Cluster Comput (2007) 10: 265–276
DOI 10.1007/s10586-007-0034-7

Combining data sharing with the master–worker paradigm in the common component architecture

Gabriel Antoniu · Hinde Lilia Bouziane · Mathieu Jan · Christian Pérez · Thierry Priol

Published online: 28 June 2007

Abstract Software component technologies are being accepted as an adequate solution for handling the complexity of applications. However, existing software component models tend to be specialized to some types of resource architectures (e.g. in-process, distributed environments, etc.) and/or do not provide a very high level of abstraction. This paper focuses on handling *data sharing* on operation invocations between components as a solution allowing applications to be efficiently executed on all kinds of resources. In particular, the data sharing pattern appears in master–worker applications, when workers need to access only a part of a large piece of data, either in read or write mode. This approach is applied to the Common Component Architecture model. Its benefits are discussed using an image rendering application.

Keywords Software components · Data sharing · Master–worker paradigm · Grid computing · Common component architecture

1 Introduction

e-Science application programmers are facing a serious challenge for the following years, due to both the intricacy of these applications and the increasing variety of computing resources. For instance, resources may be multicore multithread multiprocessors, clusters, or grid computers. To cope with these two concerns, it is necessary to provide programming models able to make the programming of these applications simple, independent of the target computing infrastructures, while preserving high-performance. Software component models aim to handle the increasing complexity of today's applications [26]. However, component models need to be adapted for addressing high performance *independently* of the resource infrastructure. In previous work, we have addressed several shortcomings of these models: encapsulation of a parallel code into a component [25], support for the master–worker programming paradigm [11], and support for data sharing among components [7].

Communication between components is to a large extent based on the exchange of messages, which may be associated with control transfers (RPC or RMI) or with data transfers (events, streams, etc.). For many applications, especially e-Science ones, this communication model is not satisfactory because it requires to explicitly compute which pieces of data need to be exchanged. When data structures are complex and sparse in memory, or when data access patterns are irregular, this communication model is not suitable nor efficient: a shared memory communication model seems more appropriate. The challenge is thus to support this communication model while keeping the good properties of software components, that is to say: the composition, the port-based communications, the deployment unit, etc.

This paper aims at showing how an existing component model, the Common Component Architecture (CCA) [10], can be extended to support shared data in general, and focuses more particularly on the master–worker paradigm. To this end, it fills the gap between data sharing and operation invocations. The proposed approach is illustrated with an image rendering application that has been chosen as it requires to share data and exhibits a master–worker pattern.

Section 2 describes the state-of-the art with software component models in general, and briefly summarizes our previous works on the use of the transparent data access paradigm and on the master–worker paradigm within com-

G. Antoniu · H.L. Bouziane · M. Jan · C. Pérez (✉) · T. Priol
IRISA/INRIA, Campus de Beaulieu, 35042 Rennes cedex, France
e-mail: Christian.Perez@irisa.fr

ponent models. Section 3 presents CCA and proposes extensions to the CCA model, in order to support the two concepts, data sharing and master–worker paradigm. Section 4 describes how data sharing can be enabled on operation invocation, first independently of any component model, then in the CCA model. Its benefits with respect to an image rendering application are discussed in Sect. 5. Section 6 concludes the paper and outlines some future works.

2 Software component models

Even though software component technology is not a recent idea [23], it has been emerging for not more than few years [9]. The re-emergence of this technology appears as a response to the failure of used programming models to deal with the continuous growth in complexity of applications. For instance, the object-oriented approach has failed to handle very complex and large applications, mainly because dependency between objects are hidden in the code.

Recently, several component models have been proposed like the OMG CORBA Component Model [24] (CCM), the Common Component Architecture [10] (CCA), the ObjectWeb FRACTAL component model [12], GRID.IT [1], DARWIN [20]), etc. Their aim is to facilitate the design of applications and reduce the complexity of their building process. Even though these component models offer different properties, they have all a unified definition of a software component, which is based on the Szyperski [26] definition. According to Szyperski, a 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*. This definition considers a component as a black box for which the following concepts are attached:

Composition. A component is able to be composed with other components by a third party. This composition is possible thanks to well-defined interfaces that allow components to interact. Some contracts are attached to these interfaces and must be accepted. They allow specifications of constraints related to interaction, such as security.

Ports. To be able to interact with other components, a component defines external interfaces named ports. A port is a programming artifact to which an interface can be attached. It can be categorized in two types: a *client* or *server* port. The interaction between two components is then performed by connecting a *client* port of a first component to a *server* port of a second component with compatible type. In existing component models the interaction mode through ports can be method invocations or message passing (events or streams).

Assembly. The assembly phase is the process of building an application. In particular, an assembly generates a specification of component instances and their interconnections. Component assemblies can be described using an Architecture Description Language (ADL), like for example in CCM or in FRACTAL. Another approach consists in using run-time composition, like for example in CCA, CCM or in FRACTAL.

Deployment. A component is a binary unit of deployment. It should reference an implementation (binary code) and the constraints associated to it, like the operating systems, processors and amount of memory requirements. It may also contain several implementations and so, alternative requirements. These properties help a deployment tool to decide on which resource in a Grid or other distributed environment, an instance of the component may be installed.

Despite the fact that component models offer facilities to program complex applications, they do not support all distributed application paradigms in an easy way. The remainder of this section recalls previous work about the enhancement of component models with the support of two paradigms: (1) the *data sharing* paradigm and (2) the *master–worker* paradigm.

