Distribution of a Hierarchical Component in a Non-Connected Environment

Didier Hoareau, Yves Mahéo

Distribution of a Hierarchical Component in a Non-Connected Environment

Didier HOAREAU, Yves MAHÉO
Valoria - Université de Bretagne Sud, France
{Didier.Hoareau,Yves.Maheo}@univ-ubs.fr

Abstract

This article addresses the utilisation of the component-based approach for building and executing a distributed application that can offer services over a set of heterogeneous and possibly volatile devices. We propose to rely on a hierarchical component model and present a method for distributing a component that makes the set of its interfaces available everywhere. The runtime system associated with this model allows disconnections of devices to be reflected on the architecture of the component by disabling some of its interfaces while allowing it to perform in a degraded mode.
keywords*: hierarchical components, distributed components, context-awareness, disconnections, Fractal.*

1. Introduction

Mobile devices such as PDAs, laptops or mobiles phones are becoming more and more popular. When used together with traditional workstations, they can form new distributed platforms for which new distributed applicative needs emerge. One of the challenges we are faced is to be able to implement distributed applications that take into account the heterogeneous and dynamic characteristics of this kind of equipment. Indeed, designing and deploying applications in this context requires to consider software and hardware specificities of the various devices as well as to support the (more or less important) dynamic characteristics of the runtime environment, which are namely induced by the volatility of the hosts and the connections.

Existing component models [11, 12, 13] and more generally the component approach have proved their interest in designing, implementing and maintaining distributed applications. They allow an application to be designed as a set of interconnected components accessible through well defined interfaces, the whole forming a potentially complex architecture. But most of the existing models and their associated middleware support have been designed for business client/server applications targeting traditional networks of workstations. They often rely on rather strong assumptions on the stability of the execution platform (*e.g.* permanent availability of a server component) and suppose that each of the hosts offers sufficient resources. In general, an application designed this way cannot be installed or execute on networks of potentially volatile hosts with sometimes limited resources.

This article addresses the utilisation of the component-based approach for building and executing an application that should offer services over a set of mobile devices with possibly limited hardware and software resources. The different parts of the application do not necessarily reside on all the hosts but on each of them, the application should be usable. A degraded functioning mode may take place if all the parts of the application are not accessible.

In this perspective, we propose to use a hierarchical component model in which a component can be itself an assembly of components (it is then called a *composite* component). A common way to distribute a component-based application consists in installing each component instance on a host; the distribution then refers to the fact that a component can make distant invocations to the services implemented by another component. As for us, we propose to take advantage of the hierarchical structure of the component architecture and to allow a composite component to be accessible on several hosts although each of its subcomponents is instantiated on a single machine. In this context, the volatility of the devices and the connections eventually results in temporary failures of some bindings between subcomponents. To avoid that the entire application becomes unusable, the runtime support implements mechanisms that detect these failures and make inactive some of the component's interfaces. The other interfaces remain active so that the component can continue to perform part of its functions. The runtime support we have built provides for introspection on the fact that an interface is active or not, thus allowing the development of applications that can adapt to disconnections.

This article is structured as follows: we first present the hierarchical component model we have adopted and detail how a hierarchical component can be distributed. Secondly, the notion of active interface is taken up and we precise how it is implemented in the support of our hierarchical distributed component model. The next section gives some details on the prototype we have built based on the Fractal model. Finally we conclude by mentioning some related works and the future research directions we envisage.

2. Towards a distributed hierarchical component

2.1. Hierarchical component

In a hierarchical component model, composite components provide a uniform view of an application across different abstraction levels. Hence, a composite component represents a more or less complex structure of interconnected components –described by a configuration stored in an architecture descriptor– and thus can be used as a simple component with well-defined required and provided interfaces. Recursion stops with primitive components that correspond to computing units. Components are interconnected through bindings that represent each a link (local or distant) between a required interface and a provided interface.

The notion of composite component is often used at design time and is found in so-called architecture description languages (ADL) [10]. In the applicative framework we have chosen, it is however interesting to also be able to manipulate a composite at execution time. Indeed, it eases the provision for dynamic adaptation mechanisms whose aims are to allow subcomponents to be added, withdrawn or replaced, and bindings between components to be redefined. The hierarchical structure is then a means to take into account several abstraction levels at execution time.

When compared with other models of hierarchical components like Darwin [7] or Koala [14], Fractal [3] offers most of the characteristics we need. We will rely on a subset of this hierarchical component model if the following.

