Middleware Support for Ubiquitous Software Components

Didier Hoareau, Yves Mahéo

Middleware support for the deployment of ubiquitous software components

Didier Hoareau and Yves Mahéo
Valoria, University of South Brittany
Campus de Tohannic, 56017 Vannes, France
{Didier.Hoareau|Yves.Maheo}@univ-ubs.fr

Abstract A number of emerging distributed platforms include fixed and robust workstations but, like dynamic and pervasive networks, are often built from mobile and resource-constrained devices. These networks are characterized by the volatility of their hosts and connections, which may lead to network fragmentation. Although increasingly common, they remain a challenging target for distributed applications. In this paper we focus on component-based distributed applications by addressing the distribution and the deployment of software components on dynamic pervasive networks. We present a distribution scheme and some associated middleware mechanisms that allow a component to provide its services in an ubiquitous way. First, an architecture description language extension is proposed in order to specify a deployment driven by constraints on the resources needed by components. Then, a propagative and autonomic deployment process is explained, which is based on a consensus algorithm adapted for dynamic networks. Lastly, implementation details and experiment results are given.

1 Introduction

1.1 Dynamic pervasive platforms

During the last years have emerge new distributed platforms, often qualified as pervasive, that are no longer restricted to an interconnection of workstations that forms a stable network. These platforms may still include powerful and robust machines but they are rather composed of resource-contrained and mobile devices (laptops, personal digital assistants–or PDA–, smartphones, sensors, etc.). Due to the mobility and the volatility of the devices involved, dynamism is one of their major characteristics. A dynamic network hence formed can be 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.

In this paper, we are interested in medium-size dynamic pervasive platforms. Figure 1 shows a simple 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 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).

1.2 Ubiquitous applications

Although this kind of distributed platform is increasingly common, it remains a challenging target for building, deploying and maintaining distributed applications. The pervasiveness of the equipment should be reflected on the distributed application, leading to some form of ubiquitous applications. Many applications should benefit from ubiquity in this context: enhanced classical applications such as PIM (Personal Information Management) or collaborative applications, but also envisioned applications in e-home or e-business. A ubiquitous application is supposed to render its services everywhere, or at least wherever it makes sense, accounting for the constraints of the hosting devices. For example a PIM application is much more usable if

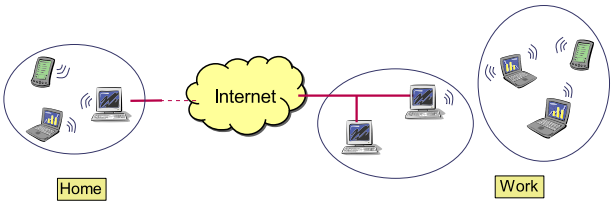

Fig. 1: Example of a dynamic network, possibly partitioned in three islands

it offers its services on all the machines owned by a user, even if the entire application is not installed on each machine. It is not desirable however that the application be designed and administered as a collection of target-specific codes. Ubiquity must be made as transparent as possible. Of course, it may occur that some of the services are temporarily not available on a specific host (eg access to an up-to-date shared agenda from a PDA that is isolated from any network). In addition, some functionality may not be accessible everywhere due to a lack of resources (eg extended graphical view on a device with a small display). We believe that a minimal set of mechanisms should be provided to implement this adaptation in order to reduce the complexity of the design and the administration of ubiquitous applications.

1.3 Ubiquitous components

Software components have 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 itself be an assembly of components, recursive inclusion ending with primitive components that encapsulate computing code.

Using a hierarchical component-based approach for building a ubiquitous application that targets a dynamic network seems an attractive solution. Yet, several problems remain that are not treated by 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, ie to allow interactions between components in a not-always-connected environment?

1.4 Outline of our approach

This paper describes a distribution scheme for hierarchical components and its associated deployment process that target dynamic pervasive 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.