2.1 Transparent data access model and component models

2.1.1 Enabling transparent access to data

An attractive programming paradigm allowing data to be shared by multiple concurrent entities is the *shared memory* paradigm. Its advantage relies on the ease of programming: multiple entities can read/write data in a global space without any need to explicitly handle data location. This concept has been successfully applied in several contexts: (1) multithreading within the same process, (2) data segment sharing among multiple processes running on the same host, or (3) global data sharing across a cluster of workstations through Distributed Shared Memory (DSM) systems. This concept has not really been exploited in grid environments. Currently, the most widely-used approach to manage data on distributed environments is based on a set of *data catalogs*, such as Giggle [16] or LDR [28] for instance. The goal of theses data catalogs is to return locations of a given data based on one or several of its attributes. Therefore, an *explicit data access model* is used, where clients have to move data to computing servers. In this context, grid-enabled file transfer tools have been proposed, such as GridFTP [2], RFT (Reliable File Transfer, an OGSA service for asynchronous data transfers), or DiskRouter [19].

Let us recall that one of the major goals of the grid concept is to provide an easy access to the underlying resources, in a *transparent* way. The user should not need to be aware of the localization of the resources allocated to applications.

When applied to the management of the data used and produced by applications, this principle means that the grid infrastructure should automatically handle data storage and data transfer among clients, computing servers and storage servers as needed. It should also handle fault tolerance and data consistency guarantees in such dynamic, large-scale, distributed environments and again, in a transparent way.

In order to achieve a real virtualization of the management of large-scale distributed data, a step forward has been made by enabling a *transparent data access model* through the concept of *grid data-sharing service* [5]. In this *transparent data access* approach, the user accesses data via global identifiers, which allow to do argument passing by reference for shared data. The service which implements this model handles data localization and transfer without any help from the programmer. The data sharing service concept is based on a hybrid approach inspired by 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 three main properties.

Persistence. The data sharing service provides persistent data storage and allow the applications to reuse previously produced data, by avoiding repeated data transfers between clients and servers.

Data consistency. Data can be read, but also *updated* by the different codes. When data is replicated on multiple sites, the service has to ensure the consistency of the different replicas, thanks to *consistency models and protocols*.

Fault tolerance. The service has to keep data available despite disconnections and failures, e.g. through replication techniques and failure detection mechanisms.

The concept of data-sharing service is illustrated by the JUXMEM software experimental platform, described in detail in [6]. Its implementation relies on the JXTA [27] generic P2P framework. The JUXMEM API provides to users classical functions to allocate and map/unmap memory blocks in a globally shared space: `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` or `juxmem_acquire_read`. This allows the implementation to apply consistency guarantees according to the consistency protocol specified by the user at allocation time. The choice of a C-style malloc interface for the API of JUXMEM is motivated by the targeted e-Science applications of JUXMEM: they mainly share arrays of data.

2.1.2 Data sharing and component models

In current component models, *ports* are defined based on the assumption of an explicit communication operation between two components. As such, components are only able to deal with data as a part of a message actually exchanged between two components. As explained earlier, sharing data among multiple components may be more appropriate for some applications where data structures are more complex and access patterns are more irregular. Implementing such a functionality using "classical" ports inside a component-based application is possible. For instance, the shared data can be physically located into a component and accessed by other components through provided ports. 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. In such a case, the functional code of a component would have to 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 would be handled within the functional code of components, leading to an unnecessary increase in the complexity of applications.

To summarize, existing software component models do not efficiently support a *transparent data access* model. We claim that this is a limitation for current component models as data persistence, consistency and fault-tolerance are not handled.

2.1.3 Data port model

In a previous work [7], we proposed an approach to allow transparent data sharing in component models. The idea is to logically attach the shared data to a component. The main role of this component is to provide a global reference of the data to share it by several components. The localization of the data and the management of concurrent accesses to this data rely on the *transparent data access model* described in Sect. 2.1.1.

More concretely, the proposed approach is based on an additional family of ports named *data ports*. Two kinds of data ports were defined: a *shares* port to give an access to a shared data and an *accesses* port to enable a component to access a data exported through a *shares* port. In a transparent way for the programmer, such ports may delegate the data to be managed to a data sharing service.

In order to access a data, an interface named *AccessPort* is implicitly associated to a data port. The API of this interface is shown in Fig. 1. This interface is available through *accesses* ports as well as through *shares* ports. Indeed, a component that *shares* some data may also need to access the data. The interface provides `get_pointer/get_`

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

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

`size` primitives to respectively retrieve a pointer to the shared data and its size. This interface is currently enough for scientific codes, mainly written in FORTRAN, that typically handle arrays. 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 a given data, whereas `acquire` sets a lock in exclusive mode.

A component which aims to access a data through an *accesses* port should have this port connected to a *shares* one. Connecting the two ports implies passing the reference of the shared data from the shares component to the accesses component. The shared data was previously allocated and associated to the *shares* port by an appropriate interface provided only on the *shares* port side.

Let us stress that a data port allows a data to be shared between components without worrying the user with the mechanism used to share the data. Such a mechanism is expected to be the memory between components collocated within the same process, a shared memory segment for components in two different processes but on the same host, a DSM for cluster and a grid data-sharing service like JUXMEM for grids. It is the responsibility of the component model framework not of the component implementation.

We projected this model on the CORBA Component Model and we realized a prototype implementation as a proof of the concept and of the facilities offered to the programmer. JUXMEM was in particular used to share data between components located on distinct clusters. An application example using data ports can be found in [7].

2.2 Master–worker paradigm and component models

2.2.1 The importance of the master–worker paradigm

The master–worker programming paradigm is widely used in distributed applications. For instance, parametric applications are based on such a paradigm: several workers simultaneously execute a same code but with different parameter values. Numerous research activities are dealing with the design of *master–worker* software grid-enabled environments such as for global computing systems (SETI@Home [3], XtremWeb [18], BOINC [4], etc.) or for network-enabled server environments (DIET [14], NetSolve [15], Nimrod/G [13]).