The Fractal model defines a component as being composed of two parts: a membrane (or controller) and a content. The membrane exposes the interfaces (required and provided) of the component and intercepts all the invocations at the interface level. A composite component is defined as a component exposing its content, *i.e.* a set of subcomponents. To each required and provided interface of a composite corresponds the interface of a subcomponent.

Composition in Fractal is done through bindings between required an provided interfaces and through the presence of components inside a composite. The model is recursive: a composite component can itself appears in the content of a composite component.

The initial configuration of the components is described in an architecture descriptor written in Fractal ADL. Standard features of Fractal ADL allow the user to specify namely the details (name, type, implementation,...) concerning the primitive and composite components and the bindings between components. New modules may be added to this ADL to express other aspects related for example to the architecture (*e.g.* inheritance, attributes) or to the deployment (such as the placement directives mentioned below).

A number of control interfaces (as opposed to functional interfaces) are provided in order to introspect an reconfigure the internal features of the component, for example to perform the dynamic addition and withdrawal of bindings between components. The user may add new such interfaces to handle specific reflection mechanisms.

Figure 1 shows the architecture of an application composed of a composite component that contains two primitive subcomponents. Two control interfaces are depicted that are responsible of controlling actions respectively on the subcomponents and on the bindings bewteen subcomponents.

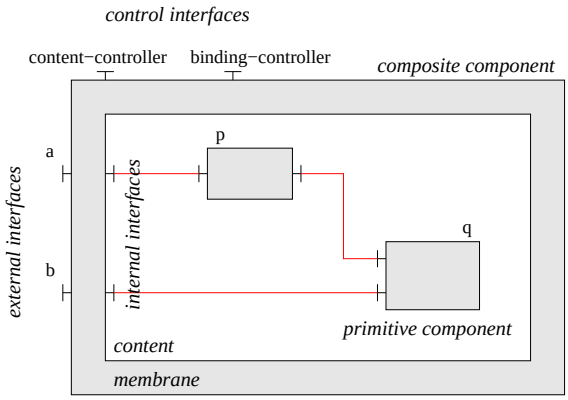

Figure 1. Internal view of a Fractal component.

2.2. Distribution of a hierarchical component

As mentioned in the introduction, we wish to deploy a hierarchy of components on a distributed platform that is characterized namely by its heterogeneity and its volatility. The application components will be distributed on a set of hosts. The placing of the components is performed while taking account of the hardware and software resources available on each host. The way this placement is initially chosen and how it evolves is one of the aspects of

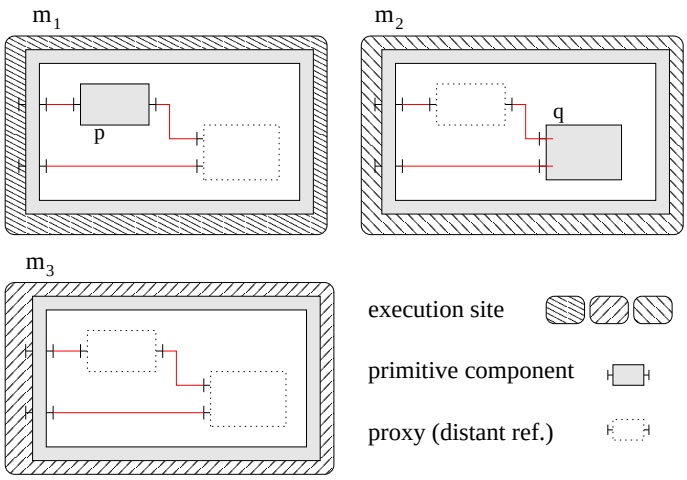

Figure 2. Distribution of a composite component on three hosts.

our project but is not detailed in this paper. We will focus on describing the mechanisms allowing a distributed execution of hierarchical components.

In our approach, the architecture of a component is coupled to its placement and this relationship is dealt with differently for composite components than for primitive components. As far as distribution is concerned, a primitive component executes on one host whereas a composite is possibly physically distributed on a set of different hosts. This mainly allow the services implemented by a composite to be invoked on more than one host.

A single instance corresponds to primitive composite. On the contrary, an instance of composite may be replicated. In a context of dynamicity and therefore as new hosts may appear, the target set of this replication should not be considered as fixed. However, for simplification reasons, we will consider here that it is explicitly defined in the architecture descriptor. In this descriptor, a single host is associated with a primitive component whereas a set of hosts is associated with a composite component. The set of hosts associated with a given composite component must be a subset of the set of hosts associated with the including component. If no set is specified, the placement set is inherited from the including component.

