Parallel Computing xxx (2010) xxx–xxx

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Extending software component models with the master–worker paradigm

Hinde Lilia Bouziane [a], Christian Pérez [b,*], Thierry Priol [c]

[a] LIP, ENS Lyon, France
[b] INRIA/LIP, ENS Lyon, France
[c] INRIA/IRISA, Campus de Beaulieu, 35042 Rennes cedex, France

A R T I C L E I N F O

Article history:
Received 5 December 2008
Received in revised form 25 September 2009
Accepted 28 December 2009
Available online xxxx

Keywords:
Software component
Master–worker
Grids
Clusters
CCM

A B S T R A C T

Recent advances in computing and networking technologies—such as multi-core processors and high bandwidth wide area networks—lead parallel infrastructures to reach a higher degree of complexity. Programmers have to face with both parallel and distributed programming paradigms when designing an application. This is especially true when dealing with e-Science applications. Moreover, as parallel processing is moving to the mainstream, it does not seem appropriate to rely on low-level solutions requiring expert knowledge. This paper studies how to combine modern programming practices such as those based on software components and one of the most important parallel programming paradigms which is the well-known master–worker paradigm. The goal is to provide a simple and resource transparent model while enabling an efficient utilization of resources. The paper proposes a generic approach to embed the master–worker paradigm into software component models and describes how this generic approach can be implemented within an existing software component model. The overall approach is validated with synthetic experiments on clusters and the Grid'5000 testbed.

1. Introduction

Computing infrastructures are reaching an unprecedented degree of complexity. First, parallel processing is coming to mainstream, because of the frequency and power consumption wall that leads to the design of multi-core processors. Second, there is a wide adoption of distributed processing technologies because of the deployment of the Internet and consequently large-scale grid and cloud infrastructures.

All of these combined together make the programming of computing infrastructures a very difficult challenge. Programmers have to face with both parallel and distributed programming paradigms when designing an application. This is especially true when dealing with e-Science applications when they are made of several software codes that are executed on various computing resources spread over the Internet within a grid or cloud-based infrastructure.

While multi-core parallel processing is entering mainstream, distributed processing requires some works, in particular to provide good programming and software engineering practices that combine in a seamless way several programming paradigms from both distributed and parallel processing worlds. However, there are several hurdles before reaching this goal. Indeed, such practices should preserve the final objective of these infrastructures: achieving high-performance. Moreover, it has to be done while providing high level programming abstractions and portability. The combination of these constraints makes the approach an intricate task.

* Corresponding author. Tel.: +33 4 37 28 76 44; fax: +33 4 72 72 80 80.
E-mail addresses: Hinde.Bouziane@inria.fr (H.L. Bouziane), Christian.Perez@inria.fr (C. Pérez), Thierry.Priol@inria.fr (T. Priol).

0167-8191/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2009.12.012

So far, software engineering approaches to design applications to be executed on PC clusters, massively-parallel systems, or even grid infrastructures have adopted a ''lowest common denominator'' strategy that is message-passing. Because of the wider availability of message-based runtime systems, such as those based on the Message-Passing Interface (MPI), it is tempted to design and implement parallel and distributed applications using a message-passing approach even though it represents a low-level programming paradigm mainly designed for parallel computing. Although it is not our purpose to criticize this approach that has been proved very effective in the past, we think that it is reaching a limit in its inability to separate functional and non-functional aspects when programming these computing infrastructures that are both parallel and distributed.

In this paper, we advocate an approach that successfully combines modern programming practices such as those based on software components and one of the most important parallel programming paradigms which is the well-known master–worker paradigm. This paper proposes a generic approach to embed the master–worker paradigm into software component models and describes how this generic approach can be implemented within an existing software component model.

The remainder of this paper is divided as follows. Section 2 presents several implementations of the master–worker paradigm in existing parallel and/or distributed programming environments. Section 3 describes several component models and their adaptation to high-performance computing. Section 4 discusses how master–worker applications can be designed and implemented with existing component models showing their actual limitations. Section 5 gives an overview of our generic model to improve the support of master–worker applications when using software components whereas Section 6 presents a projection of the proposed generic model into an existing component model: the CORBA Component Model (CCM). Section 7 evaluates the master–worker Component Model using CCM and the Grid'5000 testbed. Finally some conclusions are drawn in Section 8.

2. The master–worker paradigm and dedicated environments

Many works deal with the master–worker paradigm. With respect to distributed computing, they can be divided in two categories. On one side, some works focus on Network Enabled Servers (NES), usually based on GridRPC operations, standardized by the OGF [1]. Examples of such systems are NetSolve [2], Ninf-G [3] or DIET [4]. On the other side, the master–worker paradigm is very popular on desktop grids such as SETI@Home [5], BOINC [6] or XtremWeb [7].

The main difference between these two categories stems from the architecture design. NES assume a quite stable and dedicated set of workers and intermediate nodes so that the system can aim to compute efficient scheduling of master requests. On the contrary, Desktop grid projects rely on volunteer nodes; they are sometimes based on P2P systems. Hence, the system is highly volatile and not predictable.

However, these projects share the same goal, which is to simplify master–worker application development. They all rely on a more or less automatic management of non-functional properties such as the worker management, the request scheduling and the transport of requests between masters and workers. The desired level of transparency is provided through an API that implements the user visible part of the system: for our concerns, there are two main API; one for the master side and one for the worker side.

While these projects succeed in meeting their goals, they have some limitations. First, they provide a solution specific to the master–worker paradigm. If such a paradigm is only used within a part of an application, the overall design and implementation of the application becomes quite difficult. It is typically the case in code coupling applications, where a code may use the master–worker paradigm while the other is a parallel (MPI) code. Second, most of these environments are suited only for a specific case. For example, Netsolve is based on a central scheduler, which is quite fine for medium scale systems whereas DIET and its hierarchy of agents are more efficient on large-scale systems. P2P based systems are usually better for very large-scale systems with highly volatile nodes. Hence, the choice of a system depends on the kind of architecture, which limits the portability of the application. Providing a common API—such as the GridRPC API [1]—is a valuable step but it is not enough as executing an application also involves deploying and configuring the middleware system, which varies a lot with respect to the selected environment.

From an application point of view, the master–worker paradigm should be kept separated from its implementation and the architectural consideration: The management of the paradigm (worker management, request scheduling and transport, etc.) depends heavily on the target architecture. Hence, the challenge is to provide a simple view to the programmer while being able to handle efficiently the non-functional part of the paradigm. Moreover, the master–worker paradigm shall be available in large applications, potentially using other paradigms and involving code coupling. Many existing programming models attempt to handle such suited properties. In particular, two programming models seem interesting: *algorithmic skeletons* and *software component models*.

Algorithmic skeletons [8] have been introduced to support typical parallel composition schemes (pipe, farm, divide-and-conquer, etc.), called *skeletons*. Skeletons based programming is done by instantiating and nesting predefined skeletons. This approach is well suited to provide a high level of abstraction for parallel paradigms: it is at the responsibility of a skeleton programming framework to consider all aspects relative to parallelism exploitation (parallelism setup, scheduling management, etc.). Behavioral skeletons [9] enrich skeleton models with the automatic management of such non-functional properties. However, programming paradigms supported by such frameworks are limited and not sufficient for large applications implicating for instance code coupling.

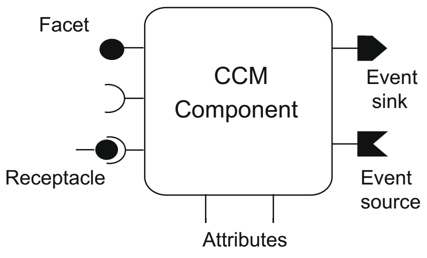


Fig. 1. A CCM component.

Component models appear to be more adequate for code coupling as they also deal with deployment issues. They are also well-known for code reuse capabilities. However, existing models do not offer a sufficient abstraction level to simplify the design of complex paradigms, as will be explained for the master–worker paradigm in Section 4.

There are some attempts to support skeletons in component models [10,11]. However, these works mainly deal with component-based implementations of skeletons for grid environments; they do not reach a sufficient abstraction level to simplify the designer view of parallel paradigms. Hence, this paper studies the possibility to support the master–worker paradigm within software component models.

3. Software component models and high-performance computing

Software component turns out to be a very efficient and promising technology for handling the increasing complexity of applications. Compared to other technologies, its main advantages are its focus on code reuse through the central notion of composition and its support for resource issues through the consideration of the deployment phase.

A software component according to Szyperski et al. [12] *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 being able to be composed with other components. This composition is possible thanks to well-defined ports that allow components to interact. A port may express the fact (1) that a component provides itself some functionality or (2) that it needs (uses) some functionality provided by another component. A component-based application is built by composing multiple components through an assembly process. Its execution can be launched after installing binary codes of all components on a given set of resources. During the deployment process, some properties associated to a component should be considered to perform a convenient resource choice. For example, such properties can be operating systems, processors and amount of memory requirements. There exist several models and tools to deploy a component-based application, such as OMG specifications related to component deployment [13], Fractal Deployment Framework [14] or Adage, an automatic and generic deployment framework [15]. However, verifying that the assembly is valid and deploying it is out of the scope of this paper.

