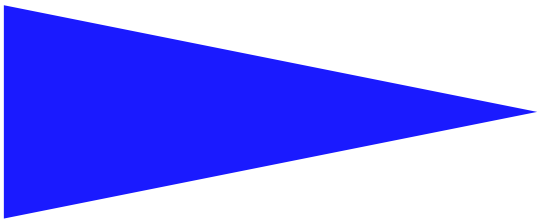
I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1774



ENABLING TRANSPARENT DATA SHARING IN COMPONENT MODELS

GABRIEL ANTONIU, HINDE LILIA BOUZIANE, LANDRY BREUIL,
MATHIEU JAN, CHRISTIAN PÉREZ

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Enabling Transparent Data Sharing in Component Models

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

Systèmes numériques
Projet Paris

Publication interne n1774 — January 2006 — 22 pages

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 through remote procedure call (RPC), remote method invocation (RMI) or event based ports. The distributed shared memory paradigm has demonstrated its utility by enabling a *transparent* access to data via a globally shared data space. Data localization, replication and transfer as well as synchronization of concurrent accesses are delegated to an external data-sharing service. This paper proposes to extend software component models with shared memory capabilities: a) a transparent access to shared data, and b) the possibility to use shared data as parameters of operations provided by component ports. The proposed model is instantiated as an extension of the CORBA Component Model (CCM) and the Common Component Architecture (CCA), using the transparent data access model provided by the JUXMEM grid data-sharing service.

Key-words: data sharing, component, grid, JUXMEM

(Résumé : tsvp)

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Extension aux modèles de composants: partage transparent de données

Résumé : La rapide mise en place de réseaux longue distance à haut-débit a favorisé le développement des grilles de calculs. De telles infrastructures sont nécessaires pour répondre au besoin croissant en terme de puissance de calcul des applications de simulations numériques. Toutefois, la conception de tels logiciels est de plus en plus complexe. Pour résoudre ce problème, la technologie des composants logiciels est très prometteuse puisqu'elle permet la composition et la ré-utilisation d'entitées logicielles. Toutefois, les modèles actuels de composants logiciels permettent seulement de faire des transferts *explicites* de données entre les composants, par l'intermédiaire d'appels de procédures distantes (RPC), d'invocation de méthodes distantes (RMI) ou des ports d'évènements. Le paradigme des mémoires virtuellement partagées a démontré son intérêt en fournissant l'illusion d'un espace d'adressage global qui permet un accès *transparent* aux données. La localisation des données, leur réplication et leur transfert, ainsi que la synchronisation des accès concurrents sont délégués à un service de gestion de données externe. Ce papier propose d'étendre les modèles de composants logiciels avec les capacités d'une mémoire partagée afin de fournir : a) un accès transparent aux données partagées, et b) la possibilité d'utiliser des données partagées comme paramètres des opérations fournies par les ports d'un composant. Le modèle proposé est instancié comme une extension du modèle composant CORBA (CCM) et du modèle CCA (Common Component Architecture), et utilise le modèle d'accès transparent aux données fournit par le service de gestion de données pour grille JUXMEM.

Mots clés : partage de données, composant logiciel, grille, JUXMEM

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 coding of the application logic, a 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 program distributed systems have been pursued such as *Remote Procedure Call* or *Distributed Objects*. They allowed usual programming paradigms (function call 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 the distribution of data used in a computation. While distributed shared memory systems have mainly been restricted to parallel machines, they appear as a convenient programming model, since applications do not have to worry about data localization. For example, data transfers become meaningless, as data are accessible from anywhere.

Recently, software component models have emerged and appear as a very promising approach for programming the grid. Instead of following an object-oriented approach, and its associated inheritance mechanism, a component approach 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.

However, one limitation of current software component models is their lack of support for data access. Existing component models assume an active communication operation between two components: a message triggers some reaction from the receiving component. As such, they are only able to deal with data as a part of a message actually exchanged between two components. 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 increase in the complexity of an application.