Such grid-enabled environments relieve the programmer of dealing with non functional tasks: collecting workers through a large network to participate in a computation, managing requests transfer and scheduling from a master to its workers and dealing with fault tolerance on volatile resource infrastructures. However, these environments only focus on supporting the master–worker paradigm. Therefore, such environments seem not to be convenient to support an application partially based on the master–worker paradigm. However, if such an environment is not used, a programmer has the burden of managing workers. Consequently, the code complexity is increased by the previously cited non functional concerns. Moreover, it is further increased since implementing request transport policies from masters to workers may be very complex as they may depend on the underlying execution environments. These drawbacks remain valid for component-based applications as explained hereafter.

2.2.2 Master–worker paradigm and component models

Current component models do not offer efficient mechanisms to provide a suitable level of abstraction for developing *master–worker* applications. More explicitly, a component based *master–worker* application is mainly composed of an instance of a *master* component and several instances of a *worker* component. To assemble the *master* with the *workers* with existing component models, the programmer should consider the management of the workers.

The workers management may be introduced inside the implementation of the components or may be considered when the application is assembled. In the last case, the programmer may introduce components between the *master* and the *workers*. These additional components have the responsibility to manage requests transport. The programmer should also select both the number of *workers* to be defined in the assembly and an adequate requests transport policy. However, this choice depends on the resource infrastructure on which the application will be deployed. As a consequence, component models fail in providing the independence of the application development from the used resource infrastructures. Moreover, the transparency of workers management which can be offered by existing *master–worker* environments is missing.

2.2.3 A model for the master–worker paradigm

In a previous work [11], we aimed at extending software component models to increase the abstraction level for *master–worker* applications. The proposal only requires a designer to specify a set of *worker* instances to which a

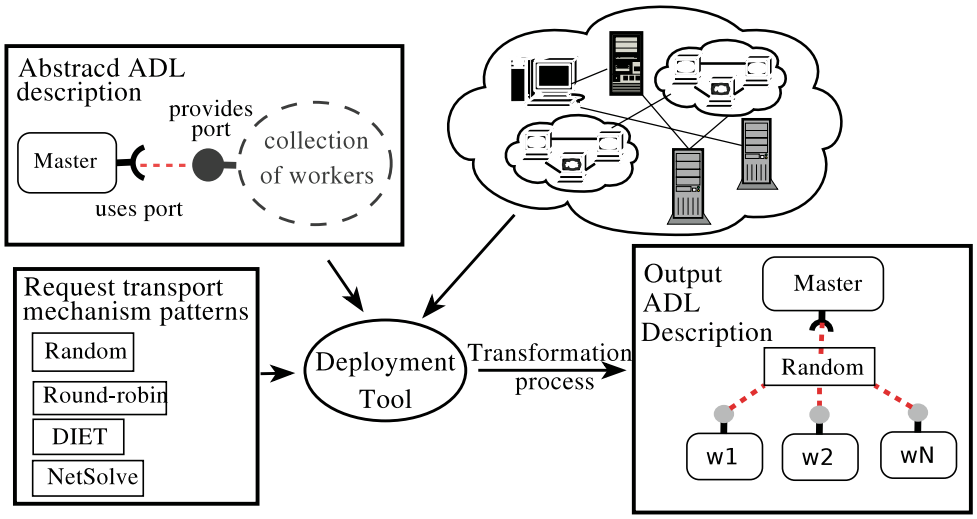

Fig. 2 An overview of a *master–worker* application model

master component is connected. Request transport concerns are handled separately while advanced transport policies are possible. Existing master–worker environments may be utilized. Last, the number of *workers* is handled as a nonfunctional property and thus may be delegated to an adaptivity service.

Figure 2 presents an overview of the different elements of the proposal. The application designer specifies a *collection* of *worker* components. A collection is a set of *exposed* ports, bound to some internal component ports. It is described with an *abstract architecture description*. Independently, request transport *patterns* are defined by some experts. They represent request delivery policies that may be used between *master* and *worker* components. They should be based on software components, even though existing *master–worker* environments, such as the hierarchical Network Enabled Server DIET [14], may be used.

Once the deployment environment is known, an initial number of worker components and a suitable requests transport policy can be decided. From these choices, the abstract architecture description is converted into a concrete ADL description during a *transformation* process. In the example of Fig. 2, the selected pattern is a hierarchical random scheduling policy implemented by a tree. The concrete ADL is a standard ADL, typically the ADL of the component model.

This generic model was projected to two specific component models, CCM and FRACTAL [11].

3 Extending CCA

This section applies the generic models of both data sharing and master–worker paradigms to CCA. The projection is based on an extension of the CCA specifications (version 0.7.8). Data ports, collections and request delivery policy patterns are introduced by adding new operations in the framework service APIs. Before exposing these extensions, let us first provide an overview of the CCA component model and specifications.

3.1 Overview of CCA

The Common Component Architecture [10] 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. The specification of such ports is done by using the Scientific IDL. Unlike many component models, 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 on run-time calls, as ports are dynamically added or removed to components.

The specifications of CCA define standard SIDL interfaces that should be provided by any CCA compliant framework implementation. Three of them are of particular interests for this paper. First, the `Port` interface is an empty interface which all ports must derive from. Second, the `BuilderService` interface deals with component creation as well as composition. For example, a user can create a component instance thanks to the `createInstance` operation and connect two ports via the `connect` operation. It may also obtain the list of ports available for a given component through introspection mechanisms available in this `BuilderSevice` interface. Third, the `Services` interface deals with port management with respect to a component implementation. A unique instance of this interface is given to each component instance. Through this interface, a component can declare *provides* ports (respectively

uses port) by respectively calling the `addProvidesPort` operation (resp. the `registerUsesPort` operation). The component can subsequently obtain a reference to a declared port as a result of a `getPort` invocation.