Several component models have been proposed including industrial distributed models such as CCM [16] or SCA [17], or academic distributed models such as FRACTAL [18], GCM [19] GRID.IT [20], ICENI [21], DARWIN [22] or CCA [23]. Moreover, there exist some specialized scientific environments that have more or less similarities with component models such as Cactus [24], MCT [25], MpCCI [26], and Salome [27]—some of them being based on component models such as Salome, some are seeking interoperability such as ESMF and MCT with CCA. Component models as well as specialized scientific environments aim to facilitate the design of applications and reduce the complexity of their building process. However, to reach this goal, they should be able to support most of distributed application paradigms in an easy way, which is currently not the case. They mainly only support direct communications between components, communications that can be based on message-passing and/or method invocation.

Let introduce the CORBA Component Model (CCM) as the used model in the implementation of the present work. We have chosen this model as it is a representative software component model for which there exist mature framework implementations. In addition, tools such as ADAGE are available for deploying CCM applications on grid infrastructures.

The CORBA *Component Model* CCM [16] is part of the CORBA specifications. It enables the deployment of components to heterogeneous distributed environments. A CORBA component (Fig. 1) can define five kinds of ports. Facets (*provides* ports) and receptacles (*uses* ports) allow synchronous communications based on remote method invocations. Event sources and event sinks allow asynchronous communications based on data transfer. Attributes are values exposed through accessor (read) and mutator (write) operations intended to be used for component configuration.

CCM provides a complete model to develop a component-based application: (1) a design model to describe component types and their ports using the OMG Interface Definition Language (IDL3); (2) an assembly model to describe an application architecture (i.e component instances and their interconnections) thanks to an Architecture Description Language (ADL); (3) a packaging and deployment model to deploy an application from an assembly description[1]; (4) an execution model to

[1] Note it is the initial state of the application. CCM allows dynamic modifications of connections and addition or removal of components.

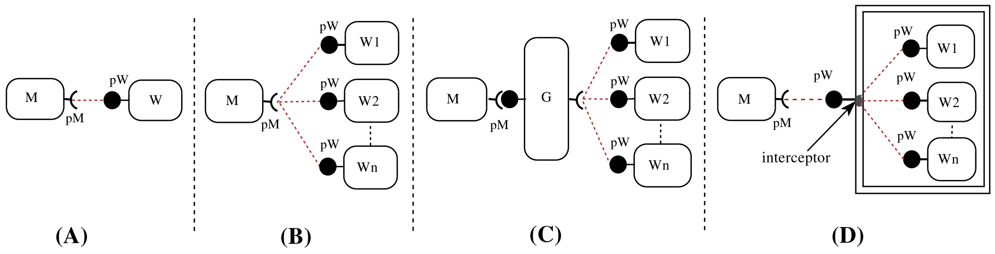


Fig. 2. Four designs of a master–worker application based on components.

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*.

4. The master–worker paradigm within existing component models

This section discusses how a master–worker application can be designed with existing component models. Fig. 2 sums up the principal designs into four kinds of composition. Let analyze them to show their limitations.

A first design, illustrated in Case **A** of Fig. 2, consists in composing a master component (M) with a unique worker component (W). The worker has to deal with all the requests sent by the master. It can be done sequentially or in parallel if W has a multithreaded implementation. This composition is well suited for multi-core machines with the issue that W has to control the number of active threads so as not to overload the machine. As of today, such an issue is implementation specific.

A second design—Case **B**—consists in connecting several worker components (W_i) to the master. Such a design can be applied for multi-core machines as well as for distributed memory machines such as clusters or grids. However, in existing component models and dedicated frameworks, this design often burdens the user with the choice of the number of workers. This choice has to be ideally made at deployment time, when the resources are known. Hence, it requires modifying the composition at deployment time, which is not a desired requirement. Moreover, the master is responsible to schedule the requests on the set of workers. The main reason is that the usage of the master–worker paradigm is not explicit through the composition. As scheduling policies are complex and depend on resources properties, their management seems to be a large burden for master developers. This easily leads to a strong dependency between a master implementation and target resources, minimizing master code reuse.

A third design—Case **C**—aims at simplifying the second design by inserting a special component (G) between the master and the workers. It relies on a separation of the master concerns from the request management concerns. Even though this design improves the code reuse of component G and can be adapted to various scenarios (by selecting an adequate G), it has some drawbacks. The worker management is still visible to the user and the worker management policy is fixed in the assembly. Hence, the assembly is still to be adapted to the actual resources.

The last approach—Case **D**—which is specific to hierarchical component models such as FRACTAL or GCM, consists in grouping all workers in a same composite and to place the management code into a controller. Controllers can act on request redirection as well as on the management on the composite content. However, the problems of choosing the right controller and of managing workers remain.

While it is possible to handle master–worker applications within current component models, the management of non-functional properties of the master–worker paradigm burdens the user as the application assembly depends on the resources. Next section deals with a generic component model being able to hide such a management.

5. A model for a component-based master–worker paradigm

In [28,29], we propose a model to improve the support of the master–worker paradigm in component models. The approach consists in increasing the abstraction level of a component assembly involving such a paradigm. The aim is to enable a designer to only deal with functional concerns when programming and to encourage the vision of automatic and transparent management of resource dependencies of the master–worker paradigm (worker management and request scheduling). The design of a master–worker application should be separated from computing resources to improve reusability and the management of resource dependencies should ensure an efficient execution on given resources. Regarding to the relevant efforts which have been already done to manage the complexity of the master–worker paradigm, our aim is also to enable the reuse of existing master–worker environments like DIET or XtremWeb.

The proposed model follows a generic approach, which can be projected on different existing component models. Fig. 3 presents an overview of the concepts of the proposed model. The user point of view of the proposal allows a designer to specify a *collection* of `Worker` components in an *abstract assembly description*. When execution resources are known, the abstract assembly is transformed to a concrete assembly. In this latter, a collection is turned into an assembly formed by internal component instances and by an instance of a *request transport pattern*. A *pattern*, as a scheme, represents an implementation

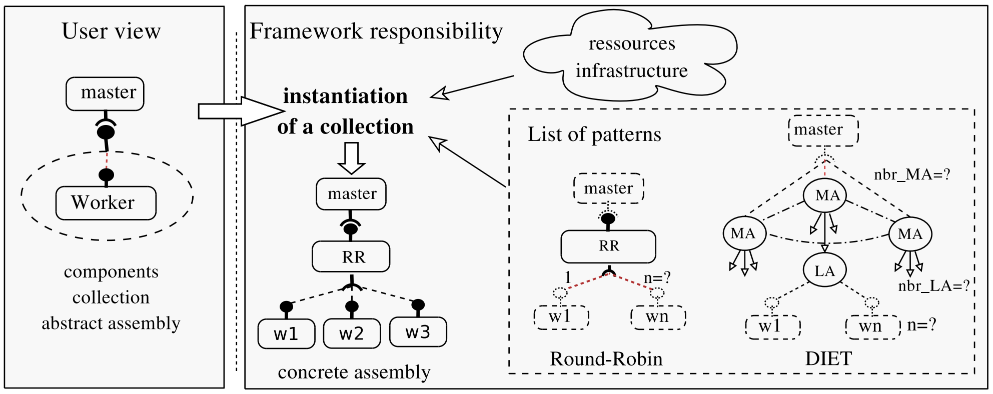
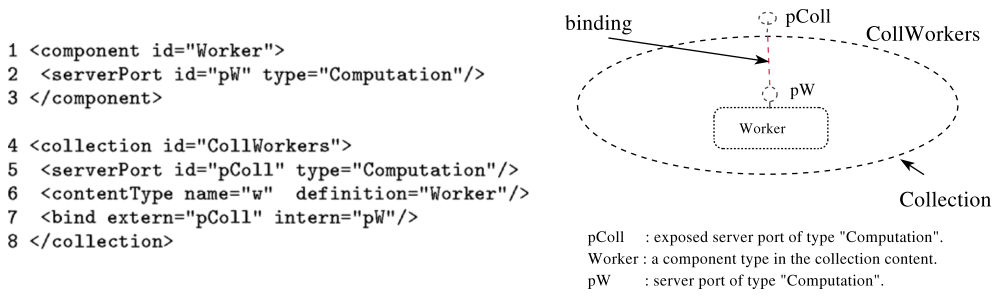


Fig. 3. Global overview of the master–worker proposed model.



```
1 <component id="Worker">
2  <serverPort id="pW" type="Computation"/>
3 </component>

4 <collection id="CollWorkers">
5  <serverPort id="pColl" type="Computation"/>
6  <contentType name="w"  definition="Worker"/>
7  <bind extern="pColl" intern="pW"/>
8 </collection>
```

binding
pColl
CollWorkers
pW
Worker
Collection
pColl : exposed server port of type "Computation".
Worker : a component type in the collection content.
pW : server port of type "Computation".

Fig. 4. Example of a the collection definition.

of a request transport policy from a *master* to *worker* components. Such an implementation should be done by experts and may be based on software components. In the example of Fig. 3, the selected pattern is a centralized round-robin scheduling policy implemented within a proxy component. The remaining of this section introduces in more detail the proposed concepts and steps of the model.

5.1. An abstract assembly model

This section describes the user view of a master–worker based component assembly according to our proposal. This view offers an abstraction level that hides non-functional aspects of the paradigm. For that, the *collection* and the *abstract assembly* are introduced.

Definition of a collection: The core of the proposal is based on the concept of *collection*. A collection is defined as a set of *exposed* ports. An exposed port can be a client or a server port, bound to some internal component type ports. A collection behaves like a component: it can be connected to other components and/or other collections.