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. Moreover, as it may occur that resources needed by an already deployed component become not sufficient, the placement choice for a component can be called in question dynamically. The deployment process can thus be considered as autonomic. We propose an algorithm that supports this propagative and autonomic 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. In section 2, 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, we present the overall propagative and autonomic deployment process, and we detail the distributed instantiation algorithm that forms the basis of the distributed deployment. Section 4 briefly describes the status of the development of our prototype. After discussing related work in section 5 we conclude the paper in section 6.

2 Distributed Hierarchical Components

We describe in this section what we understand by distributed hierarchical components. The basic features of our component model are explained and we detail how the components are distributed 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 (ie 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 3.2. 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 some information about the configuration of its subcomponents. Hence, a distributed composite component c distributed over a set of hosts H respects the following properties:

- The provided and required interfaces of c are accessible on all the hosts h_i of 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 $h_j \in H$ $(j \neq i)$, 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 the 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.

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 3.2 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 every host for our *PhotoApp* example.

2.4 Support for disconnections

The replication of a composite component eases the access to the services it implements as it permits the use of 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

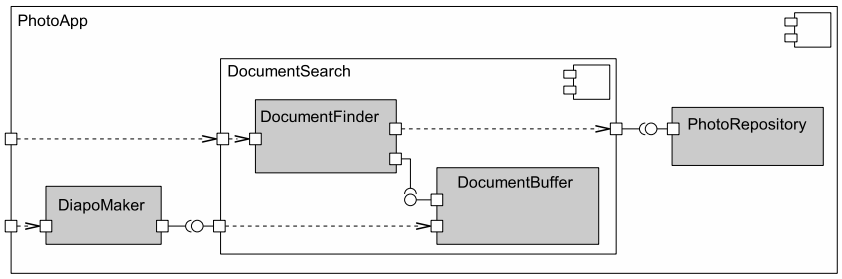

Fig. 2: Architecture of the photo application (in UML 2.0)

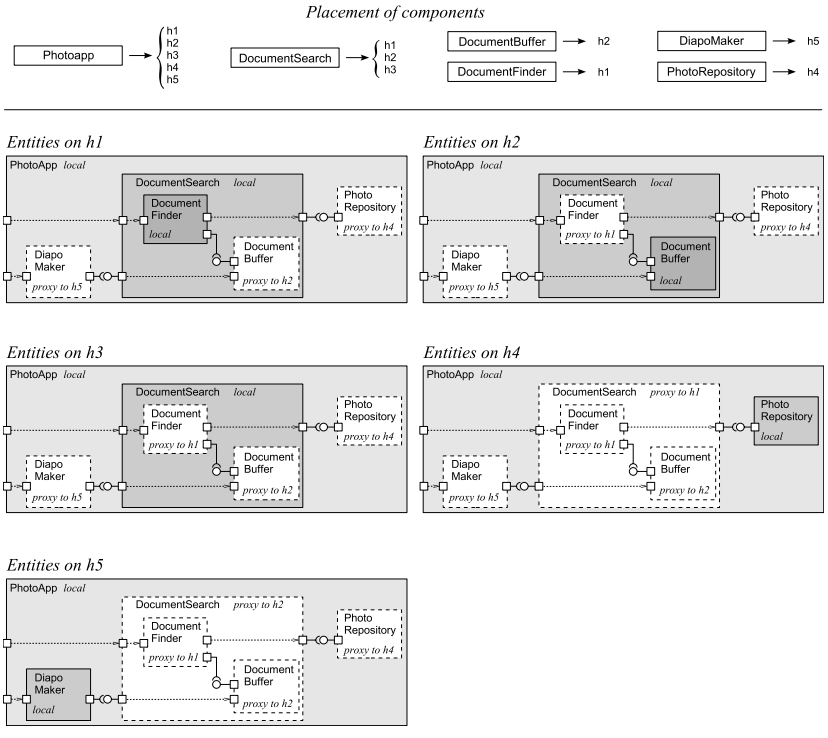

Fig. 3: Placement of components and entities maintained on hosts h_1 to h_5

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

3.1 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 also about the characteristics of each of them. Moreover, for a hierarchical component-based application, every component instance at each level of the hierarchy has to be handled.

