Modeling and executing Master-Worker applications in component models

Hinde Lilia Bouziane, Christian Pérez, Thierry Priol
IRISA/INRIA, Campus de Beaulieu, 35042 Rennes cedex, France
{Hinde.Bouziane,Christian.Perez,Thierry.Priol}@irisa.fr

***Abstract*— This paper describes work in progress to extend component models to support Master-Worker applications and to let them to be executed on Grid infrastructures. The proposed approach is generic enough to be applied to existing component models such as the OMG CORBA and the ObjectWeb FRACTAL component models. One objective of our research is to relieve Grid application designers of managing low level programming and implementation aspects. With the proposed approach, a designer has only to cope with the description of an abstract view of the application architecture in which he has to specify what the master and the workers have to do while leaving the system environment to manage the low level aspects such as communication between the master and the workers.**

I. INTRODUCTION

The Grid vision introduced at the end of the nineties has now become a reality with the availability of quite a few Grid infrastructures. Although most of the research and development efforts have been spent in the design of Grid middleware systems, the question of how to program such large scale computing infrastructures remains open. Programming such computing infrastructures is quite complex considering their parallel and distributed nature. The programmer vision of a Grid infrastructure is often determined by its programming model. The level of abstraction that is proposed today is rather low, giving the vision either of a parallel machine, with a message-passing layer such as MPI, or a distributed system with a set of services, such as Web Services, to be orchestrated. Both of these two approaches offer a very low level programming abstraction and are not really adequate, limiting the spectrum of applications that could take benefit from Grid infrastructures. Of course such approaches may be sufficient for simple applications but a Grid infrastructure has to be generic enough to be able to handle both simple and complex applications. To overcome this situation, it is required to propose high level abstractions to ease the programming of Grid infrastructures.

Several research groups are already investigating how to design or adapt programming models that provide a higher level of abstraction. Among these models, component-oriented programming models are a good candidate to deal with the complexity of programming Grid infrastructures. A Grid application can be seen as a collection of components interconnected in a certain way that must be deployed on available computing resources managed by the Grid infrastructure. Components can be reused for new Grid applications, reducing the time to build new applications. However, component models are usually unable to cope with the requirements of scientific applications that have to be executed using a Grid infrastructure. This is especially true when trying to combine parallel programming paradigms with component programming. Previous efforts [1] have been made to extend component models to be able to encapsulate a SPMD code into a collection of components and to let several collections to be composed like ordinary components. Such an extension has been proved valuable for scientific applications that require the coupling of several simulation codes (multi-physics applications).

In this paper, we propose to handle the *Master-Worker* parallel programming paradigm into component models. This paradigm is very relevant to parametric applications where several instances of the same code have to be executed simultaneously with different parameter values. This kind of application is suitable for computational Grid infrastructures since they are embarrassingly parallel. This can be shown by numerous research activities dealing with the design of *Master-Worker* software Grid-enabled environments such as for Global Computing systems (SETI@Home [2], Javelin [3], XtremWeb [4], BOINC [5]) or for Network Enabled Server environments (DIET [6], NetSolve [7], Ninf-G [8], Nimrod/G [9]). Most of these Grid-enabled environments only focus on supporting the execution of *Master-Worker* applications by providing the master-worker paradigm. Some of them provide another paradigm, like task farming for Ning-G, but they remain very specialized environments.

The remainder of this paper is divided as follows. Section II gives an overview of existing component models and how they cope with the *Master-Worker* programming paradigm as well as their shortcomings. Section III presents our approach to better support the *Master-Worker* programming paradigm using a higher level of abstraction. Section IV explains how our approach can be applied to existing component models such as OMG's CCM and ObjectWebs's FRACTAL. This is further illustrated in section V by taking an example of application scenario involving a *Master-Worker* pattern. Finally, we present some concluding remarks in section VI.

II. COMPONENT BASED MASTER-WORKER APPLICATIONS IN EXISTING COMPONENT MODELS

A. Overview of component models

Software Component technology has been emerging for a few years [10], even though its underlying intuition is not very recent [11]. A software component according to Szyperski [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*. Let us review the different elements of this definition:

1) 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.

2) Ports: to be able to interact with other components, a component defines external interfaces named ports. A port is a programming artefact 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 component to a *server* port of another component with compatible type.

3) Separation of concerns: component models aim to separate functional and non functional concerns. The goal is to not burden component implementor with non functional features like security. Such features are activated during the configuration phase and are generally provided by the framework in which the component are executed.

4) Assembly: the assembly phase 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 [13] or in FRACTAL [14]. Another approach consists in using run-time composition, like for example in CCA [15].

