Constraint-Based Deployment of Distributed Components in a Dynamic Network

Didier Hoareau, Yves Mahéo

Constraint-Based Deployment of Distributed Components in a Dynamic Network

Didier Hoareau and Yves Mahéo

Valoria Lab – University of South Brittany
{Didier.Hoareau|Yves.Maheo}@univ-ubs.fr

Abstract. Hierarchical software components offer interesting characteristics for the development of complex applications. However, supporting the deployment of such applications is difficult, especially on challenging distributed platforms. This paper addresses the distribution and the deployment of hierarchical components on heterogeneous dynamic networks. Such networks may include fixed and mobile resource-constrained devices and are characterized by the volatility of their hosts and connections, which may lead to their fragmentation. The distribution scheme and the associated mechanisms we propose allow a component to provide its services in an ubiquitous way and to operate in a degraded mode. The deployment of hierarchical components is described: we present an ADL extension for specifying a context-aware deployment and we detail a hierarchically-controlled deployment designed for dynamic networks. This deployment is performed in a propagative way and is driven by constraints put on the resources of the target hosts.

1 Introduction

Distributed platforms are no longer restricted to stable networks of workstations. One of the archetype of a distributed platform is now a network that may comprise stable and powerful workstations but also a number of mobile and resource-constrained devices. Although this kind of distributed platform is increasingly common, it remains a challenging target for building, deploying an maintaining distributed applications. Specific techniques must be applied to cope with the heterogeneity of the hosts and links as well as with the dynamism of the network. What we call dynamic networks are especially a difficult target, as hosts may become unaccessible because of their mobility or their volatility. As a consequence, one cannot rely on models and algorithms designed for fully connected networks. A dynamic network is rather described as a partitioned network, viewed as a collection of independent islands. An island is equivalent to a connected graph of hosts that can communicate together, while no communication is possible between two islands. In addition, the configuration of the islands may change dynamically.

Figure 1 shows an example of such a dynamic network. It is composed of a number of hosts a user has access to and on which a distributed application is meant to be accessible. This set of hosts includes fixed and mobile machines. Connectivity is not ensured between all the hosts. Indeed, at home, the user's connection to Internet is

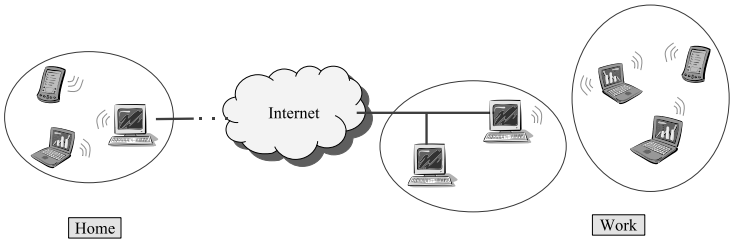

Fig. 1. Example of a dynamic network, possibly partitioned in three islands

sporadic and some of the devices are mobile (as such, they may become out of reach) and/or volatile (a PDA may for example be switched off frequently).

The applications that are to be executed on dynamic networks can be inherently complex. This complexity is even increased by the need to produce code that can adapt to the changes of the execution environment. Since a few years, the use of software components proved to be useful for developing complex distributed applications and many component models and their associated technologies are now available. In the component-based approach, the application is designed as an assembly of reusable components that can be bound in a versatile manner, possibly dynamically. Some of the proposed models are known as hierarchical models. They offer the possibility of creating high level components by composing components of lower abstraction level, which represents a software construction principle that is natural and expressive. In such models, a component –that is then called a composite component– can be itself an assembly of components, recursive inclusion ending with primitive components that encapsulate computing code.

Using a hierarchical component-based approach for building an application that targets a dynamic network seems an attractive solution. Yet, several problems remain that are not dealt with available component models and component execution supports. In particular, the two following aspects have to be dealt with: (1) how to deploy a hierarchical component in a dynamic network while ensuring that this deployment respects the architecture of the application and adapts itself to the resource constraints imposed by the target platform? (2) how to allow a distributed execution of the components, i.e. to allow interactions between components in a not-always-connected environment? This paper describes a distribution scheme of hierarchical components and its associated deployment process that target dynamic networks. Because of the very constrained environment in which the application is to be deployed, we can hardly envisage a permanent access to the services offered by the application or an optimal utilization of the resources. The emphasis is put on finding a distribution scheme and some deployment mechanisms that achieve a minimal availability while taking account of the environment.