Fig. 4 presents a collection definition using a generic *XML* syntax. The `CollWorkers` collection is defined by its exposed server port `pColl` (line 5), its content (specified by the component type `Worker` at line 6) and the binding of the `pColl` port to the server port of the component `Worker`, of the same type `Computation` (line 7). For the user, only component types inside the collection are specified: the number of `Worker` instances and the concrete representation (at execution) of the collection are hidden.

As the proposed master–worker model focuses on server ports, the remainder of this paper assumes that only server ports are exposed by a collection. Scenarios illustrating the semantic associated to a client exposed port can be found in [28].

Abstract assembly description: In the scope of the present work, an abstract assembly is an assembly allowing component and collection compositions. A collection within an assembly is viewed as a component. It is a black box with ports to be connected in a similar way as usually done for connecting component ports and with similar port compatibility constraints. The difference is that a collection is an abstract entity that cannot be directly instantiated.

Fig. 5 presents an abstract assembly example describing the architecture of a master–worker application. In this assembly, the internal description of the `workers` collection is hidden. The connection of the master component `M` to the collection (lines 8–11) expresses the fact that a master request has to be resolved by one element of the collection. At this step of the application design, the internal description of the `workers` collection—including request scheduling—remains hidden to the user. Therefore, a collection is an abstract concept. That explains the abstract specificity of the assembly, while the composition principle itself is kept unchanged.

```
<componentType id = "Master">
 <clientPort id = "pM" type = "Computation"/>
</componenttype>

<!-- the collection definition (see Figure 4) -->

<composition>
 <component id = "M" definition = "Master"/>
 <collection id = "workers" definition = "CollWorkers"/>
 <connection>
  <clientPort compRef ="M" portName = "pM"  />
  <serverPort collRef ="workers" portName ="pColl"/>
 </connection>
</composition>
```

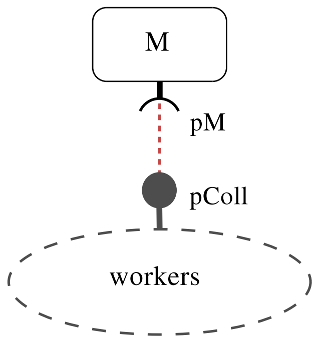


Fig. 5. Example of an abstract assembly.

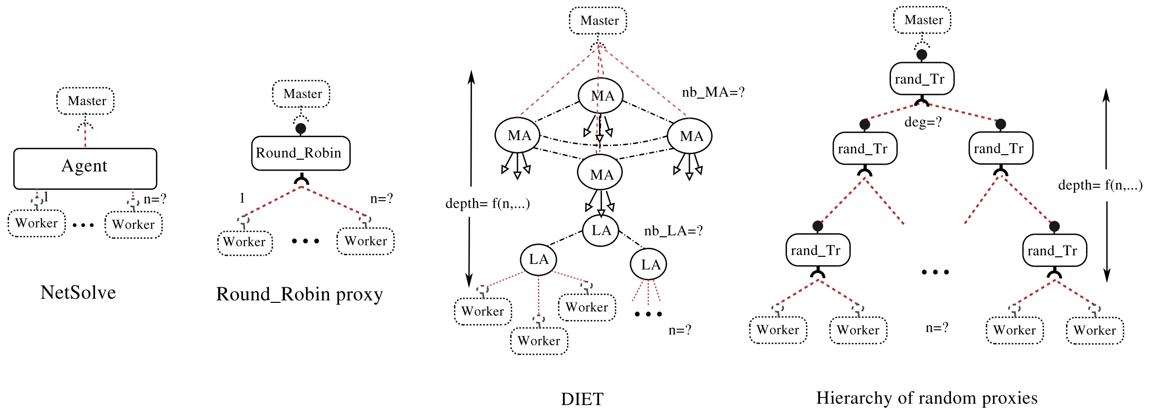


Fig. 6. Examples of request transport patterns.

5.2. Conversion from an abstract to a concrete assembly

To be deployed, an abstract assembly needs to be converted into a concrete one. That means all collections specified in an assembly need to be transformed to obtain a deployable assembly. A collection transformation consists in specifying an initial number of worker components to be created, and in introducing a request transport mechanism between master and worker components. As shown in Fig. 3, the transformation process is expected to be done automatically by a framework. To help the automation of this process, we make use of the concepts of patterns, concrete assembly and the collection transformation step. This section presents these concepts and their objectives.

Request transport pattern: A pattern describes the implementation architecture of a request transport mechanism. This architecture can be based on components or other entities like distributed objects or processes. Thus, it is conceivable to (re-)implement simple patterns using component technology as well as to reuse complex existing environments like DIET (based on distributed objects). A pattern also describes how master and worker components must be bound to this architecture. Fig. 6 illustrates examples of patterns based on different technologies. Let us describe them in a general way with respect to their architecture and their bindings with master and worker components.

A pattern's architecture is a parametric architecture. The parameters of a pattern allow it to be adapted to given execution resources. They are mainly used to specify the number of initial elements (components, objects, etc.) and bindings to be created by the pattern's usage. This number may depend on the number of workers (parameter *n* shown in Fig. 6). Other parameters, like the depth and/or the tree's degree for hierarchical patterns, may also depend on the architecture of execution resources. For example, the DIET [30] pattern is more appropriate for grid resources that have a hierarchical architecture.

A binding between a pattern and master or worker components defines a communication mechanism between the pattern and an application. This mechanism may differ depending essentially on the programming model used to implement a pattern:

- In the case of a component-based pattern, such a mechanism can be a classical connection of ports. Examples of patterns adopting such a solution are the round-robin proxy and the hierarchy of random proxies patterns illustrated in Fig. 6. Such bindings ensure the application and the pattern codes to be independent. This independence represents a relevant

```
//request submission for the operation
//  void compute(long par1, long par2)

diet_profile_t *profile;

//preparing a request for compute(par1,par2)
profile = diet_profile_alloc("compute", ...);
diet_scalar_set(..profile.., &par1,...);
diet_scalar_set(..profile.., &par2,...);

//submitting the request
diet_call(profile);
```

Fig. 7. Overview of using the DIET for request submission.

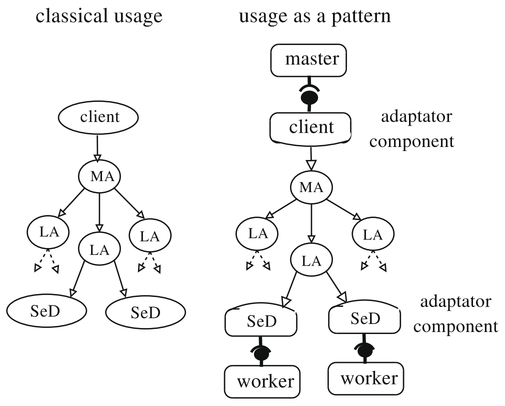


Fig. 8. Binding a master–worker application to the DIET pattern: adaptor components.

advantage as the abstract view of the proposed model can be kept during the execution: a user should be able to modify the application architecture by adding/removing or connecting/disconnecting components without dealing with possible impact on pattern architecture.

- In the case of non-component-based patterns, especially aimed for reusing existing master–worker environments, communications inside the pattern are not done through ports and a specific API is used to submit a request. Therefore, a solution for implementing a binding has to be proposed. Without loss of generality, we illustrate the adopted solution through the DIET pattern. In DIET,[2] a client defines a *profile* that deals with request parameters. Then, requests are submitted through a DIET call with the profile as parameter. Fig. 7 sums up these steps for the *compute* operation example. Conversely, on the server side, requests are extracted from a profile that is passed as parameter to a server implementation. Thus, a solution to use DIET as a pattern is to translate a call on the master's port to a DIET request—as done in Fig. 7—and to translate a server side request to a call on the worker's port. For that, an approach is to define two *adaptor* components: a master side component and a worker side component. As shown in Fig. 8, adaptor components act as a bridge between component's ports and DIET. This solution preserves the advantages of component-based patterns.

Finally, the approach to bind a pattern to the master and worker components has the benefit of keeping a port-based connection between user components and patterns. However, this solution requires port compatibility that leads adaptor components to rely on application specific definitions. At the same time, the implemented request scheduling and transfer policies within a pattern are generic. Thus, when using a pattern, its implementation can be automatically generated from the definition of a collection. That should promote reusability of patterns. This paper does not address automatic code generation[3]: we have manually implemented component-based patterns and adaptor components for specific applications. Fig. 9 shows the principle of the adopted approach through the usage of a round-robin pattern.

Concrete assembly description: A concrete assembly is an instance of an abstract assembly to be deployed. For each described collection, this instance determines an initial number of workers and replaces the external connections and internal bindings to exposed server ports by instances of request transport patterns. Fig. 3 illustrates a concrete assembly example. The `workers` collection of the abstract assembly is replaced by three worker components and one instance of the round-robin proxy pattern that links the connected master to these workers. Therefore, the application architecture at execution is composed of both functional components (master and workers) and non-functional elements (the elements of the pattern).

[2] As we started this work with an old version of DIET, we used the DIET API. Last versions of DIET support the GridRPC API.
[3] More recently, generic components have been proposed [31] that simplifies the problem.