5) Deployment: a component is a binary unit of deployment. It contains – or references – an implementation (binary code) and the constraints associated to it, like operating system, processor and amount of memory requirements. It may also contain several implementations and so, alternative requirements. These properties help a deployment tool to decide the resources an instance of the component may be installed on.

B. Overview of CCM and Fractal component models

*1) CCM (*CORBA *Component Model):* the CORBA Component Model [13] is part of the latest CORBA [16] (*Common Object Request Broker Architecture*) specifications (version 3). The CCM specifications allow the deployment of components into a distributed environment on different heterogeneous servers.

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

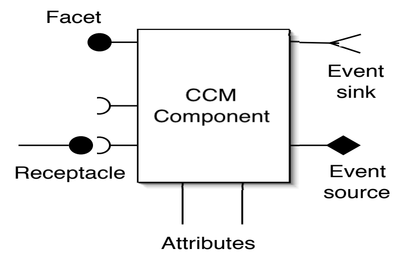


Fig. 1. A CCM component.

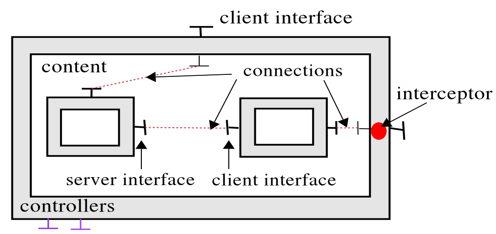


Fig. 2. The structure of a composite component in Fractal.

CCM offers a complete model to develop a component based application: (1) a design model to describe components and their ports using the CORBA 3 version of the OMG Interface Definition Language (IDL), which extends the OMG IDL of the version 2 of CORBA; (2) an assembly model to describe an application architecture thanks to an Architecture Description Language; (3) a packaging and deployment model to deploy an application from an assembly description. It is worthwhile to remark that this description is just the initial state of the application. The deployment model of CCM is fully dynamic: the architecture of an application may be changed by modifying its connections and/or by adding/removing components; (4) an execution model to offer a set of standard services to a component, such as security, events, and persistence; and (5) a component's life cycle management model to create, find or remove component instances through the use of entities named *homes*. Point (4) enables the same component to be hosted by different framework implementations.

2) Fractal: The FRACTAL component model [17] is developed by the ObjectWeb [18] consortium. It is a hierarchical component model which defines *"primitive"* and *"composite"* components. A *composite* component, as described in Figure 2, may contain several (sub-)components that form its *content*. FRACTAL defines also *"controllers"* to manage the life cycle of a component content: managing connections between sub-components or adding/removing dynamically new components, etc. A *composite* component can introduce *interceptors* (a kind of *controllers*) to enable redirection of extern requests to its sub-components. Usually, controllers are implemented as objects.

Unlike CCM, FRACTAL does not specify any IDL language. The design model is part of the assembly model: the ADL provided by FRACTAL allows then to specify both components and their composition in a same phase.

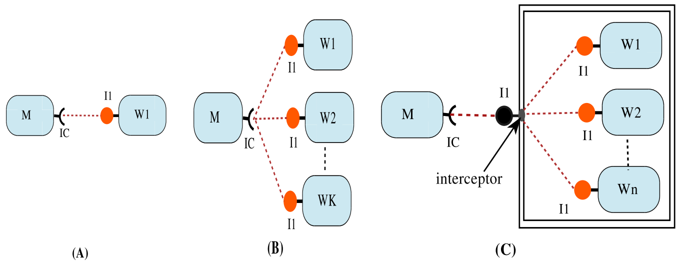


Fig. 3. The Master-Worker application architectures.

C. Composing a Master-Worker application in component models

This paper focuses on the specification of a *Master-Worker* application designed to be executed on a Grid environment. Such an application follows a *Client-Server* design pattern which can be a part of a complex component based application. A first goal is to be able to develop *Master* and *Worker* components independently from the execution environment in which they will be deployed. A *Master* defines a *client* port to send its requests to a *Worker* component that provides this port. As in existing *Master-Worker* environments, delivering a request to a *Worker* component should be transparent for the components implementor. The only constraint which can be imposed to a *Master* implementor, is the use of APIs allowing simultaneous request invocations. An implementor is at least aware of requests dependencies to allow the independent ones to be executed in parallel.

Let examine relevant points related to the description of a *Master-Worker* application with existing component models:

1) The benefit of an ADL description: assembling a *Master-Worker* application at run-time, like in CCA, results in explicitly encoding many issues inside a program like the deployment and the configuration of component instances, as well as the management of the application architecture. Therefore, the level of separation of concerns is quite low.

Using an ADL description enables these concerns to be separated. Hence, modifications to the application architecture or code reuse are eased. Moreover, even thought it is the initial application state that is described in ADL, it may change during the execution. That is why, this section goes on with components models which provides an ADL language.