A composite component c is distributed over a set of hosts $\mathbb{M}$ if it exists on every host of $\mathbb{M}$ an instance of c. All the instances of c are created according to the directives found in the architecture descriptor. At execution time, each instance of c maintains locally the configuration of its subcomponents. Hence, a distributed composite component c distributed over $\mathbb{M}$ respects the following properties:

- The provided and required interfaces of c are accessible on all the hosts m_i of $\mathbb{M}$. These interfaces are those defined in the architecture descriptor.
- let c be a composite component that contains a single *primitive* subcomponent p. It exists a single host m_i on which p executes. For every host $m_j \in \mathbb{M}$ ($j \neq i$), it exits c_j, an instance of c on m_j. Each c_j holds a distant reference to p.

Each instance that represents the same composite is locally responsible for the architectural configuration of the composite. Figure 2 carries on with the example of the composite component described in Figure 1. It depicts its distribution on three hosts. If, during execution, host m_1 becomes unaccessible from m_2 and m_3, component p cannot be referenced any longer. However, for m_2 and m_3, binding to q is still possible. On m_2 and m_3, one can still access to the services offered by q through interface b. Indeed, the interfaces of the component are available on each host. Invoking methods on these interfaces still poses problems when the subcomponents are not accessible. Indeed, instantiating the composite on all the hosts makes possible the use of its provided interfaces on each host. However, due to network disconnections, on a given site, an access to a distant primitive component can be interrupted. Consequently, an invocation method in this case is likely to raise some kind of network exception.

This problem is not specific to our approach but appears as soon as distant references are used, that may point to unaccessible components at any time. In order to prevent invocations that may not complete because of disconnections, we introduce the notion of active and non active interfaces. We propose to add a control interface to components to allow introspection on the state (active or not) of its provided and required interfaces.

3. Active Interfaces

The life-cycle of a component is significantly tied to the one of the components it requires: for a component to be activated (*i.e.* it is safe to invoke any method defined in its interfaces) it must have all its required interfaces bound. As in a dynamical environment disconnections have to be taken into account, this approach can be severely disadvantageous. We may not want a component to be deactivated even if all of its (required) interfaces are not bound. In this case the component still offers a service but in a degraded mode.

Figure 3 shows an assembly of components that points out the two parts of its architecture that are independent. Dependencies between required interfaces and provided interfaces for composite component **A** are represented by a graph. So, if a primitive component of graph grA becomes unavailable, we can still make safe method invocations on interface a_2. Interface a_2 is being kept active whereas a_0 and a_1 have to be *deactivated* to prevent failures of method invocations on these provided interfaces.

Our previous example introduces the notion of active interface. Indeed the activity of an interface represents its state regarding method invocations. An *active* interface delegates a method invocation to the component owning this interface.

A provided interface is active if it is bound to a provided interface which is also active. A provided interface is active if:

- for a primitive component: all its required interfaces are active;
- for a composite component: the provided interface of the corresponding subcomponent is active.

Inactive interfaces are defined by taking the negative form of the previous clauses. We can notice that the deactivation of only one required interface of a primitive component makes all of its provided interface inactive. This is due to the lack of *a priori* explicit information to describe dependencies between provided interfaces and required interfaces within a primitive component. To some extent, this problem could be alleviated by declarative features or by code analysis.

Consequently to our characterization of the state of components, the state of a composite component is independent of the one of its required interfaces. It is by the analysis of the hierarchical structure of the architecture that the dependencies are revealed. Indeed, the activation or the deactivation of an interface is propagated through the architecture. In a dynamical environment, disconnections will affect the component structure by inducing breaks of binding. Conversely, the appearance of a component of the architecture (after a disconnection) will result in the creation of some binding(s) and thus the activation of some interface(s).

We can use the hierarchical structure of our model to update the state of the architecture regarding interfaces. When a primitive component activates or deactivates its provided or required interfaces, this information is propagated to the corresponding enclosing composite component. The latter, which has a global view of its sub-architecture (in terms of bindings between subcomponents) is in charge of activating or deactivating the corresponding interfaces of the involved components.

We can also use active interfaces in the reconfiguration of an application by replacing the implementation of a component by an other one. We have to isolate the part of the architecture affected by this modification in order to perform the substitution at runtime. The technique that consists in stopping the composite component which contains the subcomponent to be substituted may be disadvantageous because dependencies between provided and required interfaces are not taken into account. For example, the substitution of component **H** of Figure 3 requires that the provided interfaces of **H** (h_0) be deactivated. This will gradually cause the deactivation of d_0 (component **D**) and c_0 (component **C**). We can thus call methods on interface c_0 which is still active.