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 as well as a transparent sharing of operations parameters provided by components. This is achieved by extending component models with a new kind of port for dealing with shared data access.

Section 2 introduces software component concepts and presents two component models: the CORBA Component Model (CCM) and the Common Component Architecture (CCA).

Section 3 discusses current models for data access and focuses on the *transparent data access model*, as provided by the JUXMEM grid data sharing service. In Section 4, current limitations of component models for managing data are stated. Then, Section 5 introduces our proposal to enable transparent data sharing in component models and its associated semantics. It also presents an incarnation of this proposal through an example by extending CCM and using JUXMEM as a data sharing service. In Section 6, the handling of shared operation parameters is described. The abstract model is instancied over CCA. Finally, Section 7 concludes the paper.

2 Software component models

2.1 Overview of software component concepts

The software component technology [18] has been emerging for a few years [10], even though its underlying intuition is not very recent [14]. A largely accepted definition for a software component has been proposed by Szyperski [18]: *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."* Let us analyze some implications of this definition.

2.1.1 Composition

the fundamental property of a component is its ability to be composed with other components. This interaction is done through well-defined interfaces, to which an interaction contract is attached. Components need to agree on such a contract, which explicitly specifies constraints related for instance to quality of service, security, transactional semantics, etc. In particular, interfaces can be strongly typed so that checks such as type conformance could be performed when connecting two interfaces. Hence, building an application based on components emphasizes application design by *assembly*, rather than by *programming*. The goals are to focus expertise on domain fields, to improve software quality and to decrease the time to market thanks to reuse of existing codes.

2.1.2 Ports

to be able to interact with other components, a component defines external visible interfaces named ports. A port is a programming artifact to which an interface can be attached. It can be categorized in two types: *client* or *server* port. The interaction between two components is then performed by connecting a *client* port of a component to a *server* one of another

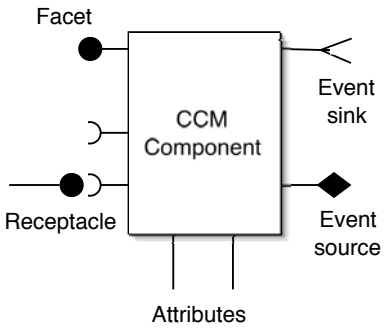


Figure 1: External view of a CCM component.

component with compatible type. A port provides also two views of the associated interface. The first one is the *external view*, exposed to other components. The second one is the *internal view*, to be used in the implementation of the component. The internal view corresponds, for instance, to the interface provided to the component implementer by a *client* port. It also corresponds to an interface that has to be implemented, e.g. to support a *server* port.

2.1.3 Packaging and deployment

a component is a binary unit of deployment. It should reference at least an implementation (binary code) and the constraints associated to it, like operating systems, or amount of memory requirements. These properties help a deployment tool to decide the resources an instance of the component may be installed on.

2.2 The CORBA Component Model

The CORBA Component Model [17] is part of the latest CORBA [16] (*Common Object Request Broker Architecture*) specifications (version 3), an industrial standard. The CCM specifications allow the deployment of components into a distributed and heterogeneous environment.

A CORBA component, as represented in Figure 1, can define five kinds of ports. Facets (*"provides"* ports) and receptacles (*"uses"* ports) allow a synchronous communication model based on the remote method invocation paradigm to be expressed. An asynchronous communication model based on the transfer on some data is implemented by event sources and sinks. Attributes are values exposed through accessor (read) and mutator (write) operations.

```
// Interface Average definition
typedef sequence<double> Vector;
interface Average {
  double compute(in Vector v);
};
// Component A definition
component aComponent {
  attribute string  name;
  provides  Average avgPort;
  uses      Display dspPort;
};
```

Figure 2: A component IDL definition.

Attributes are primarily intended to be used for component configuration, although they may be used in a variety of other ways.

