A Tuple Space Service for Large Scale Infrastructures

Sirio Capizzi and Antonio Messina
University of Bologna
Department of Computer Science
Mura Anteo Zamboni 7, 40126 Bologna ITALY
Email: {capizzi, messina}@cs.unibo.it

Abstract—Coordinating tasks is one of the main activities of every Grid middleware. There are several ways to achieve this goal, for example using publish/subscribe systems or workflow engines. In this article we will analyse another possible model: tuple spaces. The tuple space model has been successfully used for years for coordinating tasks in computational applications and has interesting characteristics like the support to open systems, allowing an application to scale without the need to reimplement it. This article will describe our experience in developing a tuple space system for the Globus Toolkit using Web Services.

I. INTRODUCTION

Coordinating tasks is one of the main activities of every Grid middleware. There are several ways to achieve this goal, for example using publish/subscribe systems or workflow engines (see, for instance, [1] and [2] for a review). Another possible approach to coordinate tasks is the use of tuple spaces [3]. This model has been introduced several years ago and has been successfully used for the coordination of numerical applications.

The main features of the tuple space model are:

- **Dynamic Membership**: services can join or leave the distributed application without the use of any kind of protocol and any influence on the application (if properly developed).
- **Inherent Load Balancing**: clients can be programmed to request tasks, execute them and store the result into the space. In this way, the more powerful ones will execute more tasks thus leading to a sort of load balancing without complex support.
- **Declarative Approach**: there is no need to specify which host will take or write a specific data item: it is simply inserted or removed from the space using generic queries. This can simplify the development and execution of applications, abstracting them from the network topology and allowing reconfiguration of the network without changes in the application code.
- **Powerful interface**: simple yet powerful interface allows distributed applications to be rapidly developed using few simple operations supporting both asynchronous and synchronous behavior.

We believe that these characteristics can be very useful in a large scale infrastructure. For this reason in this article we will describe our experience in developing a tuple space service named Grinda for the Globus Toolkit 4 [4]. This service has been designed as a generic coordination framework for applications and other services. Grinda can be used in many different ways. For example, applications can use it directly to coordinate their internal tasks whereas an index service can use it to store information about resources profiting by its implicit scalability due to the abstraction from network topology.

This article is organised as follows: section II describes briefly the tuple space model and the changes we have made in order to support the Globus Toolkit. Section III describe the overall architecture of our system whereas in section IV we explain some of the major implementation details we have employed. Finally section V describes our preliminary tests in order to evaluate the performance of the service and section VI makes some conclusive remarks.

II. THE TUPLE SPACE MODEL

The Tuple Space model has been proposed by Gelernter and Carriero as coordination model for distributed application [3]. It is based on the concept of a virtual shared memory, the tuple space, on which the various hosts can operate using a limited number of synchronous and asynchronous operations.

These operations are:

- **out** that write a tuple in the space
- **in** that synchronously remove a tuple from the space that matches the given template. If no such tuple is found the application waits until a matching one is inserted.
- **rd** that synchronously read a tuple from the space that matches the given template. Like the previous operation, if no tuple is found the application waits until a matching one is inserted.

The non blocking version of **in** and **rd**, **inp** and **rdp** respectively, are also defined.

Tuples are usually ordered arrays of values whereas templates can also contain wildcards used by the matching operations: two tuples match if they have the same length and every corresponding pair of elements has the same type or the same value.

In the past years several tuple space implementations have been proposed. The most important are TCP Linda [5], the last incarnation of the original model, TSpace [6], JavaSpaces [7], GigaSpaces [8], Lime [9], Tucson [10] and many others.
Each of these implementations has added peculiar features to the original model.

The tuple space model we have employed in our system is inspired by JavaSpaces. It uses full fledged objects as tuples instead of the usual array structure. We have decided to use this approach for two reasons:

- it simplifies the developing of applications because it doesn’t need additional code to map data to tuples. Moreover, tuples in Linda often require additional labels or values to be correctly taken from the space. This is fault-prone since an error in setting them could lead to malfunctions.
- it maps better into XML messages used by the Globus Toolkit for the communication and requires less effort for the serialization.

The matching operation of our model is defined in the following way. A tuple $t$ of type $\tau$ matches another one $t'$ of type $\tau'$ if $\tau$ and $\tau'$ are the same type or $\tau$ is a super type of $\tau'$ and each non null attribute of $t$ matches the corresponding one in $t'$.

Using this model, normal objects can be directly written or taken from the space without manual conversions. Moreover, the subtype matching can be used to simplify the developing of applications. In fact it is possible, for example, to define a generic task type and other more specific task subtypes that inherit from the generic one implementing more specific behaviours. In this way, the system promotes modularisation, because only specific behaviours should be coded, whereas generic one can be reused wherever possible. Moreover, Grinda does not impose specific interfaces for tuples as JavaSpaces: every data type can be used as a tuple. Details on how this is achieved are described in Section IV.

III. ARCHITECTURE

The architecture of the Grinda service is composed by two main modules: a client-side module and a server-side one.

The main purpose of the client-side module is to hide the details of the communication with the server, in order to simplify the development of applications based on Grinda. It contains three categories of objects: the objects used directly by the applications to operate on the space, those involved in the serialization/deserialization of the data to/from XML and finally those employed to mimic the synchronous operations. The serialization objects have been designed to allow different types of serialization strategies to be loaded at run-time.

The server-side module contains the logic responsible for storing the tuples and for implementing the tuple space operations. A factory, GrindaContext, is used to create and manage tuple spaces according to the configuration. To be more flexible as possible, the tuple space operation logic (defined in subclasses of TupleBroker) is separated from the tuple storage implementation (defined in subclasses of TupleSpace). In this way components are better modularised and they can be mixed together to meet application requirements. For example, it would be possible to create distributed tuple spaces with a persistent storage of tuples.

Although different behaviours of tuple spaces are possible, they are completely transparent for the clients that use the same interface to operate on them.

IV. IMPLEMENTATION

The Grinda server-side module is a Web Services implemented in Java and uses the WSRF framework provided by the Globus Toolkit.

The client-side module is designed to be loosely-coupled with the service, allowing the use of different libraries or programming languages for a better integration with the application. Actually, two different client-side modules have been implemented: one in Java and the other in C++. Both share a similar architecture with some differences that will be discussed later in this section. These two different type of clients provide the possibility that both new and legacy applications interact with the tuple space.

The implementation of Grinda shows various interesting aspects. The most important ones are the service implementation, the transparent serialization of tuples, and the implementation of the C++ client-side module. They will be analysed in details in the following sections.

A. Service Implementation

As stated previously, the server-side module is based on the WSRF specification [11] and thus it could have also employed the factory pattern provided by this specification. However, we have discovered that following this approach the performance of the system is more than halved. This is probably due to an inefficient implementation in the Globus Toolkit. For this reason our service does not use a WSRF-based factory.

To implement the synchronous behavior of some tuple space operations, like take or read, we have used a notification approach based on the WSN [12] specification. Actually, it is impossible to implement directly synchronous operations using only Web Services interactions: socket timeouts and a correct network programming do not allow it. Thus, we have employed a client-side thread and a register, that maintain and handle synchronous operation requests and unblock applications when needed.

Unfortunately, we were unable to use all available WSN features for this purpose, due to some limit in the Globus Toolkit implementations. Actually, according to the WSN
specification, the Grinda service could have exposed a standard NotificationProducer interface allowing direct registration to topics, each of which representing a space. Filters could have been used to select the tuples. However, the WSN registration module does not implement an adequate API for custom registration and notification mechanisms and filters are not implemented. Thus, to support this architecture, an almost complete reimplementation of the WSN module would be needed.

As discussed early the operation logic and the tuple storage are implemented in separate classes in order to provide more flexibility. They will be instantiated at runtime according to the provided configuration.

Until now, the only tuple space operation logic we have implemented is a centralised one, but we are working on a distributed implementation too.

Instead, two different types of tuple storage have been implemented: a transient and a persistent one. The first type uses spatial indexes to speed-up space operations. Since transient spaces lose their content on system crashes or service shutdowns, a persistent tuple space based on a XML database has been implemented. Write operations store the XML description of the data received from the clients into the databases, take operations are accomplished using XPath queries on these data. As database implementation we have used the Oracle XML Berkeley DB because according to [13] it seems to have the best performance. An implementation based on Xindice [14] is planned for a future release.

B. Serialization of Tuples

One of the most important problems faced during the development of Grinda service was the handling of different data types in tuples. In fact, web service developers usually have to deal with predefined data structures that are serialized (or deserialized) using the stubs created automatically by tools on the basis of WSDL definitions. Even when an element with the XML Schema type any is part of input or output of a web service operation, serializers can be called at client or server side to obtain the corresponding object in order to manipulate it with the used programming language. This can be seen as a "tyranny of serializers" that forces all data types transmitted to have a corresponding serializer/deserializer pair at both client and service side.

This approach is used because hard-coding the serialization is more efficient than other methods like reflection, but for our service we need something more flexible for the following reasons:

• Developers should be able to use their own custom data types in a simple way without the need to create stubs or coding serializers. This avoid the proliferation of data types whose unique purpose is to encapsulate legacy data: why don’t transmit them directly? The program complexity and the development efforts would be reduced.

• Third part developers have no control on the service and thus they cannot deploy new serializers for data types not yet supported. This is a great problem especially for the

class Person {
    String firstName="John";<firstName>John</firstName>
    String lastName="Doe";<lastName>Doe</lastName>
    Date dateOfBirth= new Date("01/01/1970");<dateOfBirth>01/01/1970</dateOfBirth>
    int age="37";<age>37</age>
}

| TABLE I: Example of XStream serialization |

present tuple space implementation that is unable to know a priori what types will be transmitted to the service.

To support these features, a tuple model that can be simply converted to XML and vice versa is needed. JavaSpaces seems the most simple model to satisfy this goal simply, because it allows a direct translation as described in the following.

However, to do such an operation it is required a library that allows the direct translation of data into XML. We have used the XStream library [15]. It takes in input a Java object and serialize it in XML using its own internal structure without the definition of any kind of specific serializer. The library uses the Java reflection API to obtain the object fields and translates them into XML elements: field names become tag names and their values are recursively converted to textual form. The root element name is equal to the class name. In this way theoretically only a serializer would be enough to marshal every possible type and we do not need to create new serializers for every type in use. If some type still needs specific serializers the XStream library provides a simple plugin system and specific serializers for many standard Java types. An example of XStream serialization is reported in Table 1.

Like any other SOAP based communication, our approach is not well suited for transmitting large amount of binary data, that should use more suitable protocols like GridFTP. Nevertheless sometimes it is necessary to transmit small arrays or drive complex data movement.

For this reason the Grinda client is able to manage the transmission of arbitrary data outside the tuple space, using the most suitable protocol. At this stage of development, array serializers using plain XML, base64 encoding and SOAP Attachment [16] have been implemented. Moreover, files can be directly used as tuples: only a link to them will be transmitted through the space. Receivers will download them with the specified protocol reducing the server load.

To manage subclass matching or out-of-space transmissions, some attributes are added to the corresponding XML serialization to allow their correct deserialization.

Using this approach it is possible to transmit every type with the minimum effort for the developer. Apart efficiency in avoiding the proliferation of "wrapper types", this approach enforces a better modularisation. In fact, since every type can be serialized, it is possible to use directly part of the business logic of the application allowing methods to operate directly on serialized data. So, methods can be directly invoked on
received objects and other support classes are no more needed. In this way a result similar to Java Serialization is obtained: a data is transmitted and immediately one of its method can be called. This approach does not support code mobility because the class definition is not transmitted and should be present on every tuple space client.

C. C++ Client

The development of the C++ client has been quite challenging to maintain the possibility of an automatic type serialization. Unfortunately, C++ doesn't implement a standard reflection API like Java or C# and RTTI is not powerful enough to support the automatic serialization of custom types. There are several projects that try to support reflection mechanisms in C++ like [17], [18] or [19] but the most of them are only prototypes and not widely adopted. For this reason we have used the Qt4 Toolkit [20], one of the most important toolkits for developing portable applications used in many projects like for example KDE. In fact, it provides an interesting feature: the Meta Object System. This allows to have type information about classes developed using the toolkit and to instantiate them by name at runtime. Although less evolved than the Java and C# counterparts, it is still very useful for our purpose.

In the present Java implementation the reflection API has been used to collect the class fields and transform them in XML elements. The same strategy has been applied in the C++ client development with the use of QProperties. These properties are managed by the Qt runtime and can be written or read by name. So custom objects can be serialized automatically reading their properties and transforming in XML elements with the same name. To better accomplish this process, properties should be defined using the QVariant class that allows primitive types like int or float to be handled as objects. The only inconvenience of this approach is that custom objects and their properties should be manually defined by the developers using some macros.

The used Qt framework release does not implement an API for SOAP or XML-RPC messaging and so we have used the gSOAP 2.7.9 library [21], a little and embeddable library that allows to develop web services in both C and C++. It can generate client or server-side stubs and provides a simple HTTP server that has been used to implement asynchronous operations. Although the Globus Toolkit also offers a C API we have used the gSOAP library because it is more simple, lightweight and has a native support for C++.

V. Test Results

A series of tests have been performed in order to verify the behavior of the Grinda service. We have measured two different parameters: the latency of the system and its scalability, when a distributed application is deployed on a network. Since scalability can be influenced by the application type, we have tested two applications with different behavior and amount of data exchanged. Moreover, we have performed a test on a simple workflow to verify the versatility of our system.

The first type of application simulates a brute force attack to guess an hashed password. This is an highly parallel application that requires almost no communications during its execution. Thus the communication overhead should be very limited.

The second test application is a plasma simulation performed porting in Grinda the MPI-based ALaDyn code [22], [23], [24] that implements a so called Particle In Cell (PIC) based simulation [25], [26]. The algorithm used in this code is known to be not completely parallel and requires much more communications than the first application. Moreover, since this test application is derived from a MPI-based code it makes possible to measure the real performance of Grinda.

All our tests have been performed on the same testbed: a 100Mbps Ethernet LAN composed by Core Duo PCs equipped with Ubuntu Linux 7.04. This network was not dedicated because it is a student laboratory, but was the only choice we had to collect a medium-large number of hosts.

A. Latency Tests

The first test has measured the latency of Grinda service and its dependency on the host number. As latency we mean the average time required to accomplish tuple space operations with the minimum amount of data. Given a number of clients, the tests consist in calculating the average time spent for taking/writing the same tuple from/in a space. The average time values have been obtained from 1000 repeated tests. As shown in figure 2, the latency is not heavily influenced by the number of hosts. The average time required by take operations seems to be absolutely independent of the network size whereas write operations increase by a small factor, ~20%, when the size grows.

Since the two curves have a distance much larger than standard deviation, it seems that write operations require more time than take ones. Probably this is caused by an higher overhead in instantiating all the objects required for storing tuples.

B. Scalability Test

As described before, the scalability test has been performed on two different types of applications. In both cases we have measured the average time required for computation as host number increase.
The first application tested was the highly parallel one. It simulates a brute force attack on an hashed password using a master/slave strategy. At the beginning the master randomly generates a password and computes its SHA1 hash. Then it writes all tasks to the tuple space. Each of them contains the hashed password along with the interval of the strings to check. The slaves continuously loop through the following steps:

1) take a task from the tuple space
2) generate the requested strings
3) calculate their SHA1 hash
4) compare them with the one sent by the master
5) write a partial result to the space

At the end the master collects all the partial results and returns the password.

It is clear that this simple application is completely parallel and uses the implicit load balancing of the tuple space model. In fact, since all the tasks are written to the space at the beginning, the quicker slaves can collect and execute an higher number of tasks respect to the slower ones, thus increasing the overall utilisation of the distributed system. Moreover, the slaves should not wait until the master finish to write all the tasks, they start immediately when a new task become available. Finally, the task size is very small thus reducing the communication overhead. The figure 3 reports the speedup of the application with respect to node number. It is clear that this application scale very well, being near to the linear speedup.

The second application we have used to test our framework is a plasma simulation. Our test application is based on the ALaDyn code [22], [23], [24] that implements a so called Particle In Cell (PIC) based simulation [25], [26]. It uses a computational grid for calculating the electromagnetic field. The standard PIC method proceeds iterating the following steps:

- compute the particle parameters on the grid from the particle position
- solve the field equations on the grid to obtain the force field on the mesh
- compute the particle parameters at the particle position by back interpolation from grid point values
- advance the particles in time integrating the motion equation for velocities and positions

The simplest way to parallelize this algorithm is to split the grid along one or more axis and assign each of the resulting slices to one host. At every step they communicate each other the particles that have migrated from a slice to another.

The ALaDyn code uses this method to parallelize the simulation. Since it was written in C using MPI, we have ported it to Grinda substituting only the MPI calls, enabling the communications through the C++ client. The rest of the algorithm remained untouched. For the tests a 250x250x800 computational grid with $10^7$ particles has been used on the same test network of the first application.

At this point we were able to compare our framework with MPI using the plasma simulation application as a simple benchmark. In fact we have deployed the original application on the same network along with the open source MPICH 1.2.7 library. As showed by figure 4 the original MPI-based application and our code have almost equivalent performance. The difference seems to have essentially a statistical meaning.

It should be outlined that this type of application is not the ideal application type to which Grinda is focused. In fact it requires many binary data exchanges that could not occur across the tuple space for performance reason (in the test we have used a shared file system). This test has been developed only to show the feasibility and validity of our service in difficult circumstances.

C. Sample Application

To show the versatility of our system, we have also implemented and tested a simple workflow. The tuple space is used as the basis for a workflow engine, which uses the subclass matching. All tasks composing the workflow are implemented using the same interface and are written in the space. A variable number of slaves take the first available of them. Task inputs are stored inside the task class itself and, after the execution, tasks can produce other tasks or other type of data as output. In every case they will be written into the space.

To increase the application performance, it is sufficient to add more slaves allowing more tasks to be executed at the same time.

The workflow we have used is depicted in Figure 5a. It is composed by more than 250 tasks: the most of them can be executed in parallel apart the central one that is unique and
have to wait for the previous task to be executed. To test our
workflow, we have executed it on the usual test network with
an increasing number of hosts. Each of them is responsible for
executing a variable number of tasks according to its load.
Thus this number is not defined in advance and depends on
the load of the single machines. For each network size, the
average time spent for different runs of the test application
has been calculated. The experimental results are shown in
Figure 5b.

As shown by the experimental results, the application
scales quite well at the beginning, reducing its performance
for network with more than 16 hosts. This is an expected
results because the workflow we have used is not completely
parallelizable.

Finally, we have to state that the workflow support we
have implemented is only a prototype. Actually, it lacks of
many features like a module able to automatically transform
a graphical or textual representation of the workflow in a
sequence of tuple space operations.

VI. CONCLUSION

In this article we have described our experiences in de-
veloping a tuple space service for the Globus Toolkit. This
service can be used to implement distributed applications in
a simple way, and exploits interesting feature like automatic
load-balancing.

To develop such a service we have used an uncommon
model for implementing the tuple space due to the limits
imposed by the XML serialization. However this did not
reduce the expressive power of the model and it is also suitable
for large data transmissions. The produced architecture is
modular enough to support legacy applications too.

According to the test results we have collected, it seems that
our service can provide good performance to both completely
and non-completely parallelizable applications. It has to be
noted that these tests are only preliminary; more deeper tests
and further development of this service are needed, in order
to produce an industrial-strength product.

ACKNOWLEDGMENTS

This work has been partially supported by the strategic
project ESOV of the University of Bologna.

REFERENCES

many faces of publish/subscribe,” ACM Comput. Surv., vol. 35, no. 2,
ACM Transactions on Programming Languages and Systems, vol. 7,
Systems,” in IFIP International Conference on Network and Parallel
'09: Proceedings of the Thirty-Second Annual Hawaii International
Mobility,” in International Conference on Software Engineering. Los
Internet agents,” in SAC '99: Proceedings of the 1999 ACM symposium on
Source Native XML databases - A Case Study,” in The 8th International
Conference on Advanced Communication Technology (ICACT 2006).
Energy and Nuclear Physics (CHEP '04), Interlaken, Switzerland, 2004.
investigation of Maxwell-Vlasov equations. Part I: basic physics and
algorithms,” Comm. in Nonlinear Science and Numerical Simulations,
high accuracy PIC code for the Maxwell-Vlasov equations,” submitted
to IEEE-Transactions on Plasma Science.