2) Component compositions: There are two main possibilities to encode a *Master-Worker* application into a flat model like CCM. The first alternative consists in composing a *Master* component M with only one *Worker* component W_1 (Figure 3.(A)). At execution time, W_1 may treat incoming requests either sequentially or in parallel – a CCM component may be multithreaded. Hence, a multithread component may create a thread for each request. This approach appears well suited for SMP machines. However, there is no mechanism to control the number of working threads associated to a component.

The second alternative is to specify several *Worker* instances, all of them connected to M (Figure 3.(B)). The code of the *Master* should be modified as its *client* port becomes a *"multiple"* port: as the multiple instances of *Worker* to which the *Master* is connected are visible, the *Master* has the burden to implement its scheduling strategy.

A solution is to delegate to a component (or a set of components) the task of delivering the request from the *Master* to a *Worker*. Hence, the request strategy is separated from the master and the *Worker* components. We may then envision the use of classical *Master-Worker* environment to deal with request delivery policy. With existing ADLs, this work is to be done by the application designer at a phase the execution environment, i.e. the the available resources of the Grid, is not known.

3) Number of worker component instances: there is a dependency between the number of *Worker* components and a deployment environment. In fact, a *Master-Worker* application in general does not constrain the number of workers to be statically fixed as usually performed in an ADL description.

Moreover, workers are components which may be added and removed at run-time. Such a behavior is suitable to adapt the number of *Workers* to the execution requirements. Thus, it appears suitable to be able to associate this dynamic behavior to *Worker* components in an ADL description. Supported architecture modifications are then visible, for example to adaptability policies.

4) Request delivery policy: another alternative to externalize the requests transport management from the *Master* code is to use interceptor. As described in Figure 3.(C), *Worker* components are embedded into a composite component. A request delivery policy may be implemented inside an interceptor.

However, the use of an interceptor (i.e. an object) may become a bottleneck in a distributed and heterogeneous environment as it has to schedule all incoming requests. Hence, it represents a non scalable scheduling policy.

It seems more suitable to be able to use scalable and efficient request delivery policies, as the one developed within dedicated *Master-Worker* environments like SETI@Home, Javelin, XtremWeb, BOINC, or in NES like NetSolve, DIET, etc.

D. Related works

To our knowledge, there is not any specification model for *Master-Worker* applications in existing component models. However, since the expressed relation between a *Master* and *Workers* follows the one-to-many connection mode, the use of *"objects groups"* in the world of distributed objects like Fractal/ProActive [19] or OGS [20], is the closest manner to develop a *Master-Worker* application.

Fractal/ProActive has a group communication system [19] which allows point-to-point and one-to-many communications (broadcast, scatter and gather operations). It is used for replication, fault tolerance and data distribution for task-parallel applications. For a *Master-Worker* application however, the transport of a request from a *Master* to one *Worker* inside a group is not directly supported by available communication modes. To allow that, a programmer should manage the group in the *Master* code to adapt the number of the *Master* requests to the group size. Therefore, a transparent access to the *Workers* is lost.

In the OGS [20] (Object Group Service of CORBA), the principal interaction mode is based on transmitting a same method invocation on a group to each of the members. It is also used for replication, fault tolerance and data distribution for parallel applications [21]. The OGS for CORBA is implemented either by using an existent group communication system and modifying the transport service [22] of the ORB (Object Request Broker), by introducing interceptors [23], or by introducing CORBA compliant services from scratch [24]. These solutions are at a low programming level and implementation specific.

III. A MODEL FOR SOFTWARE COMPONENT BASED MASTER-WORKER APPLICATIONS

A. Overview of the proposed Model

This paper aims at extending software component models to increase the abstraction level for *Master-Worker* applications. The proposal is to allow a designer to only specify a set of *Worker* instances to which a *Master* component is connected. The number of *workers* is no longer a preoccupation of the designer. Similarly, requests transport concerns are handled separately while advanced transport policies are possible.

Figure 4 presents an overview of the different elements of the proposal. First, the application designer gives an *"abstract architecture description"* in which he/she specifies a *"collection"* of *Worker* components. 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 DIET may be used.

Once the deployment environment is known, an initial number of worker components can be fixed and a suitable requests transport policy can be chosen. From these choices, the abstract architecture description is converted into a concrete ADL description during a *"transformation"* process. In the example of Figure 4, 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.

The remaining of this section introduces in more details the concepts and steps of the model using a generic description and a generic ADL. The projections to two specific component models, CCM and Fractal, are introduced in Section IV.