3.2 Transparent data access model in CCA

Section 2.1 has described a generic proposal about transparent data sharing between components based on data ports. This section presents how data ports can be specified and implemented in CCA. It is first described from a user point of view and then from a framework implementer point of view.

3.2.1 User view

Data ports are distinct from *provides* and *uses* ports. Hence, as shown in Fig. 3, two new operations need to be added to the `Services` interface to handle them: the `createSharesPort` and the `createAccessPort` operations. They respectively creates a *shares* data port and *accesses* data port for a given data type.

These ports behaves differently from classical *provides*/*uses* ports: data ports come with their own pre-defined interfaces. For an *accesses* port (resp. a *shares* port), a programmer has access to the `AccessPort` interface (resp. a `SharesPort` interface). As shown in Fig. 3, the API of the `AccessPort` contains operations to acquire a pointer to data, to read/write them and to handle their consistency. The `SharesPort` interface contains the same operations as the `AccessPort` interface, since it inherits `AccessPort`. A programmer may indeed access a data it shares. However, it also offers two operations to deal with the association of a piece of data with the port. The `associate` operation provides the ability for a memory area to be attached to a data port, whereas `disassociate` is the opposite operation. A component implementer is still responsible for allocating/freeing the memory. Note that the actual memory area attached to a *shares* port may dynamically change.[1] However, it is transparent for all *accesses* port connected to it. Similarly, it is possible to allow a user to specify the address to which an *accesses* port maps the data. This can be done by extending the `AccessPort` interface as well as the properties of the `createAccessPort`.

```
interface AccessPort : Port {
  opaque get_pointer();
  long get_size();
  void acquire ();
  void release();
}
interface SharesPort : AccessPort {
  void associate(in opaque ptr, in long size);
  void disassociate();
}
interface ExtendedServices : Services {
  void createAccessPort(in string  portName,
                        in string  typeDataName,
                        in TypeMap properties);
  void createSharesPort(in string  portName,
                        in string  typeDataName,
                        in TypeMap properties);
}
```

Fig. 3 SIDL specifications for the CCA projection of data ports

A component implementation obtains a reference to an object providing an `AccessPort` or `SharesPort` interface through the `getPort` operation of the `Services` interface. The connection process is also unchanged from the programmer point of view: a connection between a *shares* and an *accesses* ports is done with the `connect` operation. If the data type of both data ports are not compatible, an exception is raised. Moreover, programmers do not have to worry about the underlying mechanism used to share data between components: it is the responsibility of the framework. Finally, a complete specification will require to introduce relevant introspection operations in the `BuilderService` interface. Such operations may for instance allow the programmer to obtain the full list of *shares* ports of a given component.

3.2.2 Framework implementer view

To provide the previously described view to programmers, the framework should internally be able to distinguish between classical and data ports. This distinction enables the framework to perform appropriate actions. For instance, the framework has to create and attach an instance of `AccessPort` or `SharesPort` when a data port is created. Also, when connecting data ports the data type needs to be checked in order to ensure the compatibility of ports. Another modification to the framework is required for the configuration process of the `AccessPort` object. A reference of the data exported by the `SharesPort` needs to be given to the `AccessPort`. As the interface has to be independent of any data types, we propose to use the `opaque` SIDL type that represents a pointer to the local memory. Depending on the localization of the components, several technologies may be used to actually implement the data sharing. If the data is located in the same process, there is nothing to do. For components located into distinct processes but on the same node, shared memory segment provided by most operating systems can be used. On a cluster, a distributed shared memory system may be used. Finally, on grid environments a grid data-sharing service, such as JUXMEM, can be used. Advanced framework may dynamically decide of the data sharing mechanism to be used depending on their availability and the localization of components. In [7], we

[1] `SuspendPort`/`ResumePort` operations need to be added to the `Services` interface to do it atomically.

show that the interface with respect to the framework may be generic and contains operations like data allocation and freeing, reading, writing and synchronization.

To summarize, the proposed extension of the CCA specifications to support data ports implies the extension of the `Services` interface and the re-implementation of the `BuilderService` interface in the framework. Operations like `connect`, `getPort` and operations related to introspection interests are also concerned by these changes. With the experience gained with the implementation of data ports performed in CCM [7], theses modifications seem to be limited and mostly straightforward to implement.

3.3 Master–worker paradigm in CCA

Our proposal for handling the master–worker paradigm, presented in Sect. 2.2, was applied to component models that provide an ADL language. However, the aim of this model is to be generic and therefore being possibly used in any component models offering a similar level of abstraction. This section presents a projection of this model on CCA, a component model without an ADL language. The user view is first described, followed by the framework implementer view. As for data ports, this projection is also performed by extending the CCA specifications.

3.3.1 User view

The master–worker paradigm is seen as the particular case of the connection of a component (the master) that uses a port provided by a collection of components (the workers). Hence, three sets of interfaces are of interests for the user. The first set deals with the collection creation and port bindings. The second set, which is optional, is about the handling of request transport policy patterns. The third one, which is also optional, is related to the management of the dynamic variation in the number of elements inside a collection.