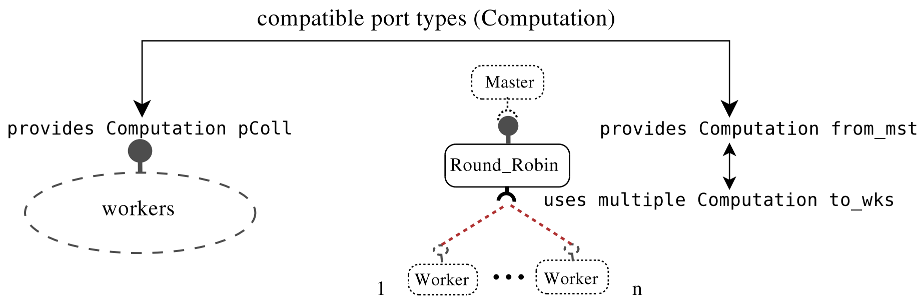

compatible port types (Computation)
provides Computation pColl
workers
Master
Round_Robin
provides Computation from_mst
uses multiple Computation to_wks
Worker
Worker
1
n

Fig. 9. Designing a round-robin proxy pattern for a particular application.

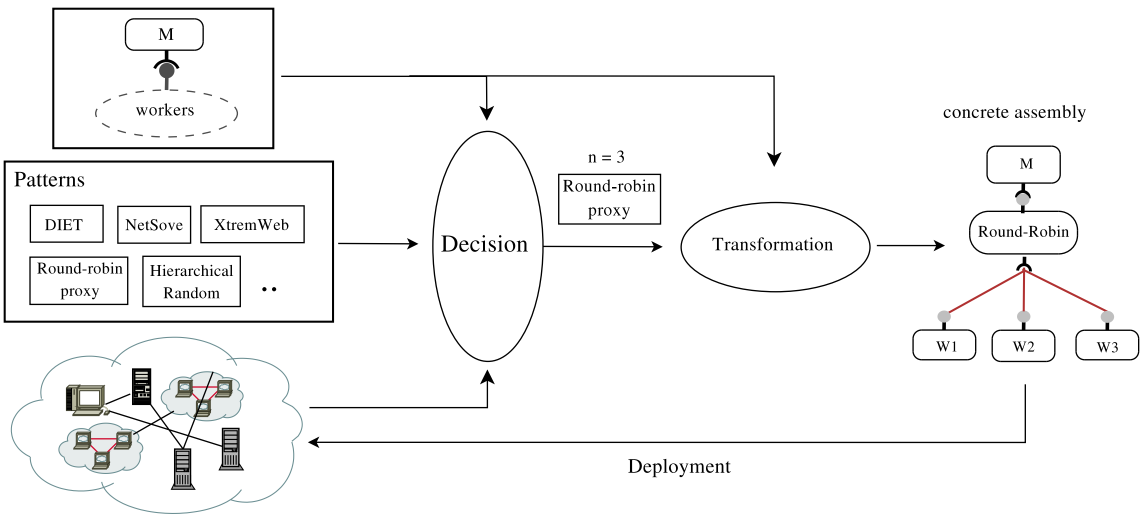


Fig. 10. Overview of collection transformation principles.

However, for patterns like DIET or *NetSolve* (Fig. 6), there is the issue of supporting both components and other technologies within a concrete assembly. The classical solution consisting in forcing one technology within an assembly has the drawback of not supporting (distributed) legacy codes. To deal with such a drawback, some advanced deployment tools such as ADAGE [15] that supports multi-technology applications have been proposed. This paper relies on the existence of such tools and does not study a generic specification of a heterogeneous assembly model. We assume then the existence of such a model when reusing an existing master–worker environment.

Collection transformation: Collection transformation is assumed to be under the responsibility of the framework. Depending on the adopted component model and its implementation, this latter can be a deployment tool (for CCM) or an execution framework (for CCA). In fact, such frameworks are already responsible of the placement of components on execution resources. Thus, they appear appropriate to be also responsible for managing resource dependencies of the master–worker paradigm. In the scope of this paper, our interest is not to present a framework implementation. Hence, we present only its expected role. To specify this role, we identified two steps: the decision step and the transformation step (see Fig. 10). Let us briefly detail them.

The main role of the decision step is to determine the number of workers and an adequate request transport pattern. Several criteria may be considered in this step. First, determining the number of workers mainly depends on the number of resources and their availability. It may also depend on meta-data providing information about the number of incoming requests, their estimated duration and/or size, etc. Second, choosing a pattern is done among a set of patterns provided to the framework. This choice must consider the suitability of a pattern with respect to the actual execution resources (targeted architecture, ability to manage heterogeneity of resources), the number of workers to be managed (scalability of the pattern), the number and the type of a master requests (suitable scheduling policy for homogeneous or heterogeneous requests), etc. At the end, the decision step is the most relevant and complex one within the proposed model as it should be able to adapt an abstract assembly to resources and at the same time to target performance of execution. The proposed model assumes that decision is under the responsibility of an adaptability framework [32,33], which in particular involves the pattern behavior.

The transformation step converts an abstract assembly to a concrete one. It is the execution of a process applying the results of the decision step. As illustrated through an example in Fig. 10, the inputs of this step are the abstract assembly to be

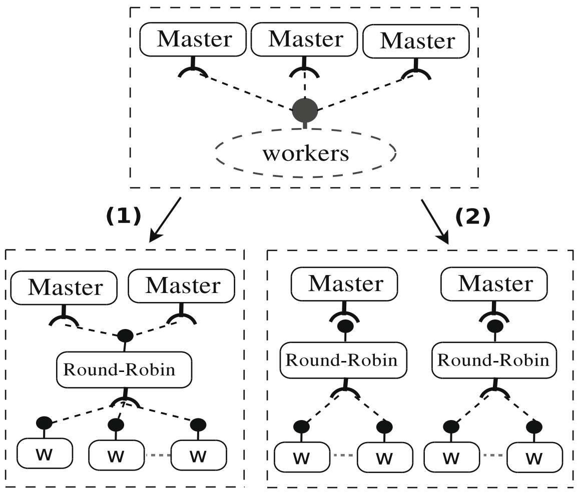


Fig. 11. Two multi-master scenarios.

```
// classical definitions
interface Computation{
  void compute (long par1, long par2);
};

component Master{
  uses Computation pM;
};

component Worker {
  provides Computation pW;
};

// extension

collection CollWorkers {
  provides Computation pColl;
};
```

```
<collection type="CollWorkers">
 <componentfiles>
  <componentfile id="Worker">
   <fileinarchive name="defWorker.csd"/>
  </componentfile>
 </componentfiles>
 <bindings>
  <binding>
   <providesCollport>pColl</providesCollport>
   <provideselementports>
    <providesport>
     <providesporti>pW</providesport>
     <componentref idref="Worker"/>
    </providesport>
   </provideselementports>
  </binding>
 </bindings>
</collection>
```

Fig. 12. A collection description in ECCM. On the left, `CollWorkers` is a collection type specified in Extended-IDL3 (EIDL3). On the right, the collection's content is described in CDL.

instantiated and the results of the decision step (three workers and the round-robin proxy pattern). The produced assembly is the actual application architecture to be deployed.

Collection instantiation in the case of multiple masters: Until now, we have assumed a unique master connected to a collection. However, several masters can be connected to a collection. Such a situation typically occurs when the master is a parallel code where each process acts independently as, for example, in multilevel parallelism computational chemistry applications [34].

A question that arises is to determine the impact on the pattern when connecting several masters to a collection. It is the implementation of the pattern selected during the collection instantiation, which determines how the pattern behaves when there are several masters. For the round-robin pattern for instance, a first solution is to connect all masters to one round-robin component instance as illustrated in the first instantiation of Fig. 11. A more advanced solution may decide to instantiate pattern elements depending on the number of masters as shown in the second instantiation in Fig. 11.

Splitting the workers into subgroups may lead to a load balancing issue with respect to the number and the load of each master requests. Hence, there is an additional criterion to decide which pattern implementation is best suited and how it should be configured: the support (or not) of several masters. The proposed model assumes that such a criterion is to be considered during the collection transformation. This can be done by an adaptability framework [32,33], which in particular involves the pattern behavior.

6. Projection of the generic master–worker model on CCM

This section describes a projection of the generic model on top of CCM as well as an example. The projection is based on extending CCM specification. The extension deals with the concepts of collection, abstract assembly, pattern and concrete assembly. Let named Extended-CCM (ECCM) the master–worker extension of CCM.

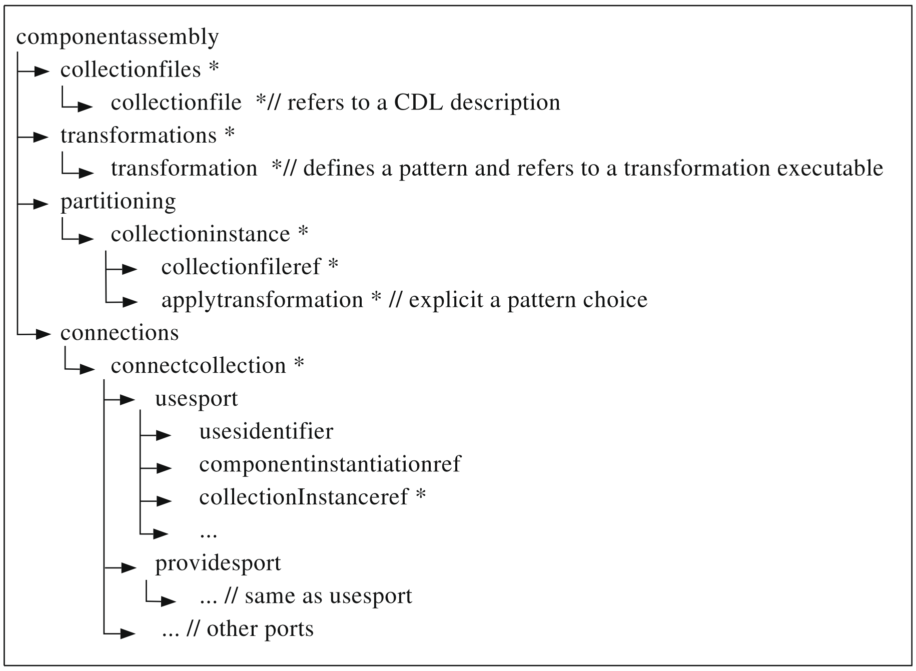


Fig. 13. DTD extension of the CCM ADL to obtain an abstract ADL. Stars indicate added elements.

6.1. A collection description

To describe a collection in Extended-CCM, we proceed in two steps. First, a collection needs to define its ports, as it is a unit of composition similar to a component. For that, we extended the Interface Definition Language of CCM (IDL3) with the keyword `collection`. This keyword allows the definition of a collection type and its exposed ports. Like a component, a collection can define several kinds of ports: *provides*/*uses*, *emits*/*consumes* (for event communications), etc. The left part of Fig. 12 shows a definition example for the collection type `CollWorkers`. It specifies a *provides* port `pColl` of type `Computation`. No further change is needed in the IDL3.

Second, to describe the content of a collection, we defined a language called *Collection Description Language* (CDL) that describes the implementation of a collection. It has the same role than the *Component Software Descriptor* (CSD) but for a collection. As a collection implementation can be seen as an abstract assembly, CDL is based on a (*XML*) syntax close to the CCM assembly language. An example that describes the `CollWorkers` type is shown on the right of Fig. 12. Component types within a collection are specified thanks to the reused `componentfiles` element (lines 2–6). Each type is referred in a standard CCM way, i.e. by a descriptor (*Component Software Descriptor*, `.csd`). The bindings of the content ports with exposed ports of the collection are described with a new *XML* element `bindings` (lines 7–17).

6.2. An abstract assembly language

To obtain an abstract assembly we extended the CCM *Component Assembly Description* language[4] (referred in this paper by the term CCM ADL). This language is based on an *XML* syntax and contains 14 new *XML* elements. The main elements are shown in Fig. 13. Among them, six elements are used to import a collection type definition (`collectionFiles`), to define a collection instance (`collectionInstance`) and to connect its exposed ports (`connectCollection`).

The other elements are dedicated to an explicit choice of patterns and their integration. As they are optional, they offer the possibility to have different levels of control of an application at its design.

As the CCM ADL is already specified to be extensible, its extension was easily feasible. We were able to reuse existing concepts that allow the extension to be minimized. Moreover, the composition principle when describing an assembly is kept unchanged when using the resulted language. Fig. 14 illustrates this principle through an example of a master–worker application assembly.

[4] We extended the CCM specification version 3.x, supported by used tools for experiments. Without any novelty in our contribution, our work can be applied to the last version (4.x).

```
<componentassembly id="ABSTRACTASSEMBLY">
       <!- Component/collection types ->
 <componentfiles>
  <componentfile id="Master">
   <fileinarchive name="master.csd"/>
  </componentfile>
 </componentfiles>
 <collectionfiles>
  <!- Import of the CDL description shown in Figure12->
   <collectionfile id="CollWorkers">
   <fileinarchive name="collworkers.cdf"/>
  </collectionfile>
 </collectionfiles>
       <!- Component/collection instances ->
 <partitioning>
  <homeplacement cardinality="1" id="MasterHome">
   <componentfileref idref="Master"/>
   <componentinstantiation id="M"/>
  </homeplacement>
  <collectionInstance id="workers">
   <collectionfileref idref="CollWorkers"/>
  </collectionInstance>
 </partitioning>
       <!- Connections ->
 <connections>
  <connectcollection>
   <usesport>
    <usesidentifier>pM</usesidentifier>
    <componentinstantiationref idref="M"/>
   </usesport>
   <providesport>
    <providesidentifier>pColl</providesidentifier>
    <collectionInstanceref idref="workers"/>
   </providesport>
  </connectcollection>
 </connections>
</componentassembly>
```

Fig. 14. Example of an abstract assembly using ECCM ADL.

6.3. Designing a request transport pattern

Section 5.2 described an approach to design a request transport pattern for component-based applications; a solution is proposed to design component and non-component-based patterns. Through the usage of extended specification of CCM, our aim in this paper is to demonstrate the feasibility of the proposal and the necessity of having several patterns for a same application. For that, we designed and implemented a component-based pattern with two centralized request scheduling algorithms (round-robin and load balancing) and also a pattern that makes use of DIET. The design and the usage of the patterns follow the principle described in Section 5.2. Their implementation is then straightforward.

The component-based pattern is composed of a proxy component. For the round-robin algorithm, when the proxy receives the *i*th request through its *provides* port, it forwards it through its *i modulo n uses* port, where *n* is the cardinality of the *uses* port. For the load balancing algorithm, when the proxy receives a request, it forwards it to the first found non-busy worker.

With respect to the DIET pattern, it is possible to integrate several scheduling policies. Hence, we introduced a request sequencing scheduling algorithm. This algorithm consists in grouping a series of requests and then in scheduling these requests according to a defined priority. Based on meta-data about the execution time of each request, a higher scheduling priority is given to longuer ones.

In Section 7, we argue the importance of varying patterns for a same application. That is done through experiments using designed patterns.

6.4. A language for a concrete assembly

Depending on the programming model used to design a pattern, a concrete assembly can be a classical CCM assembly or a heterogeneous one. The issue when using a pattern like DIET is to be able to describe its architecture (based on CORBA objects) within an assembly (Section 5.2). That is why we proposed to extend the *DTD* of the CCM ADL. This extension allows an assembly to include DIET agent compositions. An assembly example can be found in [29, p. 112]. To deploy such assembly, we have used the ADAGE deployment tool which is able to handle multi-technology application description. We adopted this solution for doing the experiments presented in Section 7.

Table 1
Round-trip time in microsecond on a local Ethernet network.

	Mean RTT (μs)	Min RTT (μs)	Max RTT (μs)
MPI-RR	95.1	89	1200
C-RR-UseMult	92.6	87	1248
C-RR-ProcColloc	99.2	92	500
C-RR-HostColloc	119.2	116	168
C-RR-NoColloc	217.2	194	519

6.5. Discussion

The presented projection of the master–worker abstract model on CCM followed an extension approach. The *collection* concept is introduced in the IDL3 language and the CCM ADL. This introduction offered the suited abstraction level of the master–worker paradigm. It has not required relevant modifications of the initial CCM specification and no framework modifications were done. In addition, we preserved the composition principle of CCM.

For the conversion from an abstract assembly to a concrete one, we presented an approach to support both component and not component-based request scheduling *patterns*, aiming to enable the reuse of DIET environment.

In a similar way, we proposed a projection on other component models: CCA [35], FRACTAL [28] and GCM [11]. The difference is that the particularity of each component model is taken into account like the hierarchy of FRACTAL/GCM and the absence of an assembly language in CCA. However, these particularities have not changed the whole principle of the abstraction approach nor required more efforts.

7. Evaluation of the master–worker component model on ECCM

The proposed model aims at increasing the level of abstraction given to a programmer while enabling high-performance. Hence, this section evaluates the main advantages that can be obtained with the proposal in various situations.

In particular, we evaluate two properties:

- Execution *performance*.
- *Design properties* offered by the proposal in terms of: (1) design *simplicity* compared with the usage of a classical component-based framework. This simplicity is evaluated by both the size of an assembly description and the nature of implicated concerns (functional and non-functional ones) and (2) *reuse* capability of a same application assembly in different execution contexts. For most of the experiments of this section, this assembly is the abstract one shown in Fig. 14.

The evaluation has been done on a synthetic master–worker benchmark that simulates the behavior of a parametric application. The parameters are the number of master requests, the size of their input data and the request duration. This duration may vary from a request to another.

All experiments have been done on the Grid'5000 experimental platform [36]. They have been deployed and executed on at most seven sites made of 64 bit AMD Opteron clusters, either with bi-processor nodes or with dual-core bi-processor nodes. All clusters are built around 1 Gbit/s Ethernet switches and are interconnected with a 10 Gbit/s backbone with latencies around 10 ms.

All codes are written in `C++`. CCM component glue codes have been generated with a home made prototype of a CCM compiler. The underlying CORBA implementation is omniORB 4.1.0. The transformation from abstract to concrete assembly has been done through *ad hoc* scripts. Then, the deployment of concrete assemblies was done with ADAGE [15].

7.1. Cluster experiments

The first question is to evaluate the proposed model compared to a high performance dedicated technology. For this comparison, we have chosen the well-known reference MPI. Hence, we implemented an MPI version of the benchmark and compared its performance, design simplicity and code reuse with those obtained with a component-based implementation.

7.1.1. Performance evaluation

The objective of performance evaluation is to measure the overhead of the proposed model. Experiments have been done on the `paravent` cluster of Grid'5000. It is made of 99 dual AMD Opteron 246 processors at 2 GHz with 2 GB of memory connected to a Gigabit Ethernet switch. The MPI implementation was MPICH2.

Round-trip and bandwidth: Table 1 and Fig. 15 report the round trip time and the bandwidth for MPI and several component-based implementations: a direct master connection to workers (*C-RR-UseMult*) and three proxy based patterns which vary in the localization of the proxy with respect to the master—same process (*C-RR-ProcColloc*), same machine (*C-RR-HostColloc*) or distinct machines (*C-RR-NoColloc*). Several conclusions can be drawn. First, though CORBA does not target fine grain computations, it can compete with MPI as we have already shown in [37]. However, on high-performance networks, the

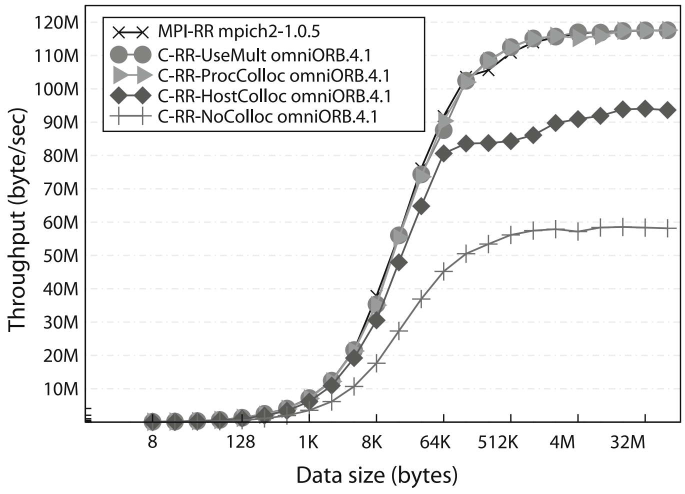


Fig. 15. Throughput on a local Ethernet network.

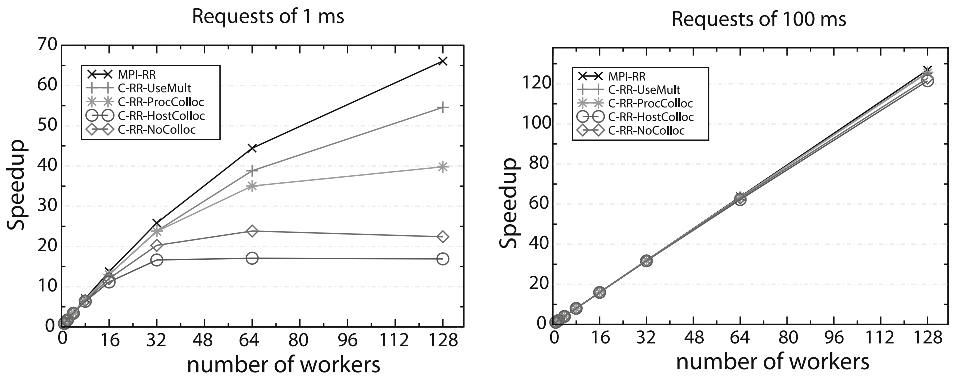


Fig. 16. Scalability experiments for sending 128 simultaneous empty requests of 1 ms (left) and 100 ms (right) duration with the RR scheduling algorithm.

latency of CORBA is a little higher than the one of an optimized MPI [37]. Second, without applying any intra-process optimization, the cost of a proxy is very acceptable whether it is collocated within the same process as the master. It only adds 5 µs to the RTT and it has no impact on the bandwidth. Without any surprise, the performance decreases if inter-process or inter-machine communications are used. If the proxy is on a third machine, the bandwidth for a request is logically divided by two.

Scalability with and without data: The second set of experiments concerns the scalability with respect to the number of workers. A first experiment, whose results are displayed in Fig. 16, considers simultaneous requests with a negligible amount of data for the five configurations described above. Without any surprise, the speedup depends on the ratio between the time to send the requests by the master and the request computation time. MPICH2 shows better ability to do it efficiently for very short requests: it is able to send 128 requests in 850 µs while it takes 1255 µs for *C-RR-UseMult* and 2121 µs for *C-RR-ProcColloc*. It represents 6.7 µs/ request for MPI and 16.5 µs/request for *C-RR-ProcColloc*. Hence, the standard CORBA mechanism is not very efficient for very fine grain computations. However, for medium grained request (100 ms), overheads are negligible and all configurations perform similarly.

A second experiment, shown in Fig. 17, measures the scalability with respect to the request size, i.e. the size of request parameters. The experiments have been done with several data sizes and with the *C-RR-ProcColloc* pattern. As expected, the speedup is limited by the number of requests that can be sent by the master. Hence, the available bandwidth is the limiting factor.

7.1.2. Design properties evaluation

To evaluate the design simplicity and reuse for the implementations listed in Table 1, a comparison is done between: (1) CCM and ECCM implementations and (2) MPI and ECCM implementations.

First, the ECCM assembly language allows the round-robin pattern and deployment constraints (components collocation) to be hidden for *C-RR-ProcColloc*, *C-RR-HostColloc* and *C-RR-NoColloc* implementations (Fig. 14). It is the transformation process that generates the three different concrete CCM assemblies for these implementations, increasing then abstract assembly

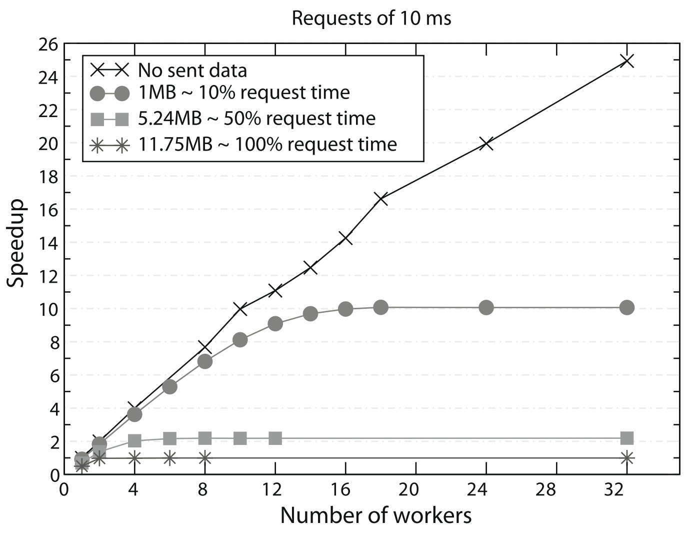


Fig. 17. Scalability experiment for sending requests with varying their parameter size. In the legend, *x*MB ~ *y*% means: the data transfer time for a parameter of size *x*MB represents *y*% of the request computation duration.

reuse. Therefore, the usage of pure CCM for our master–worker application is equivalent to describe directly corresponding concrete assemblies. Certainly, the description of these particular assemblies does not need relevant additional efforts to be done and is still simple. The main reasons are: (1) the pattern structure is simple, (2) compared to the *C-RR-UseMult* implementation, the master developer may reuse an existing code for scheduling, (3) the cluster is homogeneous, which do not need deployment constraints to be applied on worker components, and (4) the sole present `master-proxy` collocation constraint does not require relevant assembly modifications. In such a context, it is also possible to describe an undetermined number of worker components using the CCM assembly language. A result is that the size of concrete assemblies description is about 60 lines against about 40 lines for an abstract assembly. However, the relevance of having an abstract assembly is still present, as the proposal is dedicated for large execution contexts.

Second, the development process using MPI or component-based technologies is different. Then, it appears no significant to compare them according to the quantity of written codes and descriptions. However, it is clear that programming with MPI is a low-level approach. For a master–worker application, this technology requires a developer to consider several concerns within a same code: master, worker and scheduling. Compared with our proposal, this may easily lead to relevant efforts. In addition, MPI does not offer the facilities to reuse codes, written for example in different languages, or to adapt an application to different execution contexts. It is for instance not the most suitable technology to use on grid infrastructures. Therefore, even if for high-performance executions it represents a well-known reference, it is not the case for increasing code reuse and design simplicity.

To summarize, cluster experiments present a situation where the abstraction does not supply a significant improvement in design simplicity. However, the abstraction exempts a user from writing several concrete assemblies depending on used pattern and deployment constraints. Moreover, experiments show that increasing the abstraction level can be done while preserving performance.

7.2. Grid experiments

The objective of this section is twofold. The first goal is to demonstrate the need of several patterns to adapt an application to different execution contexts. The second goal is to illustrate situations where patterns are too complex to be considered by a regular designer. For that, a set of experiments is done with various patterns: a centralized proxy component encapsulating a round-robin or load balancing scheduling policy and a DIET-based pattern integrating a request sequencing scheduling policy. These patterns are those presented in Section 6.3. Experiments were done using several sites of Grid'5000.

7.2.1. Performance evaluation

This part is devoted to performance evaluation in various situations. Before exposing the experiments, the overhead of embedding DIET in a pattern is presented.

Overhead of reusing an existing grid environment as a pattern: Though it is possible to use any environment based on any kind of middleware, we have selected DIET: being based on CORBA, the analysis of the experimental results are easier. As discussed in Sections 5.2 and 6.3, DIET can be turned into a pattern thanks to two adaptor component types.

When using DIET as a pattern, we measured a global overhead of about 10 μs with respect to plain DIET if each of the two adapters (see Fig. 8) is collocated within the same process as the master and the worker. It conforms to the cluster

Table 2
Time to send one request in millisecond. The master and the MA are at Rennes. For the last column, the selected worker is at Sophia.

	One worker @Rennes	One worker @Sophia	Thirty two workers @Rennes	Thirty two workers on six sites
DIET	1.101	77.166	6.202	120.055
C-DIET	1.137	77.189	6.269	120.301

Table 3
Time and efficiency for executing 4096 homogeneous requests of 20 s each and of 4096 heterogeneous requests, whose computation load is uniformly distributed between 5 s and 20 s.

	Homogeneous requests		Heterogeneous requests	
	Time (s)	Efficiency (%)	Time (s)	Efficiency (%)
512 Workers				
C-RR	160.18	99.89	135.61	73.60
C-LB	160.18	99.89	111.90	89.20
1024 Workers				
C-RR	80.11	99.87	72.75	68.61
C-LB	80.12	99.85	62.28	80.13

experiments: the invocation between two components collocated into the same process takes 5 μs. This overhead is usually hidden by the complexity of the DIET protocol to schedule a request. There is no impact on the bandwidth: it is the bandwidth of DIET, which is quite low (about 50 MB/s) as DIET is making data copies.

Table 2 analyzes the overhead in case of one or several workers. As DIET is searching for the best place to schedule requests, the overhead varies depending on the number of workers. Fortunately, the impact of using DIET as a pattern is acceptable.

Need of several patterns: One motivation of the proposed model is to be easily able to select the most adequate pattern to be able to run the *same* application in very distinct situations. Our goal is not to determine the best pattern for every possible situations but to show that there is a clear need of several patterns and that they may become complex enough to let them be implemented by experts. Hence, the model enables to simplify the choice of a pattern to a configuration of the collection. However, the actual selection of the best pattern for a particular situation is out of the scope of this paper.

A first experiment considers a set of simultaneous requests on 512 and 1024 workers distributed over seven heterogeneous clusters. Table 3 shows the performance of round-robin and load balancing patterns with homogeneous and heterogeneous sets of requests. While the two patterns behave very well for homogeneous requests, for heterogeneous requests, the load balancing pattern is clearly better. However, the two patterns point out a decreasing performance when the number of workers become significant (>1000). This decrease is due to the occurrence of bottleneck on the proxy component.

A second experiment considers a set of simultaneous requests on 128 workers distributed over one and different clusters. The top part of Table 4 illustrates the results obtained for a particular case of heterogeneous requests: a lot of small requests followed by a long one. This situation shows that the load balancing pattern does not always perform well and more advanced algorithms are needed. In this particular experiment, the request-sequencing feature of DIET is making the difference. However, the bottom part of Table 4 shows an opposite behavior for an execution on a cluster and homogeneous requests. This is because DIET environment is more suitable for heterogeneous grid infrastructures and higher grain computations, due to its proper request scheduling costs (Table 2).

Multi-master case: Table 5 reports results obtained with multi-master scenarios. It shows that a centralized proxy is able to deal with several masters as long as the requests have a negligible amount of data. Otherwise, the input/output capabilities of the proxy limit the scalability. Fortunately, by taking a multi-master aware pattern it is possible to improve the performance. This experiment makes use of a simple multi-master pattern: the workers are divided into groups, each group being managed by a `LB` proxy localized in the same process than the master it is connected to. For simple and static configurations, such a pattern performs well. However, it is clear that more advanced patterns are needed to adapt the number of workers to the number of remaining requests or to the master's priority. To handle such situations, a solution is to add an adaptability support to the master–worker component model [32,33].

7.2.2. Design properties evaluation

This section evaluates simplicity and reuse capabilities offered by ECCM with respect to the implementations *C-RR* and *C-DIET*. We recall that the abstract assembly bound to these implementations is still the one shown in Fig. 14, modulo the addition of other master components for the multi-master scenario. In the following, a comparison is done with respect to a CCM implementation.

First, the concrete assembly associated to the *C-RR* implementation is generated in a similar way as done for cluster experiments (see Section 7.1.2). In particular, the assembly is the same for one master. Therefore, the design with both CCM and ECCM remains simple. This simplicity is similar for multiple masters and a centralized proxy. However, when the pattern becomes distributed, multiple proxies have to be specified and workers have to be split within the assembly. The result

Table 4
Comparison of RR/LB and *C-DIET* patterns with 128 workers.

	513 requests: 512 requests of 20 ms + 1 request of 100 ms	
	Time (s)	Efficiency (%)
Grid level experiments		
C-LB	180.01	44.9
C-DIET	100.82	80.1

	128 requests of 100 ms		256 requests of 500 ms	
	Time (ms)	Efficiency (%)	Time (ms)	Efficiency (%)
Cluster level experiments				
C-RR	102	98.0	1004	99.7
C-DIET	644	15.5	1737	57.6

Table 5
Multi-master aware or unaware patterns to execute 1000 simultaneous requests per master of 500 ms each. The request transport policy is LB. Masters are at Rennes and the 200 workers are evenly distributed on four other Grid'5000 sites. The time to send 5 MB is 10% of the computation time.

#Master	Data size (MB)	#Proxy	Global speedup	Master speedup
1	0	1	192.10	192.1
1	5	1	9.99	10.0
4	0	1	191.52	48.1
4	0	4	187.85	48.5
4	5	1	8.71	2.2
4	5	4	39.16	9.9

is a longer concrete assembly description. For the particular designed application, the size order of such an assembly is $30 + 35 * \#masters$ lines against 40 lines for the abstract assembly. For example, it represents 170 lines for four masters. Hence, it is preferable to simply build a unique abstract assembly and let the framework generate concrete ones.

Second, a concrete assembly associated to *C-DIET* is more complex. This complexity is due to: (1) the structure of DIET and configuration requirements of its elements (*MAs*, *LAs* and adaptor components), (2) the need of splitting the workers over *LAs*, and (3) the heterogeneity of the concrete assembly (Section 6.4). For comparison purpose, such a concrete assembly is equivalent to a CCM assembly, where DIET elements are assumed to be wrapped in CCM components. For our experiments, with one *MA* and one hierarchy level of *LAs*, the size order of a concrete assembly is $70 + 40 * \#LAs$ against the 40 lines of the abstract one. For example, it represents 230 lines for four *LAs*. The assembly description requires also writing DIET specific configuration files of a minimum total size of $10 + 3 * \#LAs + 4 * \#workers$. These files include for instance implicit connections of DIET elements and scheduling configuration concerns. Therefore, it is clear that handling diet architecture may easily be a huge task to be done by the designer.

To summarize, grid experiments exhibit situations where the abstraction significantly supply design simplicity. For reuse concerns, it is improved through the ability to have a same abstract assembly used in various situations, from simple to complex ones. As illustrated by obtained performance results, this property is relevant to achieve good performance in different execution contexts. It is also to note that no CCM framework modifications were done. Efforts have been done to support a heterogeneous assembly by the generic deployment tool ADAGE.

7.3. Discussion

The presented cluster and grid experiments show that the proposed model can be implemented with a very small overhead, of the order of an intra-process inter-component invocation cost. Therefore, it is possible to obtain the performance of the underlying component model or the performance of a reused master–worker environment. This paper focuses on CCM as we are very familiar with it and there are tools available for running grid experiments. We claim that the results hold for other similar component models such as CCA or GCM. As intra-process inter-component invocations are more efficient within CCA, the difference with MPI applications will be even smaller: high performance can be achieved with the proposed model.

However, several issues remain open. First, the automatic conversion from an abstract ADL to a concrete one is currently not implemented. For the present work, the conversion is done in three steps, which are to reserve a set of machines, to apply manually the conversion script, and to launch the concrete ADL with the deployment tool ADAGE. This was particularly annoying for DIET whose actual architecture has to be computed with respect to the available resources. In order to avoid doing manually this step, planning algorithms for DIET has recently been added into ADAGE.

Second, component proxies are currently specific to an application. Though their implementation is straightforward, their reuse for various applications requires mechanisms to automate this implementation. One possibility should be to define some kinds of parameterized components [31]. However, such work is very recent and does not yet handle online compilation. Another possibility should be to use reification, but this would generate overhead that may not be acceptable on HPC machines. A more promising approach would be to handle RMI invocations inside the pattern with a generic message-passing API. Such a feature is partially available in the CORBA routing specification. However, it is supported in message-based component models such as SCA [17].

Third, we face a scalability issue when using proxy components and when the number of workers becomes important ($O(1000)$). This issue is related to the RMI semantic and the number of threads created in the proxy components to manage the master requests. In particular, the RMI semantic prevents a master invocation from finishing before a worker finishes a request computation. The number of concurrent threads created in the proxy may be the same as the number of workers, which limits the scalability of the approach for some patterns: operating systems usually only support a few thousands of threads and socket connections. Moreover, there may be a problem of memory consumption because of the thread stacks. Hence, a pure RMI-based solution is challenging to scale. Therefore, being able to handle RMI invocations with a message-passing API will enable to keep the number of threads constant and to ease scalability.

8. Conclusion and future work

Designing efficient, portable and maintainable applications is an important but difficult objective. It turns out grids make it even more difficult due to the various parallelism levels of the infrastructure. A solution to achieve such a goal is to increase the level of abstraction of applications without affecting performance. This paper focused on the master–worker paradigm as it is extensively used. Its contribution consists in showing that it is possible to adapt software component technologies to support this paradigm at a high level of abstraction while enabling efficient execution. Moreover, it is possible to reuse existing master–worker environments like DIET. Ease of use and efficiency are achieved by controlling non-functional elements—the number of workers as well as the request transport policies—without modifying the application: all the patterns used in this paper have been experimented without recompiling the master or the worker components.

For an every day use of the proposed model, several steps are lacking. First, a mechanism is needed to adapt the pattern to the user component, like for example a kind of component template [31] or the ability to deal with RMI invocations with a message-passing interface for the pattern. Second, the collection instantiation needs to be integrated either into a deployment tool or into the component framework. Last, an adaptability support is required to deal with multi-master scenarios and with dynamic resources. We have started to work on these two last points, by revisiting the model of the deployment tool ADAGE and by studying how to integrate an adaptability framework such as DYNACO [32,33].

Acknowledgement

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative supported by INRIA under the framework of the ALADDIN Development action (see http://aladdin.inria.fr).

References

[1] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, H. Casanova, A GridRPC Model and API for End-User Applications, Technical Report GFD-R.052, Open Grid Forum, June 2007.
[2] H. Casanova, J. Dongarra, NetSolve: a network-enabled server for solving computational science problems, Int. J. Supercomput. Appl. High Perform. Comput. 11 (3) (1997) 212–223.
[3] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, S. Matsuoka, Ninf-G: a reference implementation of RPC-based programming middleware for grid computing, J. Grid Comput. 1 (1) (2003) 41–51.
[4] E. Caron, F. Desprez, DIET: a scalable toolbox to build network enabled servers on the grid, Int. J. High Perform. Comput. Appl. 20 (3) (2006) 335–352.
[5] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, M. Leboisky, SETI@home-massively distributed computing for SETI, in: IEEE Computer Society, Los Alamitos, CA, USA, 2001, pp. 78–83.
[6] Berkeley Open Infrastructure for Network Computing, 2002. <http://boinc.berkeley.edu/>.
[7] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Néri, O. Lodygensky, Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests, and convergence with grid, FGCS 21 (3) (2005) 417–437, doi:10.1016/j.future.2004.04.011).
[8] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Comput. 30 (3) (2004) 389–406.
[9] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Dazzi, D. Laforenza, N. Tonellotto, P. Kilpatrick, Behavioural skeletons in GCM: autonomic management of grid components, in: D.E. Baz, J. Bourgeois, F. Spies (Eds.), Proceedings of the International Euromicro PDP 2008: Parallel Distributed and Network-Based Processing, IEEE, Toulouse, France, 2008, pp. 54–63, doi:10.1109/PDP.2008.46.
[10] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, C. Zoccolo, Components for high performance grid programming in Grid.it, in: Proceedings of the International Workshop on Component Models and Systems for Grid Applications, CoreGRID Series, Springer, Saint-Malo, France, 2005, pp. 19–38.
[11] Programming Model Institute, Innovative Features of GCM (with Sample Case Studies): A Technical Survey, Technical Report, CoreGrid, D.PM.07, October 2007.
[12] C. Szyperski, D. Gruntz, S. Murer, Component Software – Beyond Object-Oriented Programming, second ed., Addison-Wesley, ACM Press, 2002.
[13] OMG, Deployment and Configuration of Component-based Distributed Applications Specification, v4.0, Document formal/2006-04-02, April 2006.
[14] A. Flissi, J. Dubus, N. Dolet, P. Merle, Deploying on the grid with deployware, in: CCGRID'08: Proceedings of the 2008 8th IEEE International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Washington, DC, USA, 2008, pp. 177–184, doi:10.1109/CCGRID.2008.59.
[15] S. Lacour, C. Pérez, T. Priol, Generic application description model: toward automatic deployment of applications on computational grids, in: 6th IEEE/ACM International Workshop on Grid Computing (Grid2005), Springer-Verlag, Seattle, WA, USA, 2005.
[16] OMG, CORBA Component Model, v4.0, Document formal/2006-04-01, April 2006.

[17] M. Beisiegel, H. Blohm, D. Booz, M. Edwards, O. Hurley, S. Ielceanu, A. Miller, A. Karmarkar, A. Malhotra, J. Marino, M. Nally, E. Newcomer, S. Patil, G. Pavlik, M. Raepple, M. Rowley, K. Tam, S. Vorthmann, P. Walker, L. Waterman, SCA Service Component Architecture – Assembly Model Specification, Version 1.0, Technical Report, Open Service Oriented Architecture collaboration (OSOA), March 2007.
[18] E. Bruneton, T. Coupaye, J.B. Stefani, The Fractal Component Model, Version 2.0-3, Technical Report, ObjectWeb Consortium, February 2004.
[19] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, C. Pérez, GCM: a grid extension to fractal for autonomous distributed components, Special Issue Ann. Telecommun.: Software Compon. – Fract. Initiat. 64 (1–2) (2008) 5–24.
[20] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, C. Zoccolo, Components for high performance grid programming in the Grid.it project, in: International Workshop on Component Models and Systems for Grid Applications, Springer-Verlag, Saint-Malo, France, 2004.
[21] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, J. Darlington, ICENI: optimisation of component applications within a grid environment, J. Parallel Comput. 28 (12) (2002) 1753–1772.
[22] J. Magee, N. Dulay, J. Kramer, A constructive development environment for parallel and distributed programs, in: Proceedings of the International Workshop on Configurable Distributed Systems, Pittsburgh, US, 1994, pp. 4–14.
[23] D.E. Bernholdt, B.A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T.L. Dahlgren, K. Damevski, W.R. Elwasif, T.G.W. Epperly, M. Govindaraju, D.S. Katz, J.A. Kohl, M. Krishnan, G. Kumfert, J.W. Larson, S. Lefantzi, M.J. Lewis, A.D. Malony, L.C. McInnes, J. Nieplocha, B. Norris, S.G. Parker, J. Ray, S. Shende, T.L. Windus, S. Zhou, A component architecture for high-performance scientific computing, Int. J. High Perform. Comput. Appl. 20 (2) (2006) 163–202.
[24] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, J. Shalf, The cactus framework and toolkit: design and applications, in: Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Springer, 2003.
[25] J. Larson, R. Jacob, E. Ong, The model coupling toolkit: a new Fortran90 toolkit for building multiphysics parallel coupled models, Int. J. High Perform. Comput. Appl. 19 (3) (2005) 277–292.
[26] Fraunhofer SCAI, MpCCI 3.1.0-1 Documentation – Part I Overview, January 2009. <http://www.scai.fraunhofer.de/mpcci>.
[27] A. Ribes, C. Caremoli, Salome platform component model for numerical simulation, Comput. Software Appl. Conf., Annual Int. 2 (2007) 553–564, doi:10.1109/COMPSAC.2007.185).
[28] H.L. Bouziane, C. Pérez, T. Priol, 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, 2006.
[29] H.L. Bouziane, De lábstraction des Modèles de Composants Logiciels pour la Programmation dápplications Scientifiques Distribuées, Thèse de doctorat, Université de Rennes 1, IRISA/INRIA, Rennes, France, February 2008.
[30] E. Caron, F. Desprez, F. Lombard, J. Nicod, M. Quinson, F. Suter, A scalable approach to network enabled servers, in: B. Monien, R. Feldmann (Eds.), Proceedings of the 8th International EuroPar Conference, Lecture Notes in Computer Science, vol. 2400, Springer-Verlag, Paderborn, Germany, 2002, pp. 907–910.
[31] J. Bigot, C. Pérez, Increasing reuse in component models through genericity, in: Proceedings of the 11th International Conference on Software Reuse, LNCS, Springer-Verlag, Falls Church, Virginia, USA, 2009.
[32] F. André, H.L. Bouziane, J. Buisson, J.-L. Pazat, C. Pérez, Towards dynamic adaptability support for the master–worker paradigm in component based applications, in: CoreGRID Symposium in Conjunction with Euro-Par 2007 Conference, Springer, Rennes, France, 2007, pp. 117–127.
[33] F. André, G. Gauvrit, C. Pérez, Dynamic adaptation of the master–worker paradigm, in: Proc. of the IEEE 9th International Conference on Computer and Information Technology, IEEE Computer Society, Xiamen, China, 2009.
[34] M. Krishnan, Y. Alexeev, T.L. Windus, J. Nieplocha, Multilevel parallelism in computational chemistry using common component architecture and global arrays, in: SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, Washington, DC, USA, 2005, p. 23, doi:10.1109/SC.2005.46.
[35] G. Antoniu, H.L. Bouziane, M. Jan, C. Pérez, T. Priol, Combining data sharing with the master–worker paradigm in the common component architecture, Cluster Comput. 10 (3) (2007) 265–276.
[36] R. Bolze, F. Cappello, E. Caron, M. DaydT, F. Desprez, E. Jeannot, Y. JTgou, S. LantTri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, T. IrTa, Grid'5000: a large scale and highly reconfigurable experimental grid testbed, Int. J. High Perform. Comput. Appl. 20 (4) (2006) 481–494.
[37] A. Denis, C. PTrez, T. Priol, PadicoTM: an open integration framework for communication middleware and runtimes, FGCS 19 (4) (2003) 575–585.