B. A collection definition

The central piece of the model is the notion of collection. A collection is a set of *"exposed"* ports. Up to Section III-E, we assume they are restricted to a *server* port. These ports are defined by components which belong to the collection. An example of an XML collection definition and a graphical representation are shown in Figure 5.

A collection is an abstract concept that does not define the number of elements. It is used to group (independent) component instances, any of them providing the same role with respect to any component outside the collection. For instance, a collection of *Worker* components inside a *Master-Worker* application defines instances of *Worker* that will participate in answering to *Master* requests. Incoming invocations are (independently) dispatched to any component in the collection according to an associated request delivery policy. It differs from a group communication where an invocation is assumed to be dispatched to *all* elements of the group.

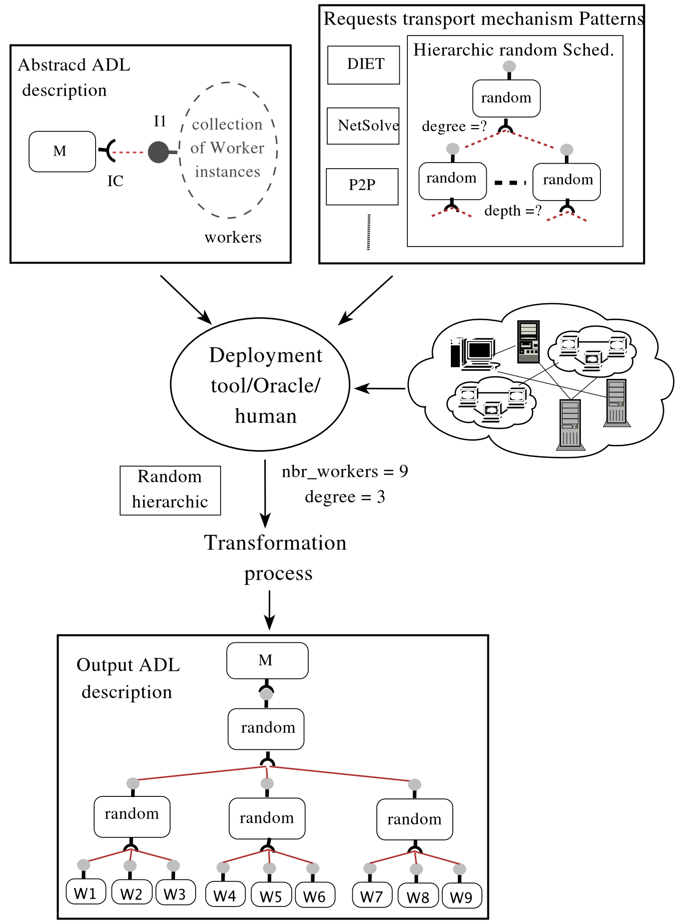


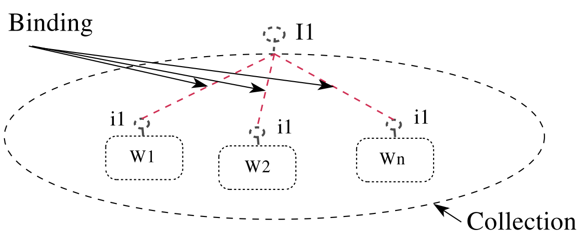
Fig. 4. An overview of a *Master-Worker* application model.

C. An abstract ADL description

We define an abstract ADL description as an assembly specification which composes both component instances and abstract elements such as collections. The ADL used in an abstract description could be an extension of an existing ADL. This ADL is said abstract because it is not a complete architecture description: the number of *Worker* instances as well as the request delivery mechanism are missing.

With respect to a *Master-Worker* application, a designer similarly handles both components and collections. He/She defines component and collection instances and composes them together by connecting compatible ports. Figure 6 shows the abstract ADL of the *Master-Worker* application corresponding to the graphical representation shown in Figure 4.

```
<componentType id="Worker">
  <serverInterface id="i1" type="Computation"/>
</componentType>
<collectionDefinition id="Workers">
  <exposedInterface id="I1" type="Computation"/>
  <element type="Worker"/>
  <binding>
    <externInterface interfaceRef="I1"/>
    <internInterface type="Worker" interface="i1"/>
  <binding>
</collectionDefinition>
```



I1 : exposed interface of type "Computation".
Wi : component instance of type Worker.
i1 : interface of type "Computation".
n : undetermined number of Worker instances.

Fig. 5. A textual and graphical definition of a worker collection.

```
<componentType id="Master">
  <clientInterface id="IC" type="Computation"/>
</componenttype>
<collectionType id="Workers">
  <exposedInterface id="I1" type="Computation"/>
</componentType>
<componentInstance id="M" type="Master" />
<collectionInstance id="W" type="Workers"/>
<connection>
  <clientInterface componentInstance="M"
                   interfaceInstance="IC"/>
  <serverInterface collectionInstance="W"
                    interfaceInstance="I1"/>
</connection>
```

Fig. 6. An abstract ADL description for a *Master-Worker* application.