As CCA does not provide any ADL language, a collection needs to be a concrete entity for the programmer. Our proposal consists in the encapsulation of a collection description in a distinct component. As illustrated in Fig. 4, we extend the `BuilderService` with operations related to collection creation and port bindings. For the sake of simplicity, we decide to provide an operation very similar to the `createInstance` operation. The only difference is that the created component receives a reference of type `CollServices` instead of `Services`. Then, the component may create as many ports and (logically internal) components as it wants. Internal collection component ports need to be connected to the collection port. However, this connection is done via the `bind` operation instead of the classical `connect` operation as these ports are of same kind. For example, for a collection of workers, both the collection and the workers have declared a *provides* port. Last,

```
interface CollServices : Services {
  CollectionID getCollectionID();
}
interface CollectionID {
  string getCollectionName();
}
interface BindingID {
  CollectionID getCollectionID();
  string getCollectionPortName();
  ComponentID getComponentID();
  string getComponentPortName();
}
interface CollBuilderService : BuilderService {
  CollectionID createInstanceCollection(
               in string instanceCollName,
               in string className,
               in TypeMap properties);
  BindingID bind(in CollectionID collID,
                 in string collPortName,
                 in ComponentID elem,
                 in string elemPortName);
  array<BindingID> getBindingIDs(
                   in CollectionID collID,
                   in string portName);
}
```

Fig. 4 Collection and binding related specifications for CCA

```
interface PatternInstantiation : Port {
  void instantiatePattern (in CollectionID collID,
                           in string collPortName,
                           in ComponentID user,
                           in string usingPortName);
}
interface PatternBuilderService :
          CollBuilderService {
  void setPatternPort(in CollectionID collID,
                      in string portName,
                      in PatternInstantiation p);
}
```

Fig. 5 Pattern related specifications for CCA

only ports of type *provides*, *uses* or *accesses* can be bound. It appears meaningless for *shares* ports.

The next step, which is optional, is to associate a pattern with a *provides* port of a collection thanks to the `setPatternPort` operation described in Fig. 5. This step is optional as the framework should provide a default pattern as explained in the framework implementer view. We choose to represent a pattern by an interface (`PatternInstanciation`). The `instantiatePattern` operation of this interface is called whenever a *uses* port is connected to a *provides* port of a collection. The implementation of such an operation has to actually connect the external *uses* port to some internal *provides* port. It may insert components to achieve more or less sophisticated request transport policies like round-robin policies or hierarchical policies.

The pattern is only applied on *provides* ports because it is meaningless for *uses* ports. When connecting a component with a *provides* port to a *uses* port of a collection, *uses* ports

```
interface CollectionManagement : Port {
  bool addElement(in CollectionID collID);
  bool subElement(in CollectionID collID);
}
interface DynCollServices : CollServices {
  void setCollManagementPort(
       in string portName,
       in collectionManagement port);
}
interface DynCollBuilderService :
          CollBuilderService {
  bool addElementToColl(in CollectionID collID);
  bool subElementToColl(in CollectionID collID);
}
```

Fig. 6 Dynamic collection management related specifications for CCA

are directly connected to the *provides* port of the component. Note that it is possible to connect a *provides* port of a collection to a *uses* port of another collections. In this case, it is the responsibility of the pattern of the *provides* port to correctly connect all *uses* ports of the other collection. It is worth to note that pattern interfaces are expected to be implemented by scheduling experts, not by the end user.

The third step, which is also optional, is dedicated to the dynamic management of a collection. For instance, the number of elements of a collection may be adapted according to its load. This adaptation may be done by the collection itself or by an external component. Hence, the `DynCollBuilderService` interface, shown in Fig. 6, provides operations to add or remove components to/from a collection. The role of this interface is to notify the collection of such events. Hence, a collection willing to receive such events has to register a port implementing the `CollectionManagement` interface. The registration is done thanks to the `setCollManagementPort` operation of the `DynCollServices`: on its creation, a collection in fact receives a reference to a `DynCollServices`. Addition or removal operations return a boolean to indicate either the success or the failure of the request (no more resources, components in use, etc.).

3.3.2 Framework implementer view

Interfaces presented in the previous section have a limited impact on the framework, except for the `connect` operation. The framework needs to identify if a *provides* port belongs to a collection. It is our main motivation for the introduction of the `createCollectionInstance` operation in the `CollBuilderService` interface. It is also possible to define an operation to turn an existing component into a collection. Such operation can be useful to delegate the management of a set of components to the framework only once that become necessary during the execution. For simplicity reasons in the presentation of the proposed model, this operation is not presented.

Whenever the `connect` operation has identified that a *provides* port belongs to a collection, it invokes its associated `instantiatePattern` operation. Note that the framework should behave correctly even if no pattern is associated to a *provides* port of a collection. Hence, a default pattern should be implemented by the framework. It may be up to the framework to decide what is the default pattern: connection of the first/random element of the collection, round-robin or introduction of a proxy component which implements a random policy for scheduling requests to the components inside the collection, etc. Such a proxy involves the connection of multiple ports to a same type of port. Some techniques, like multi-port [22] may ease this operation.

If the port of the collection is of type *uses* (resp. *accesses*), the `connect` operation has to connect all *uses* (resp. *accesses*) ports to the external *provides* (resp. *shares*) port.

The dynamic collection management has a very low impact on the framework. The related interfaces only enable a delegation pattern.

4 Enabling data sharing on operation invocation

The previous section has dealt with the issues of sharing a data between components and of supporting the master–worker paradigm in CCA. This section is about the issue of passing a shared data as a parameter of an operation invocation. This scenario was not previously handled and typically appears when a master–worker paradigm would like to have a shared data as a parameter of an operation. This section first presents a general model and then instantiates it as an extension of the CCA model. In the remainder of the section, we consider the example based on two components: a `Client` connected to a `Server` which provides an `compute` interface. The `Client` component invokes the `inverse` operation that produces a new matrix.