4. Implementation

4.1. Extension of the Fractal component model

We have implemented a middleware support for hierarchical distributed components by extending Julia [3], a Java implementation of the Fractal component model. Active interfaces have been realized thanks to the addition of a new controller to the primitive and composite components. As mentioned in section 2.1, controllers in the Fractal component model represent non functional operations (functional interfaces are directly bound to the business code of the components). The specification [3] defines some control interfaces to manage the bindings between components, subcomponents etc. Our new controller is in charge of maintaining up-to-date the state of the required and provided interfaces. It also allows the activation and the deactivation of any interface of the component. Propagation within the architecture is based on the dependencies between interfaces, which are explicitely declared with the `binding` clause in the Fractal architecture descriptor.

The controller prevents method invocations on the inactive interfaces of the component. To do so, method reification has been performed by using the interceptor mechanisms of Julia (`MetaCodeGenerator`). If an interface is inactive, the current strategy consists in stopping the method calls during a parameterizable amount of time and in throw-

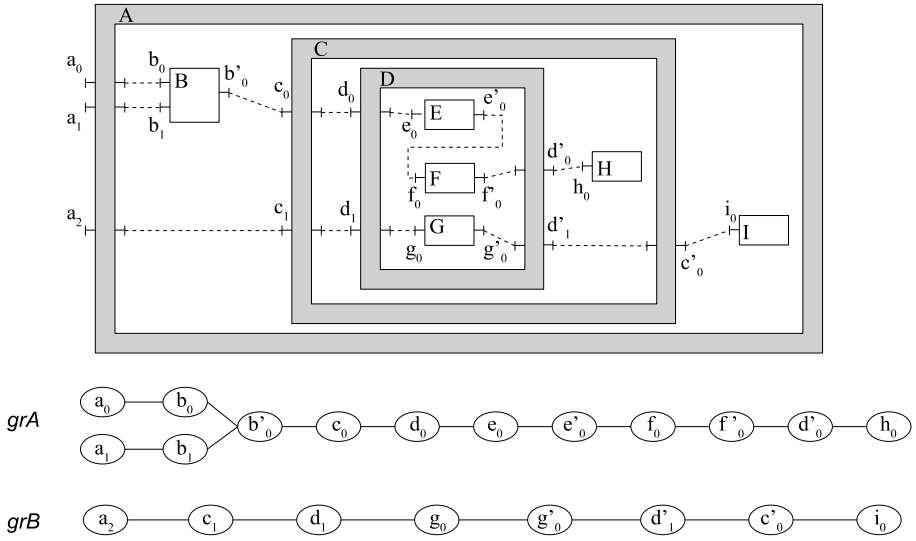

Figure 3. Dependencies between provided and required interfaces within a composite component. Graphs grA and grB emphasize the sub-architecture that can be isolated.

ing an exception if the interface has not been activated after this time. This default behaviour can be modified, as illustrated in the next section.

The support for managing active and inactive interfaces relies on the mixin mechanisms offered by Julia that allow code insertion in the membrane of the component, according to the aspect oriented programming approach. Thus it is possible to take into account this kind of interface in any application implemented with Julia, independently of our execution platform. The components are then endowed with an API for discovering the state of the interfaces (active or not) and the dependencies between interfaces.

The appearance of components results in an automatic reconfiguration of the bindings between components like in [4]. Due to interception mechanisms, the propagation time for the deactivation of an interface in the hierarchy is linear with the depth of the hierarchy. We have improved the default behaviour of our prototype by adding to each membrane the dependencies between provided and required interfaces. So, when a required interface becomes inactive for a composite component, we can directly exploit the list of the provided interfaces which have to be deactivated. For example, if component **I** of figure 3 becomes unavailable, interface c_0' is deactivated but instead of propagating this information to component **C**, we can directly deactivate interface c_1. Although this optimization is beneficial in complex architectures, we have now to manage methods calls that have not been captured between the moment we deactivate a required interface and its corresponding provided interface.