As it can be noted, the composition of an application barely changes. Moreover, the number of *Worker* instances and the request transport mechanism between the *Master* instance M and the collection W are not specified at this stage.

D. Transformation patterns

An abstract ADL description needs to be transformed into a concrete ADL to obtain an actual deployable application. For that, an initial number of workers inside a collection has to be fixed. This number can be specified in a static way. In more advanced case, it can be done by requesting, for instance, "as much workers as actual available resources". Of course, this number may vary during runtime. The concrete ADL contains also a set of components that implements a selected request delivery policy. As an efficient policy selection depends on the actual resources as well as the actual number of workers, it should be selected at deployment time. The set of selectable policies may be constrained by specifying acceptable ones.

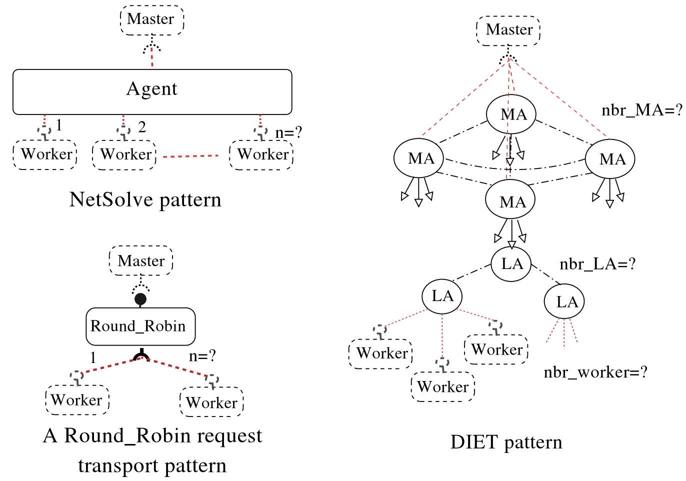


Fig. 7. Three examples of patterns.

A request delivery policy is associated to a *pattern*. A *pattern* is composed of interconnected components that implements a request transport policy. Figure 7 presents some examples of patterns: a simple Round-Robin pattern, the centralized scheduler used by NetSolve, and the hierarchical architecture used by DIET.

A pattern may depend on parameters related to the environment. For example, it may depends on the actual number of workers. A simple usage is to configure the Round-Robin or NetSolve pattern (Figure 7). A more complex usage is represented by patterns like the hierarchical DIET pattern (Figure 7) in which the number of workers determines the number of intermediate components (Figure 4).

Once a request policy is selected, the concrete architecture of the *Master-Worker* application exposed in Figure 4 can be derived. It is performed during a *"transformation"* process. This process consists in replacing a collection instance definition of an abstract ADL description by: (1) the *Worker* instances for which the initial number is now known, (2) an instance of the selected pattern, and, (3) connections between Worker instances and the pattern, and between the pattern and the *Master* component. This last step is done with respect to the pattern composition rules.

The actual pattern is concealed in its associated transformation. A *transformation* is seen as a program which transforms an abstract ADL to a concrete ADL one for a specific pattern. It can be written in any languages like C/C++, JAVA, Python, XSLT, etc. Hence, it is out the scope of the model to describe patterns. It heavily depends on the deployment tool. Section IV shows an example based on XSLT.

E. Generalizing collection usage

Up to now, only *server* ports were assumed to be exposed by a collection. It was mainly motivated by the targeted *Master-Worker* applications. While allowing *client* ports to be exposed by a collection, two other scenarios are possible. They are also supported by the model. Let review the the three possible composition scenarios, which are illustrated in Figure 8.

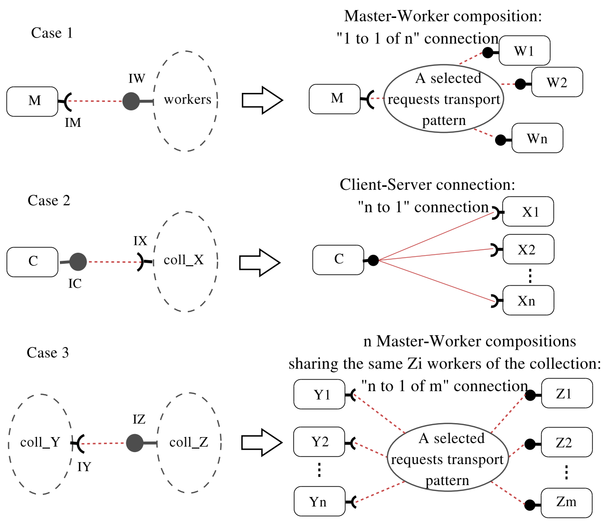


Fig. 8. The three possible component/collection compositions and their associated semantic.

```
interface Computation {..};
collection Workers {
   provides Computation I1;
};
```

Fig. 9. An extended IDL3 definition of a collection.