Outline of our approach

The distribution scheme we propose is related to the hierarchical structure of the application. This scheme is based on the replication of composite components. Indeed, we allow a composite to be accessible on a set of hosts, although each primitive component is localized on a single host. Besides, we also allow a component to operate in a degraded mode in order to account for network disconnections without making the entire application unusable. The notion of *active interface* is added to the component model. Our runtime support detects network disconnections and deactivates some components' interfaces accordingly. Introspection on the state (active or inactive) of an interface is possible so as to allow the development of adaptive components.

The deployment of a component covers several parts of the life-cycle of a component. In this paper we focus on the last phases of the deployment, covering the instantiation of the component (that creates an executable instance from a component code), its configuration (that establishes the bindings to its interfaces) and its activation (that allows the other components to invoke its interfaces). The presented techniques should be complemented with component delivery mechanisms such as those described in [1].

The deployment of the hierarchy of components is specified in a constraint-based declarative way. The architecture descriptors of the components are augmented with deployment descriptors in which constraints on the resources required by components and on their possible location can be specified.

When the deployment is triggered, all the constraints listed in the deployment descriptor may not be satisfied immediately. The dynamism of the network makes the situation even more difficult as it may occur that the set of hosts that would satisfy globally the deployment constraints are never connected together at the same time, precluding any deployment. Instantiation of some components and their activation is however possible as we allow the components to operate in a degraded mode through the dynamic management of interfaces' activation. The deployment process we implement is thus a propagative process : the instantiation and the activation of a component are performed as soon as some resources that meet its needs are discovered. We propose an algorithm that supports this propagative deployment. The scalability of the process is ensured by the distributed and hierarchical organisation of the control. Moreover, we implement a distributed consensus that guarantees that the location constraints are satisfied even in the context of a partitioned network.

The paper is organised as follows. First, the model of hierarchical component we work on is presented and we explain how a hierarchy of components is distributed over a network. The concept of activation at the interface level is briefly exposed. In section 3 we give some details on the form of the deployment descriptor that complements the architecture description. Section 4 presents the overall propagative deployment process and details the distributed instantiation algorithm that forms the basis of the distributed deployment. Section 5 briefly describes the status of the development of our prototype. Finally, we cite the related works before concluding.

2 Distributed Hierarchical Components

We describe in this section what we understand by distributed hierarchical components, through the description of the basic features of our component model and of the way we have chosen for distributing the components over a network of hosts. Further details can be found in [2].

2.1 Hierarchical Component Model

In this paper, we consider a widely applicable hierarchical component model in which a composite component represents a more or less complex structure of interconnected components that 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 each represents a link between a required interface and a provided interface. For practical reasons, we have chosen to base our development on the Fractal component model [3] and more precisely on its reference Java implementation Julia. However, the concepts developed in this paper could easily be applied to other hierarchical component models such as Koala [4], Darwin [5] or Sofa [6].

The notion of composite component is often used at design time and is found in so-called architecture description languages (ADL) [7]. In the applicative framework we have chosen, it is however interesting to also be able to manipulate a composite at execution time in order to ease dynamic adaptation. Therefore the composite is reified at runtime namely by a membrane object that stores the interfaces of the component and its configuration (the list of its subcomponents and the bindings between these subcomponents).

2.2 Distribution Model

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 the volatility of its hosts. The application components are distributed on a set of hosts. The way this placement is performed is detailed in section 4. We focus here on the description of 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 can be physically replicated on a set of different hosts. The main goal of composite replication is that the component's interfaces become directly accessible on several hosts. A composite component can then be seen as providing a ubiquitous service.

A single host is associated with a primitive component whereas a set of hosts is associated with a composite component. This set must be a subset of the set of hosts associated with the including component. By default, the placement set of a composite component is inherited from the including component.

At execution time, each instance of a composite component maintains locally information about the configuration of its subcomponents. Hence, a distributed composite component c distributed over a set of hosts $\mathbb{H}$ respects the following properties:

- The provided and required interfaces of c are accessible on all the hosts h_i of $\mathbb{H}$.
- Let c be a composite component that contains a *primitive* subcomponent p. There exists a single host h_i on which p executes. For every host $h_j \in \mathbb{H}$ ($j \neq i$), there exits c_j, an instance of c on h_j. Each c_j holds a remote reference to p (in a proxy).

2.3 Example

We give in this section an example of an application made of hierarchical components and we detail how it can be distributed on a given set of hosts.

Figure 2 depicts the architecture of a photo application that allows the user to search for a number of photos in a repository and to build a diaporama with the selected photos. The top-level composite component (*PhotoApp*) includes a generic component devoted to document searching (*DocumentSearch*). This component is also a composite component (taken off the shelf); it is composed of a *DocumentFinder* and a *DocumentBuffer*. The primitive *DocumentFinder* component provides an interface for issuing more or less complex requests based on the names of documents, on their subjects or some other meta-information, and for selecting the corresponding documents from a given set of documents (a repository). The selected documents are passed to a *DocumentBuffer*. Apart from an interface for adding new documents, the primitive *DocumentBuffer* component provides an interface for sorting and extracting documents. This provided interface and the one of *DocumentFinder* are accessible as provided interfaces of the *DocumentSearch* component. Finally, the *DocumentSearch* component is bound to a *PhotoRepository* component that constitutes the specialized document repository and a *DiapoMaker* component which allows the selected photos to be assembled in a parameterizable diaporama.

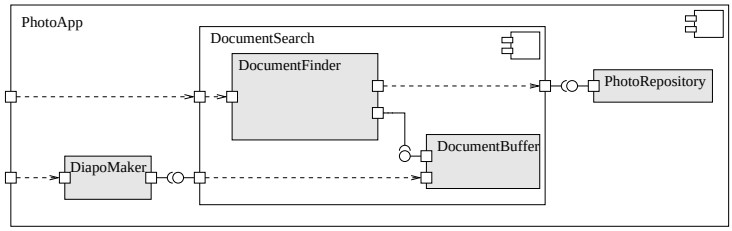

Fig. 2. Architecture of the photo application (in UML 2.0)

Consider that the photo application is meant to be usable from any of the five machines owned by the user (hosts h_1 to h_5), in a dynamic network similar to the one depicted in figure 1. Hence, the target set of hosts associated with the *PhotoApp* component is $\{h_1, h_2, h_3, h_4, h_5\}$. A subset of these hosts is dedicated to the distributed

execution of the composite component *DocumentSearch*, say $\{h_1, h_2, h_3\}$, h_4 and h_5 being excluded for licence reasons for example. Moreover, some constraints on the required resources result in the following placement of the primitive components (see section 4 for details): *DocumentFinder* on h_1, *DocumentBuffer* on h_2, *PhotoRepository* on h_4 and *DiapoMaker* on h_5.

At runtime the membranes of the composite components are maintained on each of their target hosts. A membrane contains the interfaces of the component as well as the description of its architecture (subcomponents and bindings). The instances of components (primitive or composite) that are not present are represented by proxies. Note that for a primitive component, the proxy is linked to the distant (single) instance of this primitive whereas for a composite component, the proxy is linked to one distant instance of the (partially replicated) membrane.

Figure 3 summarizes the placement of the components and shows the runtime entities (architectural information and instances) maintained on h_1 and h_4 for our PhotoApp example.

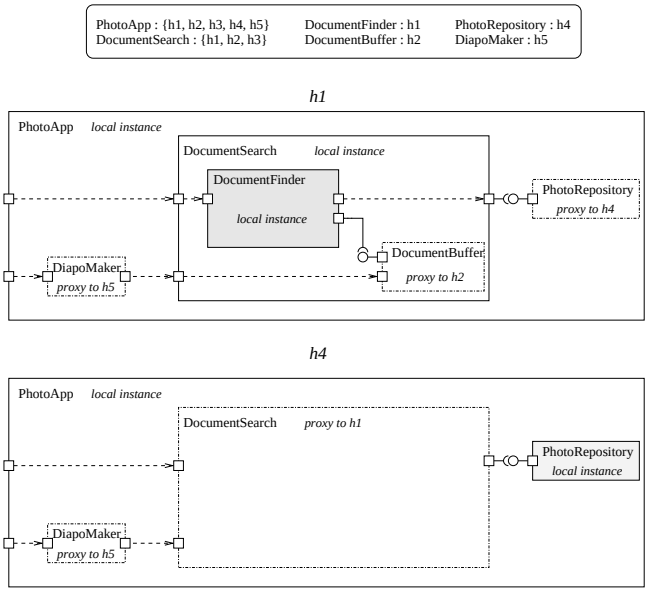

Fig. 3. Placement of components and entities maintained on host h_1 and h_4

2.4 Support for disconnections

The replication of a composite component eases the access to the services it implements as it makes possible to use its provided interfaces on each host. However, because of network disconnections, from a given site, access to a remote component can be interrupted. Consequently, a method invocation in this case may raise some kind of a

network exception. This problem is not specific to our approach but appears as soon as remote references are used, that may point to unaccessible components at any time. In a context of hierarchical components, the technique that consists in deactivating a component as soon as one of its required interface is unbound is very penalizing as a single disconnection will end up by ricochet with the deactivation of the top-level component, that is the deactivation of the entire application. In the dynamic environments we target, where disconnections are frequent, the application is likely to be rarely usable.

We address this problem in the following two ways:

- We introduce the notions of active and non active interfaces. We maintain the state (active or not) of an interface according to the accessibility of the component's instance it is bound to. Moreover, we add a control interface to components to allow introspection on the state of its provided and required interfaces.
- We allow the execution of a component even if some of its interfaces are not active.

On the PhotoApp example, if a disconnection occurs between h_1 and h_4, the *PhotoRepository* component is no longer accessible from h_1. The disconnection is detected by a dedicated monitor, and consequently, the required interface of the *DocumentSearch* component is deactivated. This triggers the deactivation of the corresponding required interface of the *DocumentFinder* and then of its provided interface. However, the second interface of *DocumentSearch* (the one bound to *DiapoMaker*) can remain active as the *DocumentSearch* component is still accessible. Globally the application is still usable, although in a degraded mode, as diaporamas can still be built from the document buffer.

Notice that this approach has an obvious impact on the programming style required when developing components, as the state of an interface should be tested before invoking methods on this interface. Indeed, the uncertainty of the accesses to needed (or required) services –inherent to the targeted dynamic platforms– enforces adaptable code. The provision for tools to introspect on the availability of the interfaces is a minimal answer that should be complemented by other facilities for describing or applying, for example, adaptation strategies. This involves research at language level and middleware level that is out the scope of the presented work.

3 Deployment Specification

When considering the deployment of distributed components, the key issue is to build a mapping between the component instances and the hosts of the target platform. This task implies to have some knowledge not only about the identity of the hosts involved in the deployment phase, but about the characteristics of each of them as well. Moreover, for a hierarchical component-based application, every component instance at each level of the hierarchy has to be handled.

At design-time, the designer is unlikely to know where to deploy each component regarding resource availability. This motivates the need to differ this task at runtime. We propose to add a deployment aspect to an existing architecture description language (such as [8, 9]).This will allow the description of the resource properties that must be satisfied by a machine for hosting a specific component.

We propose an extension to ADLs that makes possible the description of the target platform in a declarative way. The language we propose is purely declarative and descriptive and has a similar objective to the language described in [10]. It is not mandatory to give an explicit name or address of a target machine: the placement of components are mainly driven by constraints on the resources the target host(s) should satisfy. The choice of the machine that will host a component will be made automatically at runtime (during the deployment).

The description of the resources that the target platform must satisfy is defined in a deployment descriptor in which references to component instances (defined in the architecture descriptor) can be made. For each component, a *deployment context* is defined. Such a context lists all the constraints that a hosting machine has to satisfy. If these constraints are associated with a primitive component, one host will be authorized to instantiate this component whereas several hosts may be selected for hosting the membrane of a composite component, in accordance with our distribution model.

There are two types of constraints that can be defined in a deployment context: resource constraints (*ResCst*) and location constraints (*LocCst*). *ResCst*s allow hardware and software needs to be represented. Each of these constraints defines a domain value for a resource type that the target host(s) should satisfy. *LocCst*s are useful to drive the placement choice of a component if it occurs that more than one host is applicant.

```
<component name="DocumentSearch">
  <component name="DocumentFinder">
    <deployment-context>
      <resource-constraint>
      <cpu freq="1.2"
           unit="GHz"
           operator="min" />
      </resource-constraint>

      <location-constraint>
         <target varname="x"/>
      </location-constraint>
    </deployment-context>
  </component>

  <component name="DocumentBuffer">
    <deployment-context>
       <resource-constraint>
          <memory free="200"
                  unit="MB"
                  operator="min" />
       </resource-constraint>
       <location-constraint>
          <target varname="y"/>
       </location-constraint>
    </deployment-context>
  </component>

  <deployment-context>
    <location-constraint>
      <operator name="alldiff">
         <arg varname="this.DocumentFinder.x" />
         <arg varname="this.DocumentBuffer.y" />
      </operator
    </location-constraint>
  <deployment-context>
</component>
```

(a)

```
<component name="PhotoApp">
  <component name="DiapoMaker">
    <deployment-context>
       <cpu freq="1.5" unit="GHz"
            operator="min" />
       <memory free="50"
               directory="/home/"
               unit="MB"
               operator="min"/>
    </resource-constraint>
  </deployment-context>
  </component>

  <component name="PhotoRepository">
    <deployment-context>
      <resource-constraint>
        <memory free="1" unit="GB"
                directory="/home/"
                operator="min" />
      </resource-constraint>
    </deployment-context>
  </component>

  <component name="DocumentSearch">
      <locationconstraint>
        <operator name="exclude">
          <arg value="egilsay" />
          <arg value="parvati" />
        </operator>
      </locationconstraint>
  </component>

  <deployment-context>
    <locationconstraint>
      <target hostname="ambika"/>
      <target hostname="dakini"/>
      <target hostname="mafate"/>
      <target hostname="egilsay"/>
      <target hostname="parvati"/>
  <deployment-context>
</component>
```

(b)

Fig. 4. Deployment descriptor

An example of use of *ResCst* and *LocCst* is illustrated in Figure 4 which shows the deployment descriptor, in an XML notation, of the photo application introduced in the previous section. The descriptor **(a)** contains the constraints associated with the *DocumentSearch* composite component and descriptor **(b)** contains those of the *PhotoApp* component. Resource constraints are defined within the *resource-constraint* element. For every component, adding an XML tag corresponding to a resource type (e.g. cpu, memory) specifies a constraint on this resource the target host has to verify.

Location constraints are declared within the *location-constraint* element. The *target* element defines the set of hosts among which our runtime support will have to choose. Hosts can be represented in two ways: (1) by their hostname if their identity are known before the deployment or (2) by a variable. Such a variable can be used at the composite level to control the placement of components. This feature is achieved by the use of the *operator* element. This element allows relations between variables to be expressed. For example, in descriptor **(a)**, the *DocumentFinder* component is said to be deployed on host *x* and *DocumentBuffer* on host *y*. Constraining *DocumentFinder* and *DocumentBuffer* to be on two distinct hosts is achieved by using the *alldiff* operator that declares *x* to be different from *y*. For a primitive component, at most one variable can be declared (because a primitive component will be placed on an unique host). Several variables can be used for a composite component, which is physically distributed over several hosts.

When composing the application, it is possible to use only variables. Then, the definition of the target platform is made at the first level of the hierarchy (for the component *PhotoApp* on the example) by adding the list of the machines that will be involved in the deployment (lines 71–75 on Figure 4). During the deployment, as it is detailed in next section, this set of machines, together with the location constraints will be inherited by the sub-components.

4 Deployment Process

4.1 Overview of the Propagative Deployment

Because of the dynamism of the network on which we deploy our applications, it is not possible to base a deployment on a full connection of the different hosts. We are interested in a deployment that will allow an application to be activated progressively, that is, part of its provided services can be put at disposal even if some machines that are required for the "not yet" installed components, are not available. As soon as these machines become connected, the deployment will go along. Moreover, resource changes on any host may yield the deployment of components although is was not possible before. The deployment we present in this paper is thus asynchronous as it may not be possible to deploy every component immediately.

Once the architecture descriptor and the deployment descriptor have been defined, the deployment phase we consider in this article consists in choosing a target host for every component of the architecture. This selection has to be made according to the deployment context associated with every component. Indeed, the selected machine has to satisfy all the resource constraints and this machine must not contradict the location constraints. In the case of a primitive component, a single host has to be selected among

several applying machines. For a composite component, the number of applicants can be greater than the set of machines over which this composite component has to be distributed. Controlling the selection of the target hosts is essential to guarantee the consistency of the application. Indeed, in a dynamic network where islands of machines may appear, we must avoid inconsistent decisions in two different islands. For example, we have to ensure that two distinct machines from two islands will not be selected for the hosting of the same primitive component.

In the following we present the general propagative deployment algorithm in two steps. First, we consider a fully connected environment. This will help us to focus on the resolution of the constraints expressed in the deployment descriptor and to describe a possible distribution of the instantiation tasks (thus the selection of target hosts) within a hierarchical architecture. Then we present the complete propagative deployment in a dynamic network where the main difficult aspect is to ensure a unique decision regarding component instantiation.

4.2 Deployment in a Connected Network

We consider in this section a fully connected network composed of a finite set of machines. Each machine is identified and no disconnection may occur. The propagative deployment in such a network consists in diffusing the architecture descriptor and the deployment descriptor to all the machines that are listed at the top level of the application (with the XML *target* element).

Then, once a machine has received these descriptors, a recursive process is launched in order to select the components that can be hosted on this machine. The main steps of this process for a machine m_i, for a component C are the following:

1. machine m_i checks if it satisfies the location constraints associated with C. This corresponds to verifying if m_i belongs to the set of the target hosts (see XML `target` element). If the m_i is not concerned by the deployment (instantiation) of component C, the process returns for this component, else,
2. machine m_i has to launch probes corresponding to the resource constraints of C (e.g. `CPUProbe`, `MemoryProbe`). If at least one probe returns a value violating a resource constraint (e.g. not enough free memory available), the process returns, else,
3. machine m_i declares itself as an applicant host for component C and a collective decision has to be made. Indeed, more than one host may apply and if C is a composite component, it may have subcomponents with location constraint such as $x \neq y$
4. once a choice has been made, all the applicants are informed of the value of the free variables and of the fact that they are authorized (or not) to instantiate component C. The process stops for hosts that are not authorized. For the others,
5. if C is a composite component, the process is performed recursively for all the subcomponents of C

In a connected network, there is no difficulty to make the collective decision described in point 3. We could for example choose before the deployment a machine S whose role is to decide a host among applicants. In this case, when a host h satisfies all

the constraints attached to a component and thus becomes applicant, *h* announces itself to S and waits for a decision. However we prefer the approach of [11] where a deployment controller, which is in charge of well-defined tasks of the deployment, is defined for each composite component of the hierarchy. The main reason for such a distribution of the deployment controllers is scalability. Indeed, with this approach, parts of the application can be deployed independently according to its topology. Thus, we define a machine S_i per composite component of the hierarchy. This machine is responsible for the decision-making of its direct sub-architecture, *i.e.* it must choose among applicants those who don't contradict the location constraints. Applicants must be informed of the results. Thus, after a decision, each representative S_i sends to the applying hosts a new deployment descriptor which is updated with the new location information, *i.e.* the actual name of the machine hosting a specific component is added to the location constraints. Indeed, before the deployment, no explicit machine name is given and variables can be used to indicate applicant machines. For example, if the machines *ambika* and *dakini* are respectively attributed to components *DocumentFinder* and *DocumentBuffer*, the following lines are added to descriptor **(a)**:

```
// replace line 11 by:
<target varname="x" value="ambika"/>
```

```
// replace line 23 by:
<target varname="y" value="dakini"/>
```

As a consequence of the decision, some components can be instantiated. In the case of composite components, the deployment process (local evaluation, applicant announcement, decision-making) goes along recursively.

However, it may be possible that a representative machine could not find any placement solution (because no combination of applicants fullfills all the location constraints). In order to propose to the representative a new possible placement, a machine that newly satisfies some resource constraints (for uninstalled components) declares itself as applicant.

4.3 Deployment in a Dynamic Network

The previous section, with the assumption of a fully connected network, has highlighted two main ideas of the propagative deployment: (1) each host does the evaluation of the constraints attached to the components and (2) the decision making is distributed over several machines, each of them representing a composite component of the application.

In a dynamic network all the machines may not be connected at the same time. In this kind of network, islands may exist and communication paths between machines may disappear. In such an environment, a deployment based on a full connection of the different machines at the same time is not conceivable. We may want to start the deployment (*i.e.* the instantiation of parts of the components and thus to put parts of the services offered by the application at disposal) while some machines may be disconnected or inaccessible. The component model presented previously (see section 2) allows an application to run in a degraded mode but the main difficulty here is to deal with the unicity of the instantiation of the–possibly statefull–components, which is difficult to ensure in a dynamic network. Indeed, we must avoid conflicting decision to be made in the different islands that may exist in such an environment. On one hand, a machine that represents a composite component, cannot be selected before the deployment, as in a fully connected network, since this machine may not be connected. On

the other hand, if we let each of the machines that host the same replicated composite component make a decision, we cannot guarantee that in two different islands contradictory instantiations may not be performed. We tackle this difficulty by considering the consensus problem: a set of machine has to decide on a same value regarding the representative of composite components.

We use the results of [12], in which the requirements of the consensus problem are relaxed. The authors have identified *conditions* for which there exists an asynchronous protocol that solves the consensus problem despite the occurrence of t process crashes. We define the consensus to select, for each composite that is replicated on several machines, a representative host that will make future decisions on where the direct subcomponents have to be instantiated.

Algorithm The main steps of the algorithm described in the previous section are not modified. The only change concerns the designation of a representative host for each node of the hierarchy. We use the algorithm of [12] to elect such a representative and to build a common view of the placement of the components.

The consensus algorithm requires that a majority of machines can be reached among the target hosts of the composite component. This majority is not the same depending on the composite component. For example, the photo application is distributed over $\{h_1, h_2, h_3, h_4, h_5\}$, as a consequence, the majority is reached when at least three of these machines are in the same island. Whereas for the composite `DocumentSearch` component, which is distributed over $\{h_1, h_2, h_3\}$, the consensus is possible when an island, composed of at least $\{h_1, h_2\}$, $\{h_1, h_3\}$ or $\{h_2, h_3\}$, is formed.

The consensus-based algorithm consists in:

1. ensuring that the selection of a host for a representative composite component is possible if an island is composed of a majority of machines,
2. selecting a machine S_i for the future instantiation decisions for each composite component of the hierarchy
3. updating the deployment descriptor with the identifier of the selected machine.

Points 1 and 3 guarantee that if a new island composed of a majority of machines is created, there is at least one machine that possesses the most recent version of the deployment descriptor. Thus no contradictory decision can be made in this island.

The consensus may not terminate (e.g. the number of hosts within an island may not be sufficient). In order to prevent this situation, we allow a newly connected machine to participate in this consensus. This is achieved by periodically broadcasting a message asking if a consensus is still in progress. In that case the newly connected machine collects the data that have already been exchanged between the other machines and proposes a value that can make the consensus evolve.

Once a representative composite component is chosen, due to the dynamism of the network, this composite may be in a non-majority island during a more or less amount of time. In this case, if an instantiation decision is made, it cannot decide any more whereas it may exists an other island in which a consensus can be reached. Thus, if such a decision has to be made and a majority of machines composes the islands, a new representative machine is selected and the deployment descriptor is updated. No conflict

will arise later, *i.e.* when the older representative belongs to a new majority islands. Indeed in such an island, it exists at least one machine that possesses the most recent version of the deployment descriptor, thus during the consensus, the older representative will learn the existence of the new one.

5 Implementation Status

The propagative deployment presented in this paper is based on a constraint based-language for the description of the placement of components according to resource requirements. Our current prototype has been implemented using Julia, a Java implementation of the Fractal component model [3]. The features of the ADLs described in section 3 have been implemented as new modules into the Fractal Architecture Description Language. Deployment descriptors can be specified graphically through an extension of FractalGUI.

In order to evaluate the constraints defined in the deployment descriptor, we have to collect information about resources on every host. We use D-Raje [13] –a framework developed in our team, dedicated to the observation of distributed system resources in Java– to define specific probes related to resource constraints. D-Raje is also used to model and to monitor the state of physical links between hosts. A disconnection can then be captured with the result that bindings between components are withdrawn and the corresponding interfaces are deactivated. Further details can be found in [2].

We are currently implementing a distributed test platform in order to tune our consensus algorithm considering parameters such as the numbers of hosts and the frequency of disconnections.

6 Related work

The main aspects developed in this paper are related to a distribution scheme for hierarchical components on dynamic networks and to a resource-aware and propagative deployment.

Many works have taken into account a context-aware deployment, that is, the placement of components onto hosts according to some resource requirements. A formal statement of the deployment is given in [14] and a set of algorithms that improve mobile system's availability is presented. In [15] the authors propose a deployment configuration language (DCL) in which properties on the target hosts can be expressed. The deployment considered in this work extends the Corba Component Model, which is a flat component model. In [10], the authors present the Deladas language that also allows constraints to be defined on hosts and components. A constraint solver is used to generate a valid configuration of the placements of components and reconfiguration of the placement is possible when a constraint becomes inconsistent. But this centralized resolution is not suited to the kind of dynamic network we target. Moreover, the current version of Deladas does not consider resource requirements. These abovementioned works aim at finding an optimum for the placement problem of components. This aspect is not one of our objectives. Indeed, due to the dynamism of the environment, it

is hardly feasible to define a quiescent state that will allow our consensus algorithm to decide on an optimal placement. Moreover, the solutions proposed are centralized.

In [16] a decentralized redeployment is presented. The configuration to be deployed is available on every host involved in the deployment. A local decision can then be made according to the local subsystem configuration state. However the choice of the components' location is made before the deployment process. The works presented in [11] on the deployment of hierarchical component-based applications is probably the closest to ours'. The authors describe an asynchronous deployment and use the hierarchical structure of the application in order to distribute deployment tasks. In the solution developed by the authors, a deployment controller is statically chosen and defined in the deployment descriptor. In our approach we could not decide at design-time which machine will host such a controller. The approach proposed by the authors focuses on functional constraints and thus resource requirements have not been taken into account.

7 Conclusion

This paper has presented a support for deploying and executing an application built with hierarchical components on an heterogeneous and dynamic network. The main contribution of this work is that it attempts to take into account a challenging distributed target platform characterized by the heterogeneity and the volatility of the hosts, volatility that may result in the fragmentation of the network.

A distribution method has been proposed for hierarchical components. Composite components are made accessible on a set of hosts whereas each primitive component is localized on a single host. Besides, via the notion of active interface, we allow a component to operate in a degraded mode in order to account for network disconnections without making the entire application unusable.

We have presented a purely descriptive language for specifying deployment descriptors that allow for a context-aware deployment. This language is meant to extend some existing ADL. A deployment descriptor allows the description of the resource needs of a component and some placement constraints.

The deployment process we have defined is a propagative one. The instantiation and the activation of a component is performed as soon as some resources that meet its needs are discovered. This early activation is possible as some of its interfaces can remain inactive (the component then executes in a degraded mode).We have designed an algorithm that supports this propagative deployment in a dynamic network. The scalability of the process is ensured by the distributed and hierarchical organisation of the control. Moreover, we have presented a distributed consensus that guarantees that the location constraints are satisfied even in the context of a partitioned network.

The main direction of our future work consists in taking into account the possible modifications on the resources' availability after some component instantiations have been made. Indeed, even if we can respect for example a memory constraint on the instantiation of a given primitive component, the memory conditions may change that invalidates the choice afterwards. The mechanisms we have implemented in our deployment algorithm could be adapted for solving this problem, provided the component can be safely stopped. An autonomic deployment could thus be defined.

References

1. Roussain, H., Guidec, F.: Cooperative Component-Based Software Deployment in Wireless Ad Hoc Networks. In: 3rd International Working Conference on Component Deployment (CD 2005). LNCS, Grenoble, France (2005)
2. Hoareau, D., Mahéo, Y.: Distribution of a Hierarchical Component in a Non-Connected Environment. In: 31th Euromicro Conference - Component-Based Software Engineering Track, Porto, Portugal, IEEE CS (2005)
3. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: 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 (2004)
4. van Ommering, R.C.: Koala, a Component Model for Consumer Electronics Product Software. In: ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain (1998) 76–86
5. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying Distributed Software Architectures. In: Proceedings of the 5th European Software Engineering Conference (ESEC), Sitges, Spain (1995) 137–153
6. Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In: Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDS '98), Annapolis, Maryland, US (1998)
7. Medvidovic, N., N. Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Trans. Software Eng **26** (2000) 70–93
8. : xacme: Acme extensions to xarch. School of Computer Science Web Site: http://www-2.cs.cmu.edu/ acme/pub/xAcme/ (2001)
9. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: An infrastructure for the rapid development of xml-based architecture description languages. In: In proceedings of the International Conference on Software Engineering (ICSE'02), Orlando, Florida, USA (2002) 266–276
10. Dearle, A., N. C. Kirby, G., J. McCarthy, A.: A framework for constraint-based deployment and autonomic management of distributed applications. In: ICAC. (2004) 300–301
11. Quéma, V., Balter, R., Bellissard, L., Féliot, D., Freyssinet, A., Lacourte, S.: Asynchronous, hierarchical and scalable deployment of component-based applications. In: Proceedings of the 2nd International Working Conference on Component Deployment (CD'2004), Edinburgh, Scotland (2004)
12. Mostéfaoui, A., Rajsbaum, S., Raynal, M., Roy, M.: Condition-based consensus solvability: a hierarchy of conditions and efficient protocols. Distributed Computing **17** (2004) 1–20
13. Mahéo, Y., Guidec, F., Courtrai, L.: A Java Middleware Platform for Resource-Aware Distributed Applications. In: 2nd Int. Symposium on Parallel and Distributed Computing (ISPDC'2003), Ljubljana, Slovenia, IEEE CS (2003) 96–103
14. Mikic-Rakic, M., Medvidovic, N.: Software architectural support for disconnected operation in highly distributed environments. In: CBSE. (2004) 23–39
15. Li, T., Hoffmann, A., Born, M., Schieferdecker, I.: A platform architecture to support the deployment of distributed applications. In: ICC, IEEE International Conference on Communications. Volume 4. (2002) 2592–2596
16. Mikic-Rakic, M., Medvidovic, N.: Architecture-level support for software component deployment in resource constrained environments. In: Component Deployment. (2002) 31–50