At design-time, it is unlikely that the designer knows 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 xAcme[1] or [8]). 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 [9]. 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.

[1] xAcme: Acme Extensions to xArch, http://www-2.cs.cmu.edu/~acme/pub/xAcme/

Two types of constraints can be defined in a deployment context: *resource* constraints and *location* constraints. Resource constraints 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. Location constraints are useful to drive the placement choice of a component if it occurs that more than one host is candidate.

An example of use of resource and location constraints is illustrated in Figure 4 which shows the deployment descriptor, in an XML notation, of the photo application introduced in the previous section. 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 (eg 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. A variable name can be used at the composite level to control the placement of the components. This feature is achieved by the use of the *operator* elemen, which 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 the next section, this set of machines, together with the location constraints, will be inherited by the subcomponents.

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

(b)

Fig. 4: Deployment descriptor

3.2 Deployment process

3.2.1 Overview

When the architecture descriptor and the deployment descriptor are defined, the deployment phase we consider in this article consists in choosing one (or several) target host(s) for every component of the architecture. This selection has to be done in accordance with the deployment context associated with the components: the target hosts must satisfy the resource constraints and must not contradict the location constraints. Depending on the resources that are available on the machines of the network, more than one machine can be chosen for hosting a component: for a primitive component, only one host has to be selected whereas for a composite component, according to our distribution scheme, several hosts can be chosen. It is required to control the placement of components. Indeed, we have to guarantee that two islands of machines do not make inconsistent decisions (eg instantiating twice the same primitive component).

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 host. 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, the progression of the deployment is guaranteed not only thanks to the accessibility of a new connected machine but also because of resource changes on any host. This deployment is therefore qualified as *propagative*.

However, in the kind of dynamic network we target, when a component is installed and instanciated, the resources it requires may also disappear or become unavailable. A *redeployment* is then mandatory. The autonomic deployment consists in reconsidering the placement choices that have been made in the propagative phase in order to take into account the unavailability of resources.

The main difficulty of such a deployment in a pervasive network is to guarantee the unicity of the instantiations defined in the architecture descriptor. On one hand, a host 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.

In the following, we present the autonomic deployment in two steps. First, we detail the propagative deployment, then, we present the mechanisms that make this deployment autonomic.

3.2.2 Propagative deployment

When the deployment is launched from an initial machine, the deployment descriptor and the architecture descriptor are diffused to all the machines that are listed at the top level of the application (with the XML *target* element). Then, each machine that receives these descriptors, launches a recursive process (ie for each composite component) in order to select the components that can be deployed (instantiated) locally. The main steps of this process for a host h_i and for a composite component C are the following:

1. h_i checks if it belongs to the set of the target hosts associated with C (see the XML *target* element). If h_i is not concerned by the deployment (instantiation), the process returns for this component, else,
2. host h_i launches probes corresponding to the resource constraints of every subcomponent of C (eg a probe for memory observation). For each subcomponent for which the probes have returned a compatible value with regard to the resource constraints, h_i declares itself as candidate for hosting this component.
3. h_i also receives other candidatures. As soon as h_i has computed a solution in function of these candidatures, it tries to make it adopted via a consensus algorithm.
4. Once the consensus has completed, ie a majority of machines has decided (or not) to confirm the placement solution of h_i, this piece of information (which contains the values of the free variables) is sent to the other machines (and therefore to the other applicants) which will stop the process for each component they are not authorized to instantiate, else,
5. For each subcomponent that can be instantiated on h_i, the process starts again at step 1.

Since resources may fluctuate (eg become available and unavailable), discovery mechanisms (step 2) are used periodically. Moreover, it may be possible that no solution exists (step 3), that is, no combination of candidatures satisfies the location constraints. Periodic observation of resources allows a machine to apply for the instantiation of a specific (not installed yet) component as soon as its resource constraints are verified, potentially allowing the emergence of a new solution for the location constraints.