4.2. Context-awareness

Disconnections have been taken into account thanks to D-RAJE (*Distributed Resource-Aware Java Environment*) [8], an extensible Java-based middleware developed in our team. D-RAJE makes it possible to model and to monitor resources in a distributed environment. With this middleware, hardware resources (*e.g.* processor, memory, network interface...) or software resources (process, socket, thread, directory...) can be modeled in an homogeneous way. Like in a previous work on parallel components [9], D-RAJE has been extended in order to model components and bindings as resources. It is thus possible to discover the existence of a component, to look for a specific component and to collect information on the state of a component and its interfaces. Information can be obtained by direct observation or by being notified after subscription to interesting events concerning specific components.

When a physical disconnection or a reconnection occurs (*i.e.* at the network level), the bindings between components as well as the state of the interfaces are modified accordingly. For this purpose, we use the D-RAJE `NetworkLink` resource that models the physical link between two hosts and maintains information about the state of the network connection. We have added the `RemoteBinding` resource that models the binding beetween components. Each `NetworkLink` resource is related to one or more `RemoteBinding` resources. D-RAJE monitors are created on `NetworkLink` resources so that the state of the `RemoteBinding` re-

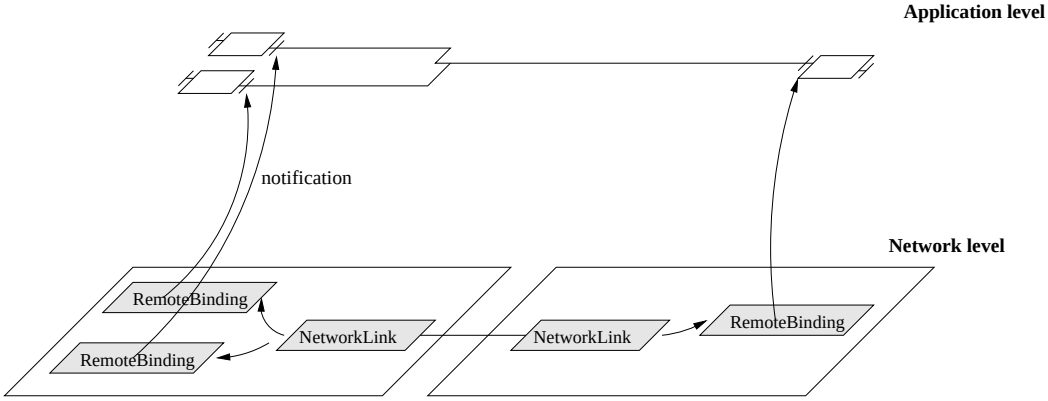

Figure 4. Management of remote bindings with resources in D-RAJE.

sources are modified when the corresponding network link fails. The event on the change of a `RemoteBinding` resource is captured with the result that the Fractal binding is withdrawn and the corresponding interfaces are deactivated. Conversely, when a monitor on a `NetworkLink` resource detects that the network becomes up again, it notifies the `RemoteBindings` resources. As previously, this results in reestablishing the broken Fractal bindings an activating the corresponding interfaces.

In D-RAJE, a naming system gives any resource a unique identity. This allows us to identify each instance of a distributed composite component. So, when a network connection becomes available, our system can discover all the instances of the same composite component and reestablish a binding if needed.

Figure 5 illustrates how we can use our API to manage strategies according to the state of the interfaces. In this example, the programmer decides to wait for the reconnection before invoking a method on an inactive interface.

```
// Obtain a reference on our controller
CubikController kcontroller
      =(CubikController)comp.getFcInterface(
                              "cubik-controller");

// Get the state of the client interface
// named "cltItf"
if (kcontroller.getFcItfState("cltItf")==false) {

  // Manual management of the disconnection
  RemoteBinding rb =
    kcontroller.getRemoteBinding("clientItf");
  while(!rb.getState()) {
   // Waiting for the interface activation
  }
  // Now we can do the invocation
  clientItf.something();
}
```

Figure 5. Introspection on the state of an interface.

5. Related works

Our project is focused on providing facilities to support distributed component-based applications that can adapt themselves. Among other works sharing this goal, few consider the use of a hierarchical distributed component model and the management of disconnections at the same time.

Our approach has some similarities to [5, 1]. These works aims at providing general adaptation mechanisms applied to hierarchical component-based applications. More specifically, in [6] adaptation to disconnections is discussed but for the Corba Component Model [13] which is a flat component model. In order to guarantee continuity of services, the authors have introduced « disconnected components » whose role are to perform the logging of methods calls and reconciliation when disconnections occur. It is not envisaged that services are maintained (even in a degraded mode) when disconnections occur.

