today’s usual technologies.

Because *tuBiG* is fully object-oriented, its architecture is related to Legion’s metacomputing framework [9, 11]. The *Core* and *Peel* layers of the *tuBiG* system manipulate tuples that cannot be accessed as objects by the entities of the superior layers. The object-oriented aspect of the tuples is revealed only at the *Interface* layer or other superior layers based on the *tuBiG*’s API.

The *tuBiG*’s architecture is more a service-oriented architecture [11] and its *Interface* layer gives support for Web services and related XML-based technologies (e.g., SOAP, WSDL, UDDI) [25], and as well for software agents.

Another important aspect in decentralized systems is *information discovery*. By using – at the *Interface* layer of the *tuBiG* infrastructure – the Perl-like regular expressions, XML-based query languages and RDF semantic descriptions, programmers will be able to design and deploy programs (e.g., Web agents or services) for resource and service discovery. Using associated metadata and different query techniques (see also [5]), we intend to develop a test application used for discovering multimedia resources within a Grid.

The proposed metadata level can offer support for the applications situated on the *Information Grid* layer mentioned in section 2.1. Also, the metadata level can be useful for Grid-like portals that integrates various distributed information and resources. These portals can be based on *tuBiG* infrastructure, too.

## 6 Conclusion and Further Work

In this paper, a metadata level was proposed to be used in conjunction with *tuBiG* infrastructure, a Java-based object-oriented system that offer support for

building Grid-aware heterogeneous and geographically distributed components. The aim of the project is the platform-, language- and communication protocol-independence.

The paper was presented the use of metadata in different cases, at all three layers – *Core*, *Peel* and *Interface* – of the system. In section 4, we described the mechanism of associated metadata to the internal tuples of the *Core* layer. Also, we gave details regarding the use of metadata on *tuBiG*'s nodes and how we can represent this metadata at the *Interface* layer. For this, an XML-based approach is proposed, adopting different metadata languages, such as RDF and *XFiles*.

Providing a flexible layered architecture, the *tuBiG* system can be used to effectively realize a Grid [8], including – among other facilities – the deployment of low-level middleware in order to offer a secure and transparent access to resources and the development and testing of distributed applications to take advantage of the available resources and infrastructure.

Future versions of *tuBiG* will make possible to develop the next generation of user agents (RSS aggregators, semantic Web browsers, etc.) that could take advantage of the tuple semantics and attached metadata.

We consider the metadata level as a foundation for other superior levels, such as the ontology level, based on Semantic Web specifications [3, 12, 25]. The *tuBiG* infrastructure should offer the possibility of designing and deployment of semantic Grid services. From this point of view, *tuBiG* can be considered as a test-bed architecture for Semantic Web applications. One of the important future directions – stated in [19] and [21] – is to give support for building Grid-based (in our vision, *tuBiG*-based) ontological services.

## References

1. L. Alboaie, S. Buraga, S. Alboaie, “*tuBiG* – A Layered Infrastructure to Provide Support for Grid Functionalities”, in M. Paprzycki (ed.), *Proceedings of the 2nd International Symposium on Parallel and Distributed Computing (ISPDC'03)*, IEEE Computer Society Press, 2003
2. S. Alboaie, S. Buraga, L. Alboaie, “An XML-based Serialization of Information Exchanged by Software Agents”, *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics – SCI 2003*, Orlando, USA, 2003
3. D. Beckett (ed.), *RDF/XML Syntax Specification (Revised)*, W3C Recommendation, Boston, 2004: <http://www.w3.org/TR/rdf-syntax-grammar>
4. T. Berners-Lee, J. Hendler, O. Lassila, “The Semantic Web”, *Scientific American*, 5, 2001
5. S. Buraga, *Semantic Web. Foundations and Applications* (in Romanian), Matrix Rom, Bucharest, 2004
6. S. Buraga, “A Model for Accessing Resources of the Distributed File Systems”, in D. Grigoraş *et al.* (eds.), *Lecture Notes in Computer Science – LNCS 2326*, Springer-Verlag, 2002
7. S. Buraga, S. Alboaie, L. Alboaie, “An XML/RDF-based Proposal to Exchange Information within a Multi-Agent System”, in D. Grigoraş *et al.* (eds.), *Post-Proceedings of NATO ARW on Concurrent Information Processing and Computing*, IOS Press, 2004 (to appear)

8. R. Buyya, "Economic-based Distributed Resource Management and Scheduling for Grid Computing", *PhD Thesis*, Monash University, Melbourne, Australia, 2002
9. S. Chapin, J. Karpovich, A. Grimshaw, "The Legion Resource Management System", *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing at IPDPD'99*, San Juan, Puerto Rico, 1999
10. J. Davies, D. Fensel, F. van Harmelen (eds.), *Towards the Semantic Web*, John Wiley & Sons, England, 2003
11. D. De Roure *et al.*, "The Evolution of the Grid", *International Journal of Concurrency and Computation: Practice and Experience*, 2003
12. M. Dean, G. Schreiber (eds.), *OWL Web Ontology Language Reference*, W3C Recommendation, Boston, 2004: <http://www.w3.org/TR/owl-ref/>
13. M. Fontoura *et al.*, "TSpaces Services Suite: Automating the Development and Management of Web Services", *Proceedings of WWW 2003 Conference*, ACM Press, 2003
14. I. Foster, C. Kesselman (eds.), *The Grid: Blueprint for a Future Computing Infrastructure*, Morgan Kaufmann Publishers, 1999
15. D. Gelernter, "Multiple Tuple Spaces in Linda", in J. G. Goos (ed.), *Lecture Notes in Computer Science – LNCS 365*, Springer-Verlag, 1989
16. K. G. Jeffery, "Knowledge, Information, and Data", A briefing to the Office of Science and Technology, UK, 2000:  
<http://www.itd.clrc.ac.uk/ActivityPublications/239>
17. L. Moreau, "Agents for the Grid: A Comparison with Web Services (Part I: Transport Layer)", in *IEEE International Symposium on Cluster Computing and the Grid Proceedings*, Berlin, Germany, 2002
18. O. F. Rana, L. Moreau, "Issues in Building Agent-based Computational Grids", *Third Workshop of the UK Special Interest Group on Multi-Agent Systems – UK-MAS 2000*, Oxford, UK, 2000
19. K. Sycara, T. Payne, "Agent Mediated Semantic Web/Grid Services", *Second International Semantic Web Conference*, ACM Press, 2003
20. P. Wyckoff *et al.*, "TSpaces", *IBM Systems Journal*, 37 (3), 1998
21. \* \* \*, *Bulletin of AIS Special Interest Group on Semantic Web and Information Systems*, Vol. 1, No. 1, April 2004
22. \* \* \*, *Dublin Core Metadata Initiative*: <http://dublincore.org/>
23. \* \* \*, *Globus Project*: <http://www.globus.org/>
24. \* \* \*, *Java Software*: <http://www.javasoft.com/>
25. \* \* \*, *World-Wide Web Consortium's Technical Reports*, Boston, 2004:  
<http://www.w3.org/TR/>