The propagative deployment requires a distributed algorithm in order to make a collective decision (step 3). This is achieved thanks to the use of a consensus algorithm on the identity of the machines that apply for the instantiation of a component. This algorithm is detailed in the next section.

The placement information is diffused to other machines (step 4) by updating the deployment descriptor with the new values, ie the names of the machines that are selected for hosting each component. Indeed, before the deployment, the location of a component can be defined without any knowledge on the identity of a specific host through the use of variables. For example, if hosts *ambika* and *dakini* are chosen respectively for the *DocumentFinder* and *DocumentBuffer* component, the following lines are modified in the deployment descriptor:

```
// replace line 12 by:
<target varname="x" value="ambika"/>

// replace line 24 by:
<target varname="y" value="dakini"/>
```

3.2.3 From a propagative deployment to an autonomic deployment

Principle The propagative deployment allows a component-based application to be deployed as soon as its required resources become available. But, in general, and especially in the kind of network we target, resources can also become unavailable (eg the amount of free memory demanded may decrease and become not sufficient) and faults may happen. In these cases, one or several components have to be redeployed. This redeployment can be divided into three steps:

1. Each of the components that depend on the unavailable resource is stopped, yielding the deactivation of its provided interfaces. All the (remote or local) required interfaces bound to these latter become inactive. Thus, all the interfaces leading to this component will be deactivated, one after the other. The application runs then in a degraded mode.
2. The state of the component is saved in a serializable form (we assume that the developer has anticipated this situation).
3. A message holding the identity of the component to redeploy is diffused. This message also con-

tains the location from which the state of the component(s) can be retrieved. Each machine that receives this message updates its deployment descriptor by removing the location of the component.

The above procedure is sufficient to define an autonomic deployment. Indeed, when receiving the message diffused at step 3, the machines—because they update their deployment descriptor—find themselves back in the propagative deployment: some components are not installed yet. Thus, because the deployment is not fully completed, the propagative deployment remains active, that is, some machines will apply for the instantiation of the uninstalled component. In our approach, the architecture descriptor of the application is viewed as a goal to achieve in terms of components' instantiations and with respect to some constraints to satisfy.

Consensus The propagative and autonomic deployment described above is based on a collective decision making algorithm. When several machines apply for the instantiation of the same component, and in order to avoid inconsistencies regarding the architecture descriptor, we have to guarantee that one and only one machine will be chosen. We use the consensus algorithm described in [10] to elect among applicants the machine whose identity will be approved by a majority of hosts. The authors of this algorithm have identified *conditions* for which there exists an asynchronous protocol that solves the consensus problem despite the occurrence of t process crashes. In our case, if there are n machines involved in the deployment, t can be as great as $\lfloor \frac{n}{2} \rfloor$. Thus, a collective decision making is possible if there is at least a majority of machines that compose the island. By relying on a majority we guarantee that within an island there is at least one machine that holds the latest version of the deployment descriptor, and so, no contradictory decision can be made in two distinct islands.

The consensus algorithm requires that the number of machines that are accessible among the target hosts of the composite component reaches the majority. 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 and h_5; as a consequence, the majority is reached when at least three of these machines are in the same island. Whereas for composite component *DocumentSearch*, which is distributed over $\{h_1, h_2, h_3\}$, the consensus is solved when an island, composed of at least $\{h_1, h_2\}$, $\{h_1, h_3\}$ or $\{h_2, h_3\}$, is formed.

Moreover, the consensus may not terminate (eg 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 the consensus. This is achieved by periodically broadcasting a message asking if a consensus is still in progress. In this 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.

4 Implementation status and results

4.1 Component distribution

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 (*cubik-controller*) to the primitive and composite components. This controller is in charge of maintaining up-to-date the state of the required and provided interfaces. The *cubik-controller* prevents method invocations on the inactive interfaces by reifying methods invocation (using the Julia MetaCodeGenerator). We propose an API to make possible the use of specific strategies when an interface is inactive: for example, one can wait for the reactivation of the interface.