The notion of hierarchical components physically distributed over several hosts is also discussed in [2] where an implementation of the Fractal component model, using the Proactive library, is presented in order to provide Grid components. Although they are not explicitly dealt with, disconnections are taken into account to some extent thanks to the use of Proactive asynchronous communications between components.

The Gravity project [4] is based on a service-oriented component model. In this approach, the bindings between components are considered as dependencies between services (like in the service oriented approach). So, automatic adaptation can be done within a component composition according to the services' availability. The availability of one or more services required by a component results in valid

or invalid instances. The current development of the Gravity project does not consider distributed components.

6. Conclusion

In this paper we have presented the utilization of a hierarchical component model for building and executing an application distributed over a set of heterogeneous and possibly volatile devices. We take advantage of the hierarchical model in the distribution of the components as well as in the management of disconnections.

Our model rely on the Fractal hierarchical component model. We have defined an operating scheme to execute a Fractal composite component that is physically distributed over several machines. In this scheme, the interface of a composite component are available on a set of machines, even if its primitive subcomponents are not replicated, thus allowing for the heterogeneity of the target platform.

The architecture of a composite component is handled in a decentralized way in order to ease the management of network disconnections. These disconnections are reflected on the architecture as widthdrawal of bindings. The Fractal hierarchical component model has been extended with the notion of active interface in order to isolate dependencies between required and provided interfaces. A network disconnection triggers the break of the component bindings on which they rely and consequently the deactivation of the related interfaces. Lastly, interface deactivations are propagated within the component hierarchy. The interfaces that remain active allow a component to perform in a degraded mode. Reconnections are dealt with along the same line, ending up with the activation of some component interfaces.

We have realized a prototype and integrated it into Julia, a Java implementation of the Fractal component model. Our mechanisms in charge of supporting disconnections have been added as a non-functional service into Julia, endowing it with an introspection interface through which one may discover the state of each interface in order to build strategies of adaptation to disconnections.

The work presented in this paper is the first stage of the development of a middleware supporting adaptive components, that is components capable of reconfigure themselves according to changes occurring in their runtime environment.

We plan to integrate adaptation strategies from information on the state of the application architecture at runtime. In particular, our current work focuses on the adaptive aspects of the deployment of a hierarchical component. Indeed such a deployment could be done in a continuous way during the execution of the application according to resources availability.

References

[1] A. Andersen. *OOPP, A Reflective Middleware Platform including Quality of Service Management*. Dr. sci. thesis, Department of Computer Science, University of Tromsø, Tromsø, Norway, Feb. 2002.

[2] F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, LNCS, Catania, Italy, Nov. 2003.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An Open Component Model and its Support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE7)*, number 3054 in LNCS, Edinburgh, Scotland, May 2004.

[4] H. Cervantes and R. S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 614–623, Edinburgh, Scotland, May 2004.

[5] P.-C. David and T. Ledoux. Towards a Framework for Self-Adaptive Component-Based Applications. In *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003*, LNCS, pages 1–14, Paris, France, Nov. 2003.

[6] N. Kouici, D. Conan, and G. Bernard. Caching Components for Disconnection Management in Mobile Environments. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, Agia Napa, Cyprus, Oct. 2004.

[7] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC)*, pages 137–153, Sitges, Spain, Sept. 1995.

[8] Y. Mahéo, F. Guidec, and L. Courtrai. A Java Middleware Platform for Resource-Aware Distributed Applications. In *Proceedings of the 2nd International Symposium on Parallel and Distributed Computing (ISPDC'2003)*, pages 96–103, Ljubljana, Slovénie, Oct. 2003. IEEE CS.

[9] Y. Mahéo, F. Guidec, and L. Courtrai. Middleware Support for the Deployment of Resource-Aware Parallel Java Components on Heterogeneous Distributed Platforms. In *30th Euromicro Conference - Component-Based Software Engineering Track*, pages 144–151, Sept. 2004.

[10] N. Medvidovic and N. R. Taylor. A Classifi cation and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), 2000.

[11] Microsoft. DCOM Technical Overview. Microsoft Windows NT Server white paper, Microsoft Corporation, 1996.

[12] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 1999.

[13] Object Management Group. Corba components. Adopted Specifi cation formal/02-06-65, OMG, June 2002. Version 3.0.

[14] R. C. van Ommering. Koala, a Component Model for Consumer Electronics Product Software. In *ESPRIT ARES Workshop*, pages 76–86, Las Palmas de Gran Canaria, Spain, Feb. 1998.