4.1 Illustration of the generic model

Let us start with component models that provide an IDL language such as CCA SIDL or OMG IDL. In order to share a parameter while invoking an operation, a notation needs to be introduced in the IDL language. 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 fulfills this meaning in the C++ language. The modification of IDL languages is thus immediate and could be done with this conventional reference notation. Figure 7 shows a simple example.

Component models enforce that all incoming and outgoing communications go through some well defined ports.

```
interface compute {
   void inverse(in matrix& m1,
                out matrix& m2);
}
```

Fig. 7 Pseudo-IDL for the `inverse` operation

```
// Create a shares data port of type matrix
SharesPort* dp1 = createSharesPort("p1", matrix)
// Associate the data to the data port
dp1->associate(ptr, size);
// Create an access data port of type matrix
AccessPort* dp2 = createAccessPort("p2", matrix);
// Invoke the method
to_server->inverse(dp1, dp2);
```

Fig. 8 Steps for invoking `inverse` operation from the component `Client` using shared data as parameters

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 Sect. 2.1: for an operation invocation, a each parameter passed by reference has to be associated to a data port. 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 callees, in particular to support multiple simultaneous invocations on possibly distinct shared data. 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. If the component model does not provide any IDL language, operation prototypes are those defined hereinbefore: they contain data port parameters.

Let us now illustrate the use of this model through the previously introduced example. Figure 7 shows the operation prototype while Fig. 8 shows how the component `Client` invokes the operation `inverse`. First, a data port has to be created for each shared parameter. For the `in` parameter, a `shares` data port of type `matrix` is created and then associated to the data. This permits to make the data available from outside the component. For the `out` parameter, an *accesses* data port needs to be created so as to receive the reference of the remote shared data returned by the `inverse` operation.

The code of the callee is shown on Fig. 9. 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 parameter modes. Classically, e.g. in CCA SIDL or OMG IDL, 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

```
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);
};
```

Fig. 9 Implementation of the `inverse` operation on the `Server` component with shared data as parameters

is responsible to allocate the data. Our reference notation specifies that the data is *shared*. Hence, it is orthogonal with parameter modes. For `in` modes, the semantic is the following: the caller provides an access to an already created data. Therefore, the callee can access the data in read and write mode but without the right to reallocate it. 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 dissociated from the *shares* data port, then another data reference can be associated to the port.

4.2 Applying the generic model to CCA

To enable data sharing on operation invocations in CCA, the projection is based on the proposed extension presented in Sect. 3.2. As the used annotations for data port interfaces are similar to those used in the generic model, the projection is straightforward. With respect to the user view, the only changes are in the SIDL language, whose is enriched with the ampersand character, and in the mapping of SIDL operation with shared parameters as explained in Sect. 4.1. With respect to the implementer view, the CCA specifications are those introduced in Sect. 3.2. No specialization is needed. However, SIDL compilers like Babel [17] need to be modified to behave accordingly. All the work to manage shared data on operation invocation, like port creation and connection, is mainly done in stubs and skeletons of *provides* and *uses* ports.

5 Data sharing with a component-based ray-tracer

5.1 A ray tracing application

To illustrate the concept of data ports, we selected a parallel rendering algorithm based on ray-tracing. This algorithm follows a master–worker paradigm to distribute the computation of pixels among a set of machines and requires to

share both the geometrical database and the frame-buffer. The ray tracing algorithm is used in computer graphics to render high quality images. It is based on simple optical laws which take effects such as shading, reflection and refraction into account. It acts as a light probe, following light rays in the reverse direction. The basic operation consists in tracing a ray from an origin point towards a direction in order to evaluate a light contribution. Computing realistic images requires the evaluation of several millions light contributions to a scene described by millions of objects. Tracing a ray requires to go through an Object Acceleration Data Structure (OADS) to discover whether a ray intersects an object (such as a polygon) without testing all objects of the 3D model. Parallelization of the ray-tracing [8] is very simple if the 3D model and the OADS is replicated or shared. For a complex 3D model with several millions of objects, sharing is more suitable than replication.

5.2 Component model

We split a ray-tracing algorithm into several components. Its architecture is shown in Fig. 10 while the main interfaces are shown in Fig. 11. The OADS Builder component creates the OADS from the 3D model of the scene to be rendered. Its `OADSconfig` port enables to retrieve references to them. The master component sends rays to Ray-Tracer components through the `render` operation. The 3D model, the OADS and the frame-buffer are passed as shared parameters of the `render` operation. Ray-Tracer components read the OADS while they write in the frame-buffer. Therefore, several Ray-Tracer components can concurrently update the frame-buffer by storing pixel values. The master component manages a set of frame-buffers for 3D animation that are accessed by a specific component (Mpeg) that produces an encoded video from these frame-buffers.

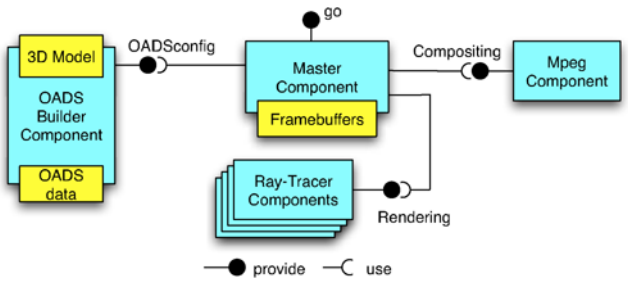

Fig. 10 A parallel component-based ray-tracer using data ports

```
interface OADSconfig extends cca.Port {
  model3D& get_model3D();
  oads& get_oads();
}
interface Rendering extends cca.Port {
   void render (in model3D& m3D, in oads& o,
                in ray_t aRay, in frameBuffer& FB);
}
```

Fig. 11 Pseudo SIDL interface examples related to ports of the ray-tracer

5.3 Discussion

Being able to actually share data like the 3D model and the OADS brings several advantages. First, the sequential code can be directly reused even though components are instantiated into distinct processes or machines. Second, it avoids to compute data distributions for the 3D model and OADS. Third, it transparently supports very large 3D models and OADS. The pressure is on the data sharing middleware, not on the components. Fourth, a modification to the 3D model, typically for animation purposes, is automatically propagated as the data sharing middleware implements the consistency model.

The ability to have shared parameters enables the decoupling of rendering operations from configuration operations: a worker component does not need to be explicitly connected to a 3D model and to OADS data. The *accesses* port creation and connection are automatically done on the operation invocation. Hence, it is very easy for a master component to simultaneously launch the computation of several frame-buffers depending or not on the same 3D model. The rendering workers act more like a service: any component having an image to render may use it.

The collection concept enables to have a master–worker relationship between only some components of the application. The handling of the collection size depending on the number of requests may be handled outside the master component. It was not represented here.

6 Conclusion

Software components is a very promising technology to build complex applications. However, current component models do not support data sharing between components and poorly support the *master–worker* paradigm. Both are important: data sharing is useful for applications dealing with complex and large pieces of data, as it eases a lot their management; the master–worker paradigm is very common. This paper presented two contributions. First, it applies our generic models of data sharing and master–worker for component models to CCA. Second, it presents a generic model allowing shared data to be used as parameters of operation invocations. Such a model is mapped to CCA and its benefits are illustrated with a ray tracing algorithm.

Future work is three-fold. First, we are implementing such an application to actually evaluate its performance. Second, we are investigating the impact of modifying a CCA framework implementation, in particular MOCCA [21], to

support the proposed *master–worker* model. And third, we are investigating the impact of such a programming model on the deployment process. Being able to describe an application independently of the resources puts the burden on the resource selection algorithm. It has to take into account both data sharing and master–worker relationships between components when selecting resources to execute the application.

References

1. Aldinucci, M., Campa, S., Coppola, M., Danelutto, M., Laforenza, D., Puppin, D., Scarponi, L., Vanneschi, M., Zoccolo, C.: Components for high performance Grid programming in the Grid.it project. In: Getov, V., Kielmann, T. (eds.) Proc. of the Workshop on Component Models and Systems for Grid Applications, Saint Malo, France, June 2004. Springer, Berlin (2005)
2. Allcock, B., Bester, J., Bresnahan, J., Chervenak, A.L., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., Tuecke, S., Foster, I.: Secure, efficient data transport and replica management for high-performance data-intensive computing. In: Proceedings of the 18th IEEE Symposium on Mass Storage Systems (MSS 2001), Large Scale Storage in the Web, Washington, DC, USA, 2001, p. 13. IEEE Computer Society, Washington (2001)
3. Anderson, D., Bowyer, S., Cobb, J., Gebye, D., Sullivan, W.T., Werthimer, D.: A new major SETI project based on Project SERENDIP data and 100,000 personal computers. Conference Paper, Astronomical and Biochemical Origins and the Search for Life in the Universe, IAU Colloquium 161, Bologna, Italy, 1997, p. 729
4. Anderson, D.P.: BOINC: a system for public-resource computing and storage. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID '04), Pittsburgh, PA, USA, November 2004, pp. 4–10. IEEE Computer Society, Washington (2004)
5. Antoniu, G., Bertier, M., Caron, E., Desprez, F., Bougé, L., Jan, M., Monnet, S., Sens, P.: GDS: an architecture proposal for a grid data-sharing service. In: Getov, V., Laforenza, D., Reinefeld, A. (eds.) Future Generation Grids, CoreGRID series, pp. 133–152. Springer, Berlin (2005)
6. Antoniu, G., Bougé, L., Jan, M.: JuxMem: an adaptive supportive platform for data sharing on the grid. Scalable Comput. Pract. Experience **6**(3), 45–55 (2005)
7. Antoniu, G., Bouziane, H.L., Breuil, L., Jan, M., Pérez, C.: Enabling transparent data sharing in component models. In: 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06), Singapore, May 2006, pp. 430–433.
8. Badouel, D., Bouatouch, K., Priol, T.: Ray tracing on distributed memory parallel computers: Strategies for distributing computation and data. IEEE Comput. Graph. Appl. **14**(4), 69–77 (1994)
9. Barroca, L., Hall, J., Hall, P.: An introduction and history of software architectures, components, and reuse. In: Software Architectures: Advances and Applications. Springer, Berlin (1999)
10. Bernholdt, D.E., Allan, B.A., Armstrong, R., Bertrand, F., Chiu, K., Dahlgren, T.L., Damevski, K., Elwasif, W.R., Epperly, T.G.W., Govindaraju, M., Katz, D.S., Kohl, J.A., Krishnan, M., Kumfert, G., Larson, J.W., Lefantzi, S., Lewis, M.J., Malony, A.D., McInnes, L.C., Nieplocha, J., Norris, B., Parker, S.G., Ray, J., Shende, S., Windus, T.L., Zhou, S.: A component architecture for high-performance scientific computing. Int. J. High Perform. Comput. Appl. **20**(2), 163–202 (2006)
11. Bouziane, H.L., Pérez, C., Priol, T.: Modeling and executing master–worker applications in component models. In: 11th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), Rhodes Island, Greece, April 2006
12. Bruneton, E., Coupaye, T., Stefani, J.B.: Recursive and dynamic software composition with sharing. In: Seventh International Workshop on Component-Oriented Programming, Malaga, Spain, June 2002
13. Buyya, R., Abramson, D., Giddy, J.: Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. High-Perform. Comput. China, IEEE CS Press, USA **1**(1), 283 (2000)
14. Caron, E., Desprez, F., Lombard, F., Nicod, J.M., Quinson, M., Suter, F.: A scalable approach to network enabled servers. In: Monien, B., Feldmann, R. (eds.) Proceedings of the 8th International EuroPar Conference, Paderborn, Germany, August 2002. Lecture Notes in Computer Science, vol. 2400, pp. 907–910. Springer, Berlin (2002)
15. Casanova, H., Dongarra, J.: NetSolve: a network-enabled server for solving computational science problems. Int. J. Supercomput. Appl. High Perform. Comput. **11**(3), 212–223 (1997)
16. Chervenak, A., Deelman, E., Foster, I., Guy, L., Hoschek, W., Iamnitchi, A., Kesselman, C., Kunszt, P., Ripeanu, M., Schwartzkopf, B., Stockinger, H., Stockinger, K., Tierney, B.: Giggle: a framework for constructing scalable replica location services. In: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02), Baltimore, Maryland, 2002, pp. 1–17. IEEE Computer Society, Washington (2002)
17. Dahlgren, T., Epperly, T., Kumfert, G., Leek, J.: Babel User's Guide. CASC, Lawrence Livermore National Laboratory, Livermore (2006), babel-1.0.2 edition
18. Germain, C., Néri, V., Fedak, G., Cappello, F.: XtremWeb: building an experimental platform for global computing. In: Grid'2000, December 2000
19. Kola, G., Livny, M.: Diskrouter: a flexible infrastructure for high performance large scale data transfers. Technical Report CS-TR-2003-1484. University of Wisconsin-Madison Computer Science Department, Madison, WI, USA (2003)
20. Magee, J., Dulay, N., Kramer, J.: A constructive development environment for parallel and distributed programs. In: Proceedings of the International Workshop on Configurable Distributed Systems, Pittsburgh, US, March 1994, pp. 4–14
21. Malawski, M., Kurzyniec, D., Sunderam, V.: Mocca—towards a distributed CCA framework for metacomputing. In: Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)—Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models—HIPS-HPGC, Denver, Colorado, USA, April 2005, pp. 907–910
22. Malawski, M., Bubak, M., Placek, M., Kurzyniec, D., Sunderamann, V.: Experiments with distributed component computing across grid boundaries. In: Proceedings of the joint Workshop on HPC Grid Programming Environments and Components and Component and Framework Technology in High-Performance and Scientific Computing (HPC-GECO/CompFrame 2006), Paris, France, 2006, pp. 109–116
23. McIlroy, M.D.: Mass produced software components. In: Naur, P., Randell, B. (eds.) Software Engineering, pp. 138–155. Scientific Affairs Division, NATO, Brussels (1969)
24. Open Management Group (OMG): CORBA components, version 3. Document formal/02-06-65, June 2002
25. Pérez, C., Priol, T., Ribes, A.: A parallel corba component model for numerical code coupling. Int. J. High Perform. Comput. Appl. **17**(4), 417–429 (2003)
26. Szyperski, C.: Component Software—Beyond Object-Oriented Programming. Addison-Wesley/ACM, Boston (1998)
27. The JXTA project, http://www.jxta.org/ (2001)
28. Lightweight Data Replicator, http://www.lsc-group.phys.uwm.edu/LDR/

Gabriel Antoniu is a research scientist at INRIA since 2002. He is working in the PARIS project-team at IRISA, Rennes. He received his Bachelor of Engineering degree from INSA Lyon in 1997; his MS degree in Computer Science from ENS Lyon in 1998; his PhD degree in Computer Science in 2001 from ENS Lyon. His research focuses on data management for parallel and distributed computing. His research topics include large-scale data sharing, peer-to-peer systems, consistency models and protocols, distributes shared memory systems.

Hinde Lilia Bouziane is a PhD student at IRISA/INRIA. Her research interests concern advanced programming models for GRIDs, involving software component and workflow models. She obtained a Master degree in computer science in September 2004 from the French Paris-South University.

Mathieu Jan is currently a postdoc researcher in the Parallel Distributed Group of the Technical University of Delft. He received a PhD degree in Computer Science in 2006 from the University of Rennes I (France). His PhD research activities were on designing a grid data-sharing service called JuxMem under the supervision of Gabriel Antoniu and within the PARIS Research Group at IRISA/INRIA Rennes. He received his Bachelor of Engineering degree from INSA Rennes and his Master degree in Computer Science from IFSIC, University of Rennes I in 2003.

Christian Pérez received a PhD degree in Computer Science in 1999 from the Ecole Normale Superieure of Lyon (France) and an "Habilitation à Diriger des Recherches" in 2006 from the University of Rennes I (France). Since 2000, he is a research scientist at INRIA. He is working in the PARIS project-team, which is located at the IRISA laboratory (Rennes, France). His research interests concern parallel and distributed computing. His research project focuses on the programming and execution models and runtime issues for applications in grid environments. His current areas of interest are software component models and automatic application deployment models for grids.

Thierry Priol is senior research scientist at INRIA/France. He is the head of the PARIS research group, a joint group with INRIA, CNRS, ENS-Cachan, University of Rennes I and INSA Rennes. His research interests aim at contributing to the programming of PC clusters and Grids for large scale numerical simulation applications. He received a PhD degree in computer science from the University of Rennes I in 1989. He also received a "Habilitation à diriger des recherches" from the same university in 1995.