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. It is thus possible to take into account this kind of interface in any application implemented with Julia without any code modification. The components are then endowed with an API for discovering the state of the interfaces (active or not) and the dependencies between interfaces.

4.2 Context-awareness

The deployment that has been presented in this paper relies on the discovery of the resources required by the components. Thanks to DRAJE (*Distributed Resource-Aware Java Environment*) [11], an extensible Java-based middleware developed in our team, hardware resources (eg processor, memory, network interface) or software resources (eg process, socket, thread, directory) can be modeled and observed in an homogeneous way. For every resource constraint of the deployment descriptor, a resource in DRAJE is created and a periodic observation is launched.

Moreover, DRAJE has been extended by adding two new types of resources: the RemoteBinding and NetworkLink resources. A NetworkLink resource models the physical link between two hosts and maintains some information about the state of the network connection. A RemoteBinding resource subscribes to a NetworkLink

in order to construct the state of a binding between two remote components. Thus, thanks to a simple notification system, when a disconnection (resp. reconnection) occurs at the network level between two machines, the state of the bindings is updated and the corresponding interfaces of components are deactivated (resp. activated).

4.3 Deployment resolution

The deployment process presented is based on a constraint language to describe the placement of the components according to some conditions on resources. This language is purely declarative. It has been implemented with FractalADL and is supported at run-time by a constraint engine developed with Cream[2]. Cream is a Java library for writing and solving constraint satisfaction problems or optimization problems. Thanks to this library, information about candidates and about the state of the local resources can be "told" to a store. This store is then used in order to get a location placement solution or to detect a constraint inconsistency (eg the amount of free memory required is no longer available).

4.4 Performance evaluation

The performance of the deployment process depends on multiple parameters imposed by the execution environment (disconnections, fluctuation of the resources, volatility of the hosts, etc.).

In a preliminary experiment, we have tried to isolate the impact of the implementation of our consensus from connectivity conditions. This experiment has been hence conducted on a (fully connected) 100 Mb/s Ethernet network of workstations (2 GHz Pentium 4). It dealt with the deployment of a component whose deployment descriptor is similar to the one of the *DocumentSearch* component described in section 3. Figure 5 shows the time taken by our algorithm to decide on a placement solution in function of the number of machines involved in the deployment. First (curve 1), we have limited to one the number of machines that apply for hosting a component. Then we have considered concurrent applicants, with 5 and 8 simultaneous candidatures (curves 2 and 3).

This experiment allowed us to verify that the time to obtain a placement solution remains acceptable and that the multiplicity of simultaneous consensus executions does not incur prohibitive overcost.

We are currently investigating the connection of our middleware support to a mobility simulator so as to emulate more realistic executions.

[2] http://kurt.scitec.kobe-u.ac.jp/~shuji/cream/

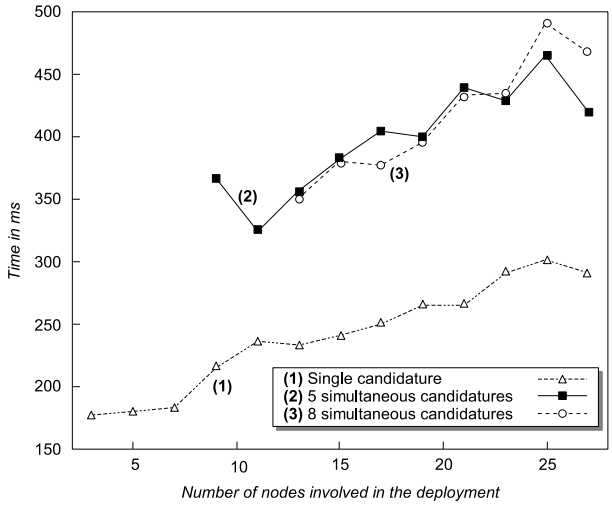

Fig. 5: