Enabling Transparent Data Sharing in Component Models

Gabriel Antoniu, Hinde Lilia Bouziane, Landry Breuil, Mathieu Jan, Christian Pérez

Enabling Transparent Data Sharing in Component Models

Gabriel Antoniu, Hinde Lilia Bouziane, Landry Breuil, Mathieu Jan, Christian Pérez
IRISA/INRIA, Campus de Beaulieu, 35042 Rennes cedex, France

Abstract

The fast growth of high-bandwidth wide-area networks has encouraged the development of computational grids. To deal with the increasing complexity of grid applications, the software component technology seems very appealing since it emphasizes software composition and re-use. However, current software component models only support explicit *data transfers between components. The distributed shared memory paradigm has demonstrated its utility by enabling a* transparent *access to data via a globally shared data space. This paper proposes to extend software component models with shared memory capabilities, enabling transparent access to shared data across components and leading to further decreased software complexity.*

1 Introduction

Programming distributed systems has always been seen as a tedious activity for a programmer. Grid infrastructures, as the latest incarnation of distributed systems, are not exception to this reality. In addition to the difficulty of coding inherent to the application, the programmer often has to deal with low-level programming and runtime issues such as communications between different modules of the application or deployment of modules among a set of available resources.

Several approaches to distributed systems programming have been pursued such as *Remote Procedure Calls* or *Distributed Objects*. They allow usual programming paradigms (function calls or objects) to be applied by transparently invoking a function of a remote program or a method of a remote object, as if they were local. *Distributed Shared Memory* is another approach that has been proposed in order to hide the aspects related to data distribution.

Recently, software component models have emerged and appear as a very promising approach for programming the grid. The component approach [9] enforces *composition* as the main paradigm for developing distributed applications. This offers the advantages of decreasing the design complexity and of improving productivity by facilitating software re-use.

Interactions between components are done through well-defined ports. However, in existing component models, only ports enable explicit data transfer as part of an exchanged message. Consequently, it is not currently possible to easily share data between components. Moreover, as several components may want to modify the same data, the functional code of a component should deal with data persistence, data consistency and fault tolerance issues. This therefore leads to an increased application complexity.

This paper proposes to enrich current *software component models* with a *transparent data access model*, solving aforementioned problems as the complexity of an application is therefore lowered. More precisely, our proposal aims at providing a transparent access to data shared across components. This is achieved by extending component models with a new kind of ports for dealing with shared data access.

Section 2 introduces *transparent data access* and JuxMem in particular. Then, Section 3 presents our general proposal to enable transparent data sharing in component models and its projection to Ccm using JuxMem. Section 4 concludes the paper

2 A transparent data access model

2.1 Benefits of transparent access to data

The most widely-used approach to manage data on distributed environments (and on grid platforms in particular) relies on the *explicit data access model*, where clients have to move data to computing servers. A typical example is the use of the GridFTP protocol [1]. In order to add some degree of persistence for data that may be useful to multiple computations, higher-level layers may build data catalogs [1] on top of GridFTP.

Catalogs may allow multiple copies of the same data to be registered, however the user has to manually handle the localization of the replicas and their consistency. Such a low-level approach makes data management on grids rather complex.

In order to overcome these limitations and make a step towards a real virtualization of the management of large-scale distributed data, the concept of *grid data-sharing service* has been proposed [2]. The idea is to provide a *transparent data access model*: in this approach, the user accesses data via global identifiers. The service which implements this model handles data localization and transfer without any help from the programmer. It transparently manages data persistence in a dynamic, large-scale, distributed environment. The data sharing service concept is based on a hybrid approach inspired by Distributed Shared Memory (DSM) systems (for transparent access to data and consistency management) and peer-to-peer (P2P) systems (for their scalability and volatility tolerance). In addition to data persistence, the service specification includes the following properties: data consistency and fault tolerance despite failures [4].

2.2 Overview of JuxMem

The concept of data-sharing service is illustrated by the JuxMem [2] software experimental platform. The general architecture of JuxMem mirrors a federation of distributed clusters and is therefore *hierarchical*. The goal is to accurately map the physical network topology, in order to efficiently use the underlying high performance networks available on grid infrastructures. Consequently, the architecture of JuxMem relies on node sets to express the hierarchical nature of the targeted testbed. Any node may use the service to allocate, read or write data as *clients*, in a peer-to-peer approach. This architecture has been implemented using the JXTA [10] generic P2P platform.

The JuxMem API provides to users classical functions to allocate and map/unmap memory blocks, such as `juxmem_malloc`, etc. The memory allocation operation returns a global data ID. This ID can be used by other nodes in order to access existing data through the use of the `juxmem_mmap` function. To obtain read and/or write access on a data, a process that uses JuxMem should acquire the lock associated to the data through either `juxmem_acquire` (exclusive lock) or `juxmem_acquire_read` (read-only lock).

3 Extending component models with data access ports

Even though it is possible to simulate data access by explicit message exchanges through classical ports, a more efficient and easier solution for data management in component models seems possible by relying on an external data sharing service, such as JuxMem. Hence management of concurrent accesses to data and synchronizations is removed from the functional code of the component and appears transparent to the component implementer. To this purpose, we introduce a *data port model* as an attempt to enrich existing *component models* with *transparent data access*. Then, this generic abstract model is instanced to the Corba Component Model (Ccm). Let us note however that this approach could also be applied to other component models, such as Cca [5] or Fractal [6].

3.1 An abstract data port model

Our proposal consists in introducing a dedicated family of ports named *data ports*, able to logically attach a shared data to a component. Such a semantics could be obtained by relying on the *transparent data access model* described in Section 2. To this end, we define two kinds of ports. On one hand, a *shares* port gives an access to a shared data. On the other hand, an *accesses* port enables a component to access a data exported through a *shares* port.

In order to access a data, two types of interfaces are required: one for accessing the data, named *data_handling*, and one for making the data available to other components, named *data_reference_export*. The *data_handling* interface is an internal component interface available through *accesses* ports as well as through *shares* ports. Indeed, a component that *shares* some data may also need to access the data. Figure 1 shows the API of the *data_handling* interface offered to the programmer. It provides `get_pointer`/`get_size` primitives to respectively retrieve a pointer to the shared data and its size. It also provides synchronization primitives, like `acquire` and `release`. The `acquire_read` primitive sets a lock in read-only mode so that multiple readers can simultaneously access the data, whereas `acquire` sets a lock in exclusive mode.

The *data_reference_export* interface is an external component interface only provided by *shares* ports. It aims at allowing a component with an *accesses* port to retrieve a reference to some data provided by a *shares* port. Typically, it contains an operation which returns the global data ID (see Section 2.2).

```
interface data_handling {
   float* get_pointer();
   long get_size();
   // Synchronization primitives
   void  acquire();
   void  acquire_read();
   void  release();
};
```

Figure 1. An interface offered to the programmer by data ports. The shared data is an array of float.

```
typedef position float[N][3];
// Component sharing a data space.
component sharer {
   shares position to_bodies;
// Component simulating a body.
component body {
   accesses position from_sharer;
};
```

Figure 2. An OMG IDL3+ example of data ports.

3.2 Case study: extending CCM via JUXMEM-based data ports

This section describes a projection of the previously introduced abstract data port model (*shares* and *accesses* ports) onto the CCM component model and JUXMEM. Let us first give a brief overview of CCM.

CCM [8] is part of the CORBA [7] (*Common Object Request Broker Architecture*) specifications (version 3). The CCM specifications allow the deployment of components into a distributed and heterogeneous environment. A CORBA component, can define five kinds of ports. *Facets* and *receptacles* provide the remote method invocation paradigm. An event communication model is offered by event *sources* and *sinks*. *Attributes*, which are primarily intended to be used for component configuration, are values exposed through accessor (read) and mutator (write) operations.

CCM offers a design model to describe components and their ports using the CORBA 3 version of the OMG Interface Definition Language (IDL). It also provides an assembly model, a packaging and deployment model, an execution model and a component's life cycle management model. Each model deals with a part of a CCM application life cycle.

We propose to extend CCM with *data ports* capabilities; this requires to enhance its IDL3 language. Our proposal therefore consists in introducing two new keywords in this language: *shares* and *accesses*. Such an extended IDL is named *IDL3+*.

Figure 2 shows an example of an IDL3+ specification for a N-body simulation. At each step of the simulation, the position of bodies in the space are computed according to the gravitational forces applied on them by other bodies. As this example is for an illustration purpose only, we consider a very naive solution to this problem. A body is represented by a component `body` which accesses the global array of positions of all bodies involved in the simulation. Note that the number of `body` components is set when assembling the application. Figure 3 shows how, in the functional code of a `body` component, the data is accessed at each step of the simulation. First, a read-only lock is acquired on the global array to retrieve the position of other bodies in the space, so that they can be used later in the local computation of the new position for the current simulated body. Then, an exclusive lock is set on the position of the current body before updating the value in the global array. Therefore, the data is handled as if it was locally present in the component without dealing with explicit data transfer. The used synchronization primitives are mapped on the synchronization API of JUXMEM. Note that, for the sake of clarity, synchronization mechanisms between simulation steps are not shown.

```
class Body_impl : virtual public CCM_body {
  private:
   data_access* positions_data_port;
   int id;
};
void Body_impl::updatePosition() {
 float* data_ptr[];
 data_ptr = position_data_port.get_pointer();
 positions_data_port.acquire_read();
 for (i = 0; i < N; i ++) {
   addToLocalComputation(data_ptr[i]);
 }
 positions_data_port.release();
 position_data_port.acquire(data_ptr[id]);
 updateOurPosition(data_ptr[id]);
 position_data_port.release();
}
```

Figure 3. C++ implementation of a N-body application with data ports.

3.3 A portable implementation for CCM

We have implemented the data port model into CCM using MicoCCM (version 2.3.11) and JUXMEM (version 0.2) as a data sharing service. Our solution is independent from any CCM implementation and it is compliant with the CORBA specifications for portability reasons. Whereas this strategy may not lead to the most efficient implementation, it allows us to validate

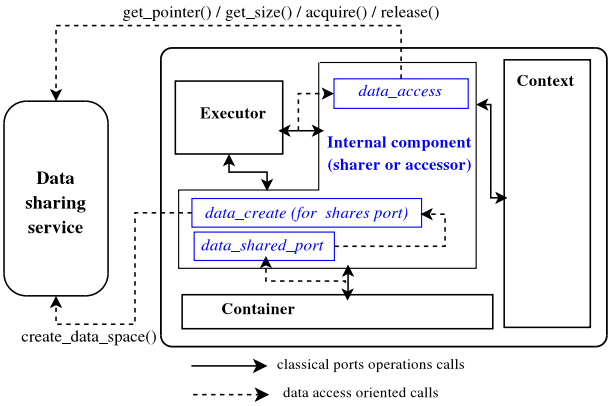

Figure 4. The internal architecture proposal

our proposal without the complex burden of modifying an existing CCM framework.

With respect to our constraints, the IDL3+ specification is projected to a classical IDL3 one. For the N-body example, the generated IDL3 contains two internal components: `Internal_sharer` and `Internal_body`. They both support operations needed by respectively the *shares* and *accesses* ports and logically belong to the (`sharer` and `body`) components. Internal components are making up an *intermediate* context and container between the executor and the other parts of a classical CCM component, as shown on Figure 4. They transparently redirect CORBA calls to the classical context and container, and intercept the data port oriented calls. Moreover, this permits to easily switch to different data access model implementations (JUXMEM, *locally-shared file* or *NFS-shared file*).

Note that an internal interface, named *data_ create*, is also attached to a *shares* port. It contains only a `create_data_space` operation, which provides a way to associate a data to a data port. This operation is invoked by the framework when an *accesses* port is connected to a *shares* port. A data shared through a *shares* data port can hence be created on the fly. A more detailed presentation is given in [3].

4 Conclusion

Software components appear to be very promising to build complex applications. However, in existing software component models data need to be explicitly exchanged. This paper has enhanced data management in component models. The proposal provides a *transparent access* to shared data within components, using a data sharing service. This is achieved by defining a new class of ports: the *data ports*, which allow to make data available and accessible to multiple components.

As a proof of concept, we have implemented the runtime part of the data port as an extension to the CORBA Component Model and used the JUXMEM data sharing service, which provides the required *transparent data access model*. The prototype has been successfully tested through a synthetic application.

We plan to develop an operational IDL3+ compiler. We are also considering the use of shared data as parameters of operations provided by a component. Furthermore, we are currently adapting a code coupling application, to compare the data port approach to standard data management approaches, in terms of programming difficulty but also in terms of performance.

References

[1] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, 2002.

[2] G. Antoniu, L. Bougé, and M. Jan. JuxMem: An adaptive supportive platform for data sharing on the grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, Nov. 2005.

[3] G. Antoniu, H. L. Bouziane, L. Breuil, M. Jan, and C. Pérez. Enabling transparent data sharing in component models. Research Report RR-5796, INRIA, IRISA, Rennes, France, November 2006.

[4] G. Antoniu, J.-F. Deverge, and S. Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing. *Concurrency and Computation: Practice and Experience*, (17):1–19, Sept. 2005.

[5] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *8th IEEE International Symposium on High Performance Distributed Computation*, page 13, Redondo Beach, California, Aug. 1999.

[6] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOP02)*, Malaga, Spain, jun 2002.

[7] OMG. The Common Object Request Broker: Architecture and Specification V3.0. Technical Report OMG Document formal/02-06-33, June 2002.

[8] Open Management Group (OMG). CORBA components, version 3. Document formal/02-06-65, June 2002.

[9] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998.

[10] The JXTA project. `http://www.jxta.org/`, 2001.