The first case is a *Master-Worker* application, made of a collection providing a *server* port. This case has been already tackled by this paper. In the second case, the collection exports a *client* port, which is connected to a server port of a standard component. It is a classical Client-Server pattern which does not require any special request delivery policy: all components of the collection may be directly connected to the standard component C. The third case considers the connection of two collections, Y providing a *client* port, and, Z providing a *server* port. It can be seen as n independent masters accessing a shared set of workers. Advanced request delivery policies are needed to guarantee an efficient handling of *Master* requests.

IV. Projecting the Proposed Model on Existing Component Models

We have so far presented an extension of component models for abstracting the level of description of *Master-Worker* applications. This section introduces its application to the Corba Component Model and to FRACTAL.

A. CCM

1) Extended IDL3: we extend the IDL3 language with the `collection` keyword to enable a collection to be described. A collection may contain the description of one or several CCM ports, like facets, receptacles, event sources and sinks. Figure 9 presents the extended IDL3 definition corresponding to the example of Section III-B.

The extended IDL3 allows only an abstract collection type to be defined. We need another language to describe the content of a collection, i.e. the component type(s) that a particular collection implementation contains as well as the connection of the ports of the collection with the ports of its internal component(s). We introduce a new language, called Collection Description Language (CDL), which is an XML vocabulary inspired from the CCM Component Assembly Language (CAD). An example of a CDL description for the *Workers* collection is presented in Figure 10. The description contains two parts. First, references to some component types are defined. Second, the (logical) connection between the collection ports and the internal component ports are defined.

```
<collection type="Workers">
  <componentfiles>
    <componentfile id="Worker">
      <fileinarchive name="def.csd"/>
    </componentfile>
  </componentfiles>
  <bindings>
    <binding>
      <providescollectionport>I1
      </providescollectionport>
      <provideselementports>
        <providesport>
          <providesporti>i1</providesport>
          <componentref idref="Worker/>
        </providesport>
      </provideselementports>
    </binding>
  </bindings>
</collection>
```

Fig. 10. Example of a collection description in CDL.

```
componentassembly
 ├─► collectionfiles
 │    └─► collectionfile // gives a reference to a CDL description
 ├─► transformations
 │    └─► transformation //gives a reference of a transformation executable
 ├─► partitioning
 │    └─► collectionInstance
 │         ├─► collectionfileref
 │         └─►applytransformation // explicit selectetion of a pattern
 └─► connections
      └─►connectcollection
           ├─► usesport
           │    ├─►usesidentifier
           │    ├─►componentinstantiationref
           │    ├─►collectionInstanceref
           │    └─►...
           ├─► providesport
           │    └─► ... equivalent of usesport
           └─► ... // other port
```

Fig. 11. The abstract CAD language is an extension of the CCM CAD language. The new elements are presented.

2) An abstract CCM *assembly language:* the CCM Component Assembly Description (CAD) needs to be extended to allow the connection between collections and components or collections. With respect to Section III-C, the result is an abstract CAD. Figure 11 presents the major elements that are inserted in the CCM CAD. XML elements related to transformations are also added. They can be optionally used to specify or restrict the choice of a request delivery pattern.

```
<definition name="Worker">
  <interface name="I1" role="server"
             signature="Computation" />
</definition>
<definition name="Worker_impl" extends="Worker">
  <content class="WorkerImpl"/>
</definition>
          <!- A pattern/transformation ->
<definitionPattern name="RoundRobin">
  <transformation type="xsl"
                  source="RoundRobinFractal.xsl"/>
</definitionPattern>
           <!-  A collection definition ->
<definition name="Workers_impl" extends="Worker">
  <component name="w" definition="Worker_impl"/>
  <binding external="this.I1" internal="w.I1"/>
  <pattern interface="this.I1"
           transformation="RoundRobin"/>
</definition>
```

Fig. 12. Example of description of collection in FRACTAL. New tags are the `definitionPattern` and the content of a definition for a collection.

If required to keep the compatibility with CCM, all these tags can be added in the `extension` tag of CCM documents.

3) A pattern and a transformation: to validate our proposition, we have implemented an XSLT transformation program. Its inputs are an abstract CAD file, the name of the collection instance to apply the transformation on, and the number of *Worker* instances. The transformation program then generates the adequate number of *Worker* instances, inserts a scheduler component between the master and the Workers, and generates all the connections. In our example, the scheduler component implements a Round-Robin policy (Figure 7). If there are several collections, the transformation program has to be invoked once per collection. Eventually, the document is a standard CAD.

The XSLT transformation, the Round-Robin component, and simple *Master* and *Worker* components have been implemented to validate the feasibility of the proposed approach.

B. FRACTAL

It is possible to use a similar approach than the one defined for CCM to implement the collection concept in FRACTAL, but by only using primitive components. However, it appears more interesting to take advantage of the hierarchic characteristic of the FRACTAL model. Hence, a collection can be implemented as a specialized composite: it is a sub-type of component.

1) An abstract ADL language: FRACTAL does not offer an IDL language to specify component types. Component types are directly defined in the ADL description. Therefore, we need to extend the `definition` tag to allow the description of a collection as represented in Figure 12. In the case of a collection, the `definition` tag contains a list of internal components (tag `component`), a list of the bindings of the internal components with some ports of the collection, and optionally the pattern(s) that may be applied to each port of the collection. As for CCM, the resulting ADL is abstract.

2) Pattern and transformation: the transformation phase aims at generating a concrete ADL. In the case of FRACTAL, we need to generate a classical FRACTAL ADL from the extended ADL presented hereinbefore. This may be achieved in a similar way as for CCM by applying an XSLT transformation.

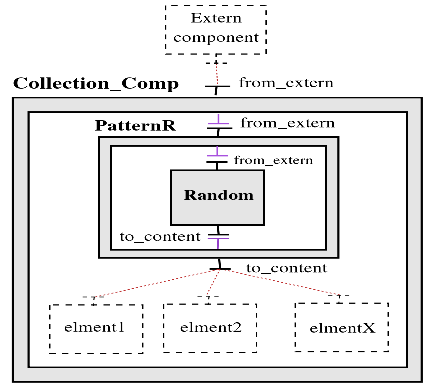


Fig. 13. A pattern example in FRACTAL.

```
do
  param1 = optimize_param1(param2);
  param2 = optimize_param2(param1);
while eval(param1, param2) < criteria
```

Fig. 14. Pseudo-algorithm of an application made of the global optimization of two dependent optimization problems.

A FRACTAL collection is a special kind of composite that contains not only the computing component instances but also the components implementing the request delivery policy. In order to group together those components, it appears adequate to put them into a composite. For example, Figure 13 presents a graphical description of a pattern: the external composite represents the collection while the internal composite embeds all component related to the request delivery policy. The Random pattern scheduler is implemented with one component.

3) Discussion: the composite feature of FRACTAL enables a definition of collections as a sub-type of component, a well-defined point to which attach adaptability features, and an encapsulation of components involved in request delivery policy. In a flat architecture, structural information, like components involved into a collection or to the request delivery policy has to be manually maintained while it is part of the application architecture description in a hierarchic model.

V. USE CASES IN CCM

This section briefly describes two examples of application scenarios involving a *Master-Worker* pattern. While the classical example is a *Master* connected to a collection of workers, a more complex pattern is presented.

Let consider an optimization application which requires the optimization of two interdependent sub-problems. Figure 14 sketches the algorithm of the main loop. A possible implementation of such an application is a *Master* component, connected to two collection, that alternatively solves one problem after the other as represented in Figure 15.

Another solution is to transform the algorithm in a hierarchical one. It leads to the architecture displayed in Figure 16.

```
interface optimize1 {
   param1_t optimize_param1(in param2_t p);
};
interface optimize2 {
   param2_t optimize_param2(in param1_t p);
};
collection optimize1 {
  provides optimize1 p1;
};
collection optimize2 {
  provides optimize2 p2;
};
component Master {
  uses optimize1 p1;
  uses optimize2 p2;
};
```

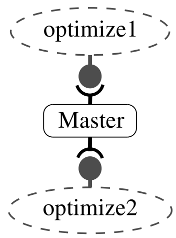


Fig. 15. Definition of the component of the global optimization application with a Master that alternatively solves the two sub-problems.

```
component opti1 {
  provides optimize1 p1;
  uses     optimize2 p2;
};
collection optimize1 {
  provides optimize1 p1;
};
collection optimize2 {
  provides optimize2 p2;
};
```

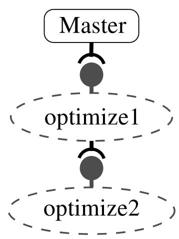


Fig. 16. Hierarchical component architecture of the global optimization application.

VI. Conclusions and future works

Software component models are becoming the dominant model to develop distributed applications. However, existing component models do not provide any abstraction for developing applications using the *Master-Worker* paradigm while it is a widely used paradigm in grid applications.

This paper studies the introduction of a redefined concept of collection and of ADL transformation to provide a component model with a higher level of abstraction to specify a *Master-Worker* application. The proposed model has several advantages captured from those provided by classical *Client-Server* environments like Global Computing systems or Network Enabled Server environments. To submit multiple and independent requests, a user has only to develop the functional codes of a *Master* and a *Worker* components. He/She can then compose its application by connecting components and collections. Request delivery concerns and workers instantiation are at this stage transparent for the application designer. The choice of a request delivery policy and the number of workers become adaptable to the actual deployment environment. The generic component model has been projected on CCM and FRACTAL. A prototype implementation has shown the feasibility of the approach.

While the presented model introduced the *collection* concept, the associated dynamic aspect is yet to be studied. In particular, we plan to study its integration with dynamic adaptation frameworks.

References

[1] C. Pérez, T. Priol, and A. Ribes, “A parallel CORBA component model for numerical code coupling,” *The International Journal of High Performance Computing Applications*, vol. 17, no. 4, pp. 417–429, 2003.

[2] D. Anderson, S. Bowyer, J. Cobb, D. Gebye, W. Sullivan, and D. Werthimer, “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, Publisher: Bologna, Italy*, p. 729, 1997.

[3] M. Neary, A. Phipps, S. Richman, and P. Cappello, “Javelin 2.0: Java-Based Parallel Computing on the Internet,” in *Proceedings of European Parallel Computing Conference (Euro-Par 2000)*, 2000.

[4] C. Germain, V. Néri, G. Fedak, and F. Cappello, “XtremWeb: building an experimental platform for Global Computing,” in *Grid'2000*, December 2000.

[5] “Berkeley Open Infrastructure for Network Computing,” http://boinc.berkeley.edu/, 2002.

[6] E. Caron, F. Desprez, F. Lombard, J. Nicod, M. Quinson, and F. Suter, “A Scalable Approach to Network Enabled Servers,” in *Proceedings of the 8th International EuroPar Conference*, ser. Lecture Notes in Computer Science, B. Monien and R. Feldmann, Eds., vol. 2400. Paderborn, Germany: Springer-Verlag, August 2002, pp. 907–910.

[7] H. Casanova and J. Dongarra, “NetSolve: A Network-Enabled Server for Solving Computational Science Problems,” *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 3, pp. 212–223, 1997.

[8] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka, “Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing.” *J. Grid Compututing*, vol. 1, no. 1, pp. 41–51, 2003.

[9] R. Buyya, D. Abramson, and J. Giddy, “Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid,” *High-Performance Computing, ASIA, China, IEEE CS Press, USA*, vol. 01, no. 1, p. 283, 2000.

[10] L. Barroca, J. Hall, and P. Hall, *Software Architectures: Advances and Applications*. Springer Verlag, 1999, ch. An Introduction and History of Software Architectures, Components, and Reuse. [Online]. Available: http://mcs.open.ac.uk/lmb3/introduction.pdf

[11] M. D. McIlroy, “Mass Produced Software Components,” in *Software Engineering*, P. Naur and B. Randell, Eds. Brussels: Scientific Affairs Division, NATO, 1969, pp. 138–155.

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

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

[14] “Fractal ADL Tutorial,” http://fractal.objectweb.org/tutorials/adl/, 2004.

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

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

[17] E. Bruneton and T. Coupaye and J.B. Stefani, “The Fractal Component Model, version 2.0-3,” ObjectWeb consortium,,” Technical report, Feb. 2004.

[18] “ObjectWeb: Open source Middleware,” http://www.objectweb.org/.

[19] L. Baduel, F. Baude, and D. Caromel, “Efficient, flexible, and typed group communications in java,” *Joint ACM Java Grande - ISCOPE 2002 Conference*, pp. 28–36, 2002.

[20] P. Felber and R. Guerraoui, “Programming with object groups in CORBA,” *IEEE Concurrency*, vol. 8, no. 1, pp. 48–58, 2000.

[21] M. Aleksy and A. Korthaus, “A CORBA-based object group service and a join service providing a transparent solution for parallel programming,” in *PDSE*, 2000, pp. 123–134.

[22] S. Maffeis, “Run-time support for object-oriented distributed programming,” University of Zurich,” PhD Thesis, February 1995.

[23] L. Moser, P. M. Melliar-Smith, and P. Narasimhan, “A fault tolerance framework for corba.” in *Proceeding of the 29th Symposium on Fault Tolerant Computing, FTCS-29*, June 1999, pp. 150–157.

[24] P. Felber, R. Guerraoui, and A. Schiper, “The implementation of a corba object group service,” *Theory and Practice of object Systems*, vol. 4, no. 2, pp. 93–105, 1998.