CCM offers a complete model to develop a component based application: (1) a design model to describe components and their ports using the CORBA 3 version of the OMG Interface Definition Language (IDL), as shown on Figure 2; (2) an assembly model to describe an application architecture thanks to an Architecture Description Language; (3) a packaging and deployment model to deploy an application from an assembly description; (4) an execution model to offer a set of standard services to a component, such as security, events, and persistence; and (5) a component's life cycle management model to create, find or remove component instances through the use of entities named *homes*. Point (4) enables the same component to be hosted by different framework implementations.

2.3 The Common Component Architecture

The CCA component model [9] is a set of standards defined by a group of researchers from US national laboratories and academic institutions. The goal of the group is to develop a common architecture for building large scale scientific applications based on well tested software components.

A CCA component can define *"uses"* or *"provides"* ports. These ports are analogous to receptacles and facets of CCM. The specification of such ports in CCA is done by using the Scientific IDL. Ports are dynamically added or removed to components.

CCA defines a repository, configuration and framework services APIs for introspection ability, assuring collaboration between components with different frameworks and framework services access like security, communication and memory management.

Unlike CCM, the assembly model of CCA is only dynamic. This means that there is not any Architecture Description Language (ADL) to describe components or component compositions. CCA relies entirely on run-time calls.

3 A transparent data access model

Current component models lack to provide any transparent data access model. This section firts presents both explicit and transparent data access models, and then, it presents the advantages of the transparent data access model.

3.1 The benefits of transparent access to data

Currently, 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 [6]. Though this protocol provides authentication, parallel transfers, checkpoint/restart mechanisms, etc., it is still a transfer protocol which requires *explicit* data localization. In order to add some degree of persistence for data that may be useful to multiple computations, higher-level layers may build data catalogs [6] on top of GridFTP. This approach is used in today's data grids in order to build data collections [1, 4], systems for sharing experimental data among multiple laboratories [2], public digital libraries (e.g. for astronomical data [3]), persistent data archive for administrative documents [15], etc. 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 [7]. 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).

The service specification includes the following properties:

3.1.1 Persistence

different application executions may need to share the same data. The data sharing service provides persistent data storage and relies on strategies able to reuse previously produced data, by avoiding repeated data transfers between clients and servers.

3.1.2 Data consistency

in the general case, shared data manipulated by grid applications may be *mutable*. Data can be read, but also *updated* by the different codes. When accessed on multiple sites, data are often replicated to enhance access locality. To ensure the consistency of the different replicas, the service relies on *consistency models*, implemented by *consistency protocols*.

3.1.3 Fault tolerance

storage resources can join and leave, or unexpectedly fail. In order to support these events, the data sharing service relies on replication techniques and failure detection mechanisms in order to enhance data availability despite disconnections and failures [8]. These aspects are beyond the scope of this paper.

3.2 Overview of JUXMEM

The concept of data-sharing service is illustrated by the JUXMEM [7] 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. They are called *cluster groups* and correspond to physical clusters. These groups are included in a wider group, the *juxmem group*, which gathers all the nodes running the data-sharing service. Any cluster group consists of *provider* nodes which supply memory for data storage. Any node (including providers) 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 [21] generic P2P platform.

The JUXMEM API provides to users classical functions to allocate and map/unmap memory blocks, such as `juxmem_malloc`, `juxmem_mmap`, etc. However, when allocating a memory block, the client has to specify: 1) on how many clusters the data should be replicated; 2) on how many providers in each cluster the data should be replicated; 3) the consistency protocol that should be used to manage this data. The allocation operation returns a global data ID. This ID can be used by other nodes in order to identify existing data.

It is JUXMEM's responsibility to localize the data and perform the necessary data transfers based on this ID. This is how JUXMEM provides a transparent access to data. 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` or `juxmem_acquire_read`. This permits to apply consistency guarantees according to the consistency protocol specified by the user at the allocation time of the data. Note that `juxmem_acquire_read` allows multiple readers to simultaneously access the same data.

4 Current component model limitations with respect to data access

As previously explained, the port definition of current component models is based on the assumption of an active communication operation between two components. As such, they are only able to deal with data as a part of a message actually exchanged between two components.

Using classical ports such as *provides* and *uses* ports of CCM or CCA, is possible to (inefficiently) simulate a shared data access. In this case, the data is physically located into a component that provides it. However, with such a centralized approach, the component storing the data can easily produce a bottleneck as the number of concurrent accesses increases. Another possibility is to have a copy of the shared data on each component that uses it. Therefore, the functional code of a component should maintain a consistent state of all copies, each time the data is updated. For example, this can be achieved through the use of a consensus algorithm. Consequently, the management of synchronizations and concurrent accesses to data is handled within the functional code of the component, leading to an unnecessary increase in the complexity of an application.

Hence, existing software component models are not able to efficiently support a *transparent data access* model, as described in Section 3. We claim that this is a limitation for current component models as data persistence, consistency and fault-tolerance are not handled. Section 2 has briefly introduced two component models: CCM and CCA for illustration purpose. However, let us note that other component models, like Fractal [11], Grid.it [5], Iceni [12], or Darwin [13] provide the same kind of ports and therefore have the same limitation.

A transparent access to data within a component does not entirely solve data management issues in component models. Operations provided by a component may indeed also use shared data as parameters. In current component models, operation parameters of provide port interfaces are indeed (logically) transfered between the caller and the callee. A caller is a component with a uses port, whereas a callee is a component with a provides

port. How the data transfer between the two components occurs is specified by the *parameter mode* associated to each parameter. However, the caller usually looses the right to access the data during the time of the invocation time. For instance, with an `inout` parameter mode the callee is allowed to destroy and re-allocate the data. Therefore, it is not possible for the caller to access or share a data with other components, while it is used by the callee. With an `in` parameter mode and depending of which component model is used, the data can be a constant for the callee (CCM) or passed by value (CCA). Hence, a parameter can not be shared either between the caller and the callee or across several concurrent operations on the same or possibly different components. In our opinion, this inability arises as another limitation to current component models. Note that data transfers for several consecutive operation invocations can be inefficient as data may be unnecessary sent back and forth between components without a shared data model.

These two issues with current component models for data management are respectively addressed in Sections 5 and 6.

5 Extending component models with data access ports

As seen in the previous section, it is possible to simulate accesses to external data, but at the cost of an increased complexity. However, a more efficient and easier solution seems possible. It consists in using a data sharing service such as JUXMEM to manage the data. Hence, the illusion that the data is located into a component even if it is physically away can be provided. Moreover, the management of concurrent accesses and synchronizations is removed from the functional code of the component. This management can also appear mainly as transparent to the component framework as it can be handled by a data sharing service. This section proposes a *data port model* as an attempt to enrich the existing *component models* with the features provided by a *transparent data access model*. First, we introduce an *abstract* data port model. Let us stress that this model is not specific to any component model. Then, a projection of this model to the concrete model of CCM is dealt within Section 5.2.

5.1 An abstract data port model

Software component models define several types of ports to enable interactions between components (cf Section 2). However and as far as we know, all available types of ports are logically based on messages. The component framework is responsible for transporting a message between two components. In our proposal, we introduce a specialized family of ports named *data ports*, able to logically attach a shared data to a component. Such a

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

Figure 3: An access interface offered to the programmer by *shares* and *accesses* ports. The shared data is an array of float.

semantics could be obtained by relying on the *transparent data access model* described in Section 3. To achieve this, we define two kinds of data ports. On one hand, a data *shares* port gives an access to a shared data. On the other hand, a data *accesses* port enables a component to access a data exported through a *shares* port.

Interfaces attached to ports precisely clarify how interactions between components can occur. In order to access a data, two types of interfaces are required: one for accessing the data, named *data_access*, and one for making the data available to other components, named *data_shared_port*. The *data_access* 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 3 shows the API of the *data_access* interface offered to the programmer of a component. 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_shared_port* 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 a data provided by a *shares* port. Typically, it contains an operation which returns the global data ID (cf Section 3.2).

Let us stress that a major difference between *data ports* and classical ports lies in the activity implied. While classical ports are intrinsically attached to the notion of message which triggers some reactions from one component, data ports are passive. By default, there is no notification of events such as data modifications, for instance. The principal role of a *shares* port is to allow a data (logically) internal to a component to be accessible from other components. Once an *accesses* port of a component `A` is connected to a *shares* port of

```
// The data type.
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 4: An OMG IDL3+ example of *shares* and *accesses* data ports.

a component B, the associated data is immediately accessible to component A. No further communications between the two components are needed to access the data. This access is handled by the underlying data sharing service.

5.2 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 CCM and JUXMEM. Extending CCM with *data ports* capabilities requires to enhance the Interface Definition Language (IDL3) of CCM. Our proposal therefore consists in introducing two new keywords in the IDL3 of CCM: *shares* and *accesses*. Such an extended IDL is named *IDL3+*.

Figure 4 shows an example of an IDL3+ specification for a gravitational N-body simulation in a three dimension space. In such application and 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 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 component `body` is set at assembly time of the application. Figure 5 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 exists locally in the component without dealing with explicit data transfer. The used synchronization primitives are mapped on the synchronization API of JUXMEM. Note that

```
class Body_impl : virtual public CCM_body {
  private:
   data_access* positions_data_port;
   // To identify the body in the space
   int id;
};

void Body_impl::computePosition() {

 float* data_ptr[];
 data_ptr = position_data_port.get_pointer();

 // Reading position of others
 positions_data_port.acquire_read();
 for (i = 0; i < N; i ++) {
   addToLocalComputation(data_ptr[i]);
 }
 positions_data_port.release();
 // Update our position
 position_data_port.acquire(data_ptr[id]);
 updateOurPosition(data_ptr[id]);
 position_data_port.release();
}
```

Figure 5: C++ implementation of a N-body application with data ports.

for the sake of clarity, synchronization mechanism between each step of the simulation is not shown. However, this can easily be implemented with event broadcasting.

5.3 A portable implementation of IDL3+ on top IDL3

We have implemented the data port model onto CCM using MicoCCM [22] (version 2.3.11) for the CCM side and JUXMEM [20] (version 0.2) as a data sharing service for data accesses. We imposed ourself two constraints. The first one is to be independent from any CCM implementations for portability reasons. The second one is to be compliant with the CORBA specifications. Whereas, this strategy may not lead to the most efficient implementation, it will allow us to validate our proposition without the complex burden of modifying an existing CCM framework.

Before introducing the implementation of *data ports*, let us first briefly describe the internal architecture of a classical CCM component, as shown on Figure 6. A CCM com-

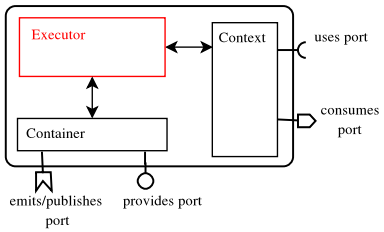


Figure 6: The internal architecture of a classical CCM component.

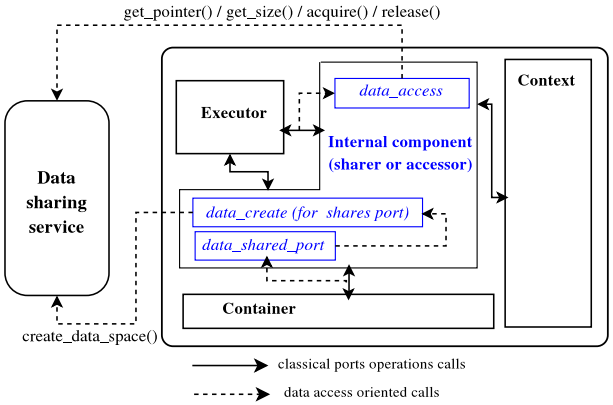


Figure 7: The internal architecture proposal

ponent is usually made of an *executor*, a *container* and a *context*. The *executor* contains the functional code of the component implemented by a developer. The *container* and the *context* are generated by an IDL3 compiler. The container glues the executor and the extern word connected to *server* ports, i.e. the *provides* and *consumes* ports. The *context* has the same role but for *client* ports like the *uses*, *publishes*, and *emits* ports. The glue is achieved by providing well defined interfaces to the executor.

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 initial components (`sharer` and `body`). 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 7. However, this intermediate context is transparent for developpers of compo-

```
interface compute {
 void inverse(in matrix m, out matrix n);
 void diagonalize(in matrix m, out matrix n);
}

component Server {
   provides compute for_client;
};
component Client {
   uses compute to_server;
};
```

Figure 8: IDL definition of a `Client` and `Server` components.

nents. Our internal components indeed 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. For example, our implementation supports two other data storage provided by a *local shared-file* and *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.

Last, data parameters such as the data replication degree or the consistency protocol are considered as classical configuration attributes of a component, and therefore defined at the assembly step.

6 Enabling data sharing on operation invocation

Previous section has dealt with the issue of sharing a data between component. This section is about the issue of passing a shared data as a parameter of an operation invocation. Our abstract model is instancied as an extension of the CCA model, as it requires dynamic port creation capabilities. In the remainder of the section, we consider the example based on two components: a `Client` connected to a `Server` which provides a `compute` interface, as described in Figure 8. The `Client` component invokes the `diagonalize` operation on a data produced by the `inverse` operation.

6.1 Illustration of the abstract model

In order to share a parameter while invoking an operation, a notation needs to be introduced in IDL languages. This new notation should express that a parameter is passed by reference instead of value. For this purpose, the ampersand character (&) appears to be an obvious choice as it already fulfill this meaning in the C++ language. The modification of IDL languages is thus immediate and could be done with this conventional reference notation.

However, component models should enforce all incoming and outgoing communications to go through some well defined ports. Therefore, the introduced notation needs to be translated into some ports. The idea is to straightforwardly map such shared parameters to the data ports introduced in Section 5: at an operation invocation, a data port has to be associated to each parameter passed by reference. Note that there may be a difference in the implementation of the caller and the callee of the operation. On the caller side, data ports need to be explicitly created by the developer of the component to make the data available to the callee to support multiple simultaneous invocations. On the callee side, data ports can be automatically generated by an IDL-based compiler if the compiler already generates skeleton code for handling incoming calls. We introduce two notations: *AccessPort* and *SharesPort* which respectively represent the handler type on an *accesses* port and on a *shares* port.

Let us now illustrate the use of this model through the previously introduced example. In our example, the caller is the component `Client` displayed in Figure 8, whereas the callee is the component `Server`. Figure 9 shows how the component `Client` invokes the operation `inverse` on a matrix. First, the client allocates and initializes a piece of data with a conventional method. Then, for the `in` parameter two steps are mandatory before invoking the operation: 1) create a `shares` data port for a data of type `matrix`, 2) associate the allocated data to the data port. This permits to make the data available from outside the component. Finally, for the `out` parameter, an *accesses* data port needs to be created. This is required as it provides the ability to use the reference of the remote shared data returned by the `inverse` operation.

The code of the callee is shown on Figure 10. The `in` parameter of the `inverse` operation has been converted to an *accesses* port: the implementation of the operation will access an already created data. For the `out` parameter, the produced result needs to be associated to the *shares* data port. It also mainly applies for `inout` parameters.

Let us stress that the data reference extension is different from the parameter modes. Classically, e.g. in OMG IDL or CCA SIDL, the parameter modes determine the owner of data. For an `in` mode, the callee can not reallocate the data while it is possible for `inout` mode. For `out` mode, the callee is responsible to allocate the data. Our reference notation specifies that the data is *shared*. Hence, it is compatible with parameter modes. For `in`

```
// Memory allocation internal to the component
ptr = allocate_matrix(...);
// Creating a shares data port
SharesPort* dp1 = createSharesPort(matrix)
// Associate the data to the data port
dp1->associate(ptr, size);
// Creating an access data port
AccessPort* dp2 = createAccessPort(matrix);
// Invoke the method
to_server->inverse(dp1, dp2);
```

Figure 9: Steps for invoking `inverse` operation from the component `Client` using shared data as parameters.

```
inverse_impl(AccessPort& dpm,
             SharesPort& dpn) {
  // Retrieve pointer & size of the data
  ptr = dpm.get_pointer();
  size = dpm.get_size();
  // Inverse the matrix with a F77 function
  res = f77_inverse(ptr, size);
  // Associate result with shares data port
  dpn->associate(res, size);
};
```

Figure 10: Implementation of the `inverse` operation on the `Server` component with shared data as parameters.

modes, the semantic is the following: the caller provides an access to an already created data. Therefore, the callee can access the data but without the right to reallocate it, as it may be used by other components or by the caller. For the `out` mode, the caller receives a reference to a shared data allocated by the callee. However, for the `inout` mode, the callee may reallocate the input data. More precisely, as the data is being shared, and thus possibly accessed by several components, it is not possible to simply deallocate an `inout` parameter of an operation. Instead, the data reference must first be de-associated from the shares data port, then another data reference is associated to the port, as shown in Figure 11.

```
interface i1 {
   void f(in matrix& m1);
}
interface i2 {
   void g(inout matrix& m2);
}

void f(AccessPort& ap) {
 // ap is associated to a shared data D1
 // ap and p1 references it forever
 p1 = a1.get_pointer();
 for(;;)
   ... = p1[...];
}

void g(SharesPort& sp) {
 // sp is associated to the shared data D1
 // sp is de-assiocated from D1
 sp->deassiociate();
 // A data D2 is created
 ptr2 = malloc(size2);
 // and associated to sp
 sp->associate(ptr2, size2);
 // sp is now associated to D2
}

Client codes:
 // p references D1, a shared data
 // f is invoked asynchronously on D1
 c1->async_f(p);
 // g is invoked synchronously on D1
 c2->g(p);
 // p now references D2, whose size can
 // differ from D1's size
```

Figure 11: Pseudo-code illustrating the behavior of `in` and `inout` parameter modes.

6.2 Case study: extending CCA

We illustrate a projection of this model as an extension of the CCA model, as this component model provides dynamic port creation. Moreover, this proves that our data port concept is generic. This discussion is based on the version 0.7.8 of the SIDL specifications of CCA.

```
interface AccessPort : Port {
 // Operations described in Figure 3
}
interface SharesPort : AccessPort {
   void associate(opaque ptr, long size);
   void deassociate();
}
interface extended_services : services {
  AccessPort
  createAccessPort(in string  portName,
                   in Type    typeName,
                   in TypeMap properties);
  SharesPort
  createSharesPort(in string  portName,
                   in Type    typeName,
                   in TypeMap properties);
  void destroyPort(in string portName);
}
```

Figure 12: A SIDL example of CCA specification extension with respect to the abstract data port model. The `opaque` type is used not to have an `associate` operation per data type. Exceptions have been omitted because a lack of space.

In our projection, we consider data port as a new type of port. Hence, as illustrated at Figure 12, we need to introduce new operations to the `Services` interface which deals with port management in CCA specifications. These operations create the data port as explained in the abstract model. The projection of the *AccessPort* and *SharesPort* interface, shown in Figure 12, is then straightforward.

The two other interfaces of CCA specifications that are of interest from us are the `ConnexionID` and the `BuilderService` interfaces. The `ConnexionID` interface, which describes a connexion between components, may be kept unchanged with the convention that a *shares* port acts as a *provides* port and an *accesses* port acts as a *uses* port. The `BuilderService`, which is an interface dealing with component composition, may also be kept unchanged with the same convention. However, it seems better to add two operations to the `BuilderService` to easily retrieve *shares* and *accesses* ports. It does not seem to be suitable to insert a new `connect` operation with the aforementioned convention.

7 Conclusion

Programming grids is still a challenging issue. The software component concept appears to be very promising, as it enforces building application by assembling rather than by programming. Existing software component models assume that data are internal to components, and thus are not directly accessible. Therefore, data need to be explicitly exchanged between components.

The contribution of this paper is on the enhancement of component models for data management. It is twofold. Firstly, our proposal provides a *transparent access* to a shared data within components, using a data sharing service. This is achieved by defining a new class of ports: the *data ports*. Two types of data ports have been introduced: the *shares* data port and the *accesses* data port, which respectively allow to make a data available to other components and to access a data provided by another component. Secondly, our proposal enables the use of shared data as parameters of operations provided by a component. It is based on the dynamic creation of data ports and on the extension of IDL languages with the conventional reference notation.

Though different implementations of the data sharing service are possible, JUXMEM appears as a promising choice, as it already provides a *transparent data access model*. As a proof of concept, we have therefore implemented the runtime part of the data port as an extension to the CORBA Component Model IDL3 and based on JUXMEM. The prototype has been successfully tested through a synthetic application.

Nevertheless, an evaluation of the prototype to fully validate our data port concept is required. To this end, we plan to first develop an IDL3+ compiler to be able to benchmark several scenarios. We are also considering to validate our proposal of shared operation parameter with a distributed memory CCA implementation. Furthermore, we are currently adapting a code coupling application, from the HydroGrid project [19] to actually compare the data port approach to standard data management approaches, in terms of programming difficulty but also in terms of performance.

References

[1] GAMESS - General Atomic Molecular Electronic Structure Systems. `https://gridport.npaci.edu/GAMESS/`.

[2] The Biomedical Informatics Research Network (BIRN). `http://www.nbirn.net/`.

[3] The Palomar Digital Sky Survey (DPOSS). `http://www.astro.caltech.edu/~george/dposs/`.

[4] US National Virtual Laboratory (NVO). `http://www.us-vo.org/`.

[5] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance Grid programming in the Grid.it project. In V. Getov and T. Kielmann, editors, *Proc. of the Workshop on Component Models and Systems for Grid Applications (June 2004, Saint Malo, France)*. Springer, January 2005.

[6] William Allcock, Joseph Bester, John Bresnahan, Ann Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, 2002.

[7] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. JuxMem: An adaptive supportive platform for data sharing on the grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, November 2005.

[8] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing. *Concurrency and Computation: Practice and Experience*, (17):1–19, September 2005.

[9] 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, August 1999.

[10] L. Barroca, J. Hall, and P. Hall. *Software Architectures: Advances and Applications*, chapter An Introduction and History of Software Architectures, Components, and Reuse. Springer Verlag, 1999.

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

[12] Nathalie Furmento, Anthony Mayer, Stephen McGough, Steven Newhouse, Tony Field, and John Darlington. ICENI: Optimisation of component applications within a grid environment. *Journal of Parallel Computing*, 28(12):1753–1772, 2002.

[13] Jeff Magee, Naranker Dulay, and Jeff Kramer. A Constructive Development Environment for Parallel and Distributed Programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 4–14, Pittsburgh, US, March 1994.

[14] M. D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, Belgium, 1969. Scientific Affairs Division, NATO.

[15] Reagan Moore. Preservation environments. In *Proc. 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2004)*, Adelphi, Maryland, USA, April 2004.

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

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

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

[19] The HydroGrid Project. `http://www-rocq.inria.fr/~kern/HydroGrid/HydroGrid.html`, 2004.

[20] JuxMem: Juxtaposed Memory. `http://juxmem.gforge.inria.fr/`, 2005.

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

[22] The Mico Corba Component Project. `http://www.fpx.de/MicoCCM/`, 2003.