

tuBiG – A Layered Infrastructure to Provide Support for Grid Functionalities

Lenuța Alboai
Institute of Theoretical Computer Science
Romanian Academy, Iași branch
adria@iit.iit.tuiasi.ro

Sabin C. Buraga
Faculty of Computer Science
“A.I. Cuza” University of Iași, Romania
busaco@infoiasi.ro

Sînică Alboai
Institute of Theoretical Computer Science
Romanian Academy, Iași branch
abss@iit.iit.tuiasi.ro

Abstract

The paper presents a Java-based object-oriented system that offers a layered infrastructure to create the adequate framework for complex interactions between Grid components (e.g., agents of Web services). Using a shared-memory modeled by sets of tuples, our proposal can be considered as an abstract communication architecture for building Grid services.

1. Introduction

The actual 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 [5, 8]. 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 [5].

In order to build open, large-scale and inter-operable distributed applications in the context of Grid computing, one of the key requirements is the design of an environment in which collaborative distributed applications may be developed in a standardized way [13]. 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 this paper, we propose a Java-based object-oriented system – *tuBiG* (*Tuple-Based Grid*) – that offers a layered infrastructure to create the adequate framework for complex interactions between heterogeneous and geographically distributed components. The authors consider *tuBiG* architecture as a natural continuation of a previous project, called *Omega* [2, 3].

Using a shared-memory modeled by sets of tuples inspired by Linda [9], the *tuBiG* project presents an abstract communication architecture that can be used to map existing communication Web-based technologies. Our proposed infrastructure can be used to design, implement and test various distributed components, such as agents or (semantic) Web services.

2. *tuBiG* Infrastructure

In order to offer the best functionality and performance, *tuBiG* system gives a *virtual addressing space* to be used by distributed applications. This virtual space is formed by a networked set of heterogeneous hosts (real or virtual machines denoted by an IP address) 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.

2.1. System Architecture

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

- *tuBiG – Core* is the central element of the system and contains the services and the features that can make possible the building applications for the Grid layers (see also section 3) in order to offer 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 [13]. The *Interface* layer will be able to give support for implementation of the 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 the following object datatypes used for the management of the superior layers of the system:

- *tuple-object* can be viewed as an atomic addressable information entity and is denoted by a special tuple. This approach is inspired by Linda [9] – a programming language originally constructed to handle the problem of writing parallel programs for massively-parallel supercomputers.

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.

- 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., *proxy-TupleSpaces*) and each of them has certain characteristics and functionalities detailed in next section. This datatype can be viewed as a superclass.

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

For a direct access to tuples, we are using the local references provided by Java [18] language at the internal implementation level. The global references offer the possibility to access any kind of tuples, without any concerns about their localization.

Tuple Structure

We'll consider the following model to describe tuples. A tuple can contain different elements of the certain types: *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*. In Linda jargon, these fields are "formals" – they do not store public data [9]. 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 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.

We can remark that 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.

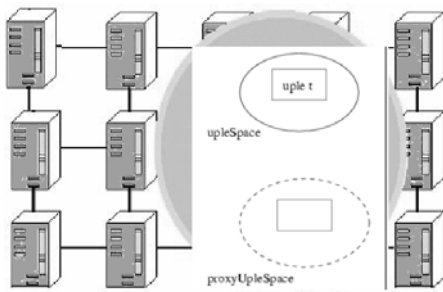


Figure 1. Node Structure

Node Structure

In the proposed infrastructure, 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 mechanism is somewhat similarly to stub/proxy/skeleton entities of the RPC or CORBA).

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 appartenance to a *knownSpaces* space of tuples.

Because the tuple spaces are included into other tuple spaces, apparently the organizational model of the system is a hierarchically one. In fact, the internal structure of the system is a shared, flat space of tuples. The nodes can directly communicate to others, without intermediary nodes. This fact is very useful to give the possibility to implement various Grid-like services at the *Interface* layer or other superior layers.

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 fields are reserved and they will contain: the name of the desired service (service type), a handler which will identify the request (request identity), and a state field.

The state field will indicate the success or the failure of a request and will be used to implement the event-based mechanism. The state field will contain different state values about the progress of the request, such as “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.

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 two mechanisms of communication:

- *asynchronous* – for this kind of request, we do not need to wait for a response (upon the request-type objects, the synchronization primitives are not invoked);
- *synchronous* – for each request, a response is mandatory; the internal implementation uses our own mechanisms (the internal synchronization primitives are invoked) to assure the synchronous transfer of information.

In both cases, the *tuBiG*'s event mechanism is used to signal the state of the completed request.

Example

To have a whole picture of the communication phases, we imagine the following scenario.

We suppose that, after previous edition of ISPDC event, there is a number of 33 participants which have stored digital pictures taken during the symposium. We want to discover all JPEG images stored on the laptops, palmtops or PCs of these participants. We consider these computers form a Grid (the physical localization of the persons could be distributed on the entire planet). The nodes of the Grid run our *tuBiG* software.

To simplify the situation, we assume the *B* node makes a synchronous request to the *A* node of the Grid to invoke a service of picture file discovery, where *A* is one of the Grid nodes. This discovery service returns tuples that will

contain the desired information (e.g., the name of the found files, plus their URI, the access rights and other metadata – for example: owner, MIME type, filesize, picture resolution and quality etc.). The *B*'s request is a *req_t* tuple which contains all information needed by *A* to fulfill the request and, eventually, to sent the response back to the *B* node.

In the *B*'s reserved space of tuples, the *req_t* tuple contains the name of the requested service that will be invoked on *A* node and a reference to a *res_t* tuple. The *res_t* tuple is the response tuple that will store information about the response given by *A*. The *res_t* tuple has a reference to a list of tuples that will denote the response (the set of found pictures).

We need to discover all tuples that have as data elements three fields (of *stringElement* base datatype) contained the data category (“image”), its type (“JPEG” file) and associated metadata (“ISPDC 2002 event”). The first three elements are the service type (“match-tuple”), the request identity (e.g., 133), and the state field (i.e. “IN PROGRESS”):

```
( service_type = match-tuple,
  req_id = 133,
  state = IN PROGRESS,
  content = image,
  type = JPEG,
  metadata = ISPDC 2002 event )
```

Instead of given strings, we can use Perl-style regular expressions to match all tuples that have three data elements of *stringElement* datatype.

The steps involved in the process of communication are (see also figure 2):

1. the request is initiated;
2. a proxy space of tuples is created on *B* and the communication mechanism is started;
3. similarly, a proxy space of tuples is created on *A* and a temporarily *t* tuple is created; this tuple corresponds to *res_t* tuple stored on *B* node; the *t* tuple will be used to bind the service requested by *B* to be invoked on *A*;
4. the *A* node executes the request and the pointer to the result is stored by *t* tuple; the system initiates an update mechanism that implies the updating process of the information stored by *res_t* tuple on the *B* node (in fact, this step is very similar to the RPC communication mechanism);
5. the result is returned in *res_t* tuple and a “FINISHED” event is triggered. In this moment, the system will notify the request-type object that had monitored the whole process of communication to continue the *B*'s execution.

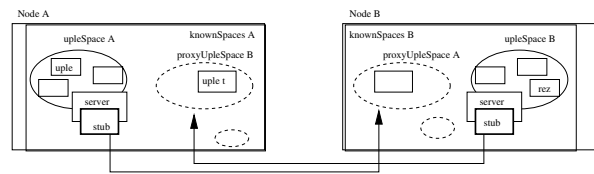


Figure 2. Process of communication between nodes *A* and *B*

For the asynchronous communication case, using the event mechanism, the *B* node could be notified if the response from *A* was arrived.

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 [16] approach, a node can register for *event notifications*, such as “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.

2.2. Implementation

The *tuBiG* project – actually, in the stage of prototype – is developed exclusively in Java, using Java 2 Standard Edition Development Kit 1.4 [18]. The XML data is processed by using the Document Object Model (DOM) [20] library of the World-Wide Web Consortium.

For data serialization, we are using Java specific serialization mechanism and our XML-based serialization techniques (for details, see [1]). 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 and Mandrake 9.1), using a Linux 2.4 kernel.

3. Related and Further Work

A three layer model [10, 14] 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.

In this context, 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 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 project [17] and its successor – the Open Grid Services Architecture (OGSA) [12] – 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. The OGSA supports the creation, deployment, and application of ensembles of services maintained by virtual organizations.

Because *tuBiG* is fully object-oriented, its architecture is related to Legion's metacomputing framework [6, 12]. 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.

Another related application is TSSuite [7] – a tuple-based infrastructure and a set of tools for the development and management of Web services. In fact, our tuple-based model is similar to TSpaces [16] approach – a network communication buffer with database capabilities that enables communication between applications and devices in a network of heterogeneous computers and operating systems.

One of the promising Java-based technologies for building Grid components is JINI [19], using the concept of network plug-and-play. Our proposal do not use JINI or other similar approaches, but at the internal implementation level has many similar functionalities. The *tuBiG*'s architecture is more a service-oriented architecture [12] and its *Interface* layer gives support for Web services and related XML-based technologies (e.g., SOAP, WSDL, UDDI) [20], and as well for software agents.

Another important aspect in decentralized systems is *information discovery*. By using the Perl-like regular expressions, XML-based query languages and RDF semantic descriptions, at the *Interface* layer of the *tuBiG* infrastructure, programmers will be able to design and deploy programs (e.g., Web agents or services) for resource and service discovery. Using different query techniques (see also [4]), we intend to develop a test application used for discovering multimedia resources within a Grid.

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

As stated in [13], a number of Grid systems make use of Web services as an XML-based communication layer. On the other hand, there are many agent-based approaches [11, 15, 14] for building different kinds of Grid applications. Of course, there are mixed implementations, too. The *tuBiG* project's goal is to support the designing and implementation of both approaches.

4. Conclusion

In this paper, a Java-based object-oriented system, called *tuBiG*, was proposed as a software infrastructure used to build a Grid-like environment for complex interactions between heterogeneous and geographically distributed components. Our proposal provides support for building solutions to the dynamic services required by the three layers – Computational, Information, and Knowledge – of the Grid. The aim of the project is the platform-, language- and communication protocol-independence.

The paper is focused on the general architecture of the system and gives certain details about design rationale, communication mechanisms and implementation is-

sues. The *tuBiG* system consists of three layers, detailed in section 2.1. The *Core* layer offers several components used to implement low-level services for communication, resource management, resource discovery and security. The *Peel* layer contains the global services intended to be publicly accessed by other superior layers. The *Interface* layer provides an API used to implement high-level abstract functionalities that could be offered by other superior layers.

To make possible the communication between remote nodes, the system uses sets of tuples (inspired by Linda) that resembles a common shared addressable memory. The paper presents the general structure of the involved tuples and describes the tuple spaces used to store different categories of information provided by a node.

The communication mechanism consists of a set of request/response pairs of tuples (see details and an example in section 2.1). The Java internal implementation of the system is discussed in section 2.2.

Providing a flexible layered architecture, the presented *tuBiG* project can be used to effectively realize a Grid [5], 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.

References

- [1] S. Alboaic, S. Buraga, and L. Alboaic. An XML-based serialization of information exchanged by software agents. In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics – SCI 2003*, 2003.
- [2] S. Alboaic and G. Ciobanu. Designing and developing multi-agent systems. In D. Grigoraş, editor, *International Symposium on Parallel and Distributed Computing (ISPDC) Proceedings*, volume Scientific Annals of the “A. I. Cuza” University, Tome XI of *Computer Science*, Iaşi, Romania, 2002. “A. I. Cuza” University Press.
- [3] S. Buraga, S. Alboaic, and L. Alboaic. An XML/RDF-based proposal to exchange information within a multi-agent system. In D. Grigoraş et al., editors, *Proceedings of NATO Advanced Research Workshop on Concurrent Information Processing and Computing*. IOS Press, 2003. To appear.
- [4] S. Buraga and M. Brut. Different XML-based search techniques on web. In *Transactions on Automatic Control and Computer Science*, 47 (61) – 2, Timişoara, Romania, 2002. Politehnica Press.
- [5] R. Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Computing*. PhD thesis, Monash University, Melbourne, Australia, 2002.
- [6] S. Chapin, J. Karpovich, and A. Grimshaw. The legion resource management system. In *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing at IPDPD’99*, 1999.
- [7] M. Fontoura et al. TSpaces services suite: Automating the development and management of web services. In *Proceedings of World-Wide Web 2003 Conference*. ACM Press, 2003.
- [8] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.
- [9] D. Gelernter. Multiple tuple spaces in Linda. In J. G. Goos, editor, *Proceedings of PARLE’89*, Lecture Notes in Computer Science – LNCS 365. Springer-Verlag, 1989.
- [10] K. G. Jeffery. Knowledge, information, and data. In *A briefing to the Office of Science and Technology*, UK, 2000. URL: <http://www.itd.clrc.ac.uk/ActivityPublications/239>.
- [11] M. L. Kahn and C. D. T. Cicalese. CoABS grid scalability experiments. In *Second International Workshop on Infrastructure for Scalable Multi-Agent Systems at Autonomous Agents*, Montreal, Canada, 2001.
- [12] D. De Roure, N. Jennings, and N. Shadbolt. The evolution of the grid. *International Journal of Concurrency and Computation: Practice and Experience*, 2003.
- [13] 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.
- [14] O. Rana and L. Moreau. Issues in building agent-based computational grids. In *Third Workshop of the UK Special Interest Group on Multi-Agent Systems – UKMAS 2000*, Oxford, UK, 2000.
- [15] O. Rana and D. Walker. “The Agent Grid”: Agent-based resource integration in PSEs. In *Proceedings of the 16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, Lausanne, Switzerland, 2000.
- [16] P. Wyckoff et al. TSpaces. *IBM Systems Journal*, 37(3), 1998.
- [17] Globus Project. URL: <http://www.globus.org>.
- [18] Java Software. URL: <http://www.javasoft.com>.
- [19] JINI. URL: <http://www.jini.org>.
- [20] World Wide Consortium’s Technical Reports, 2003. URL: <http://www.w3.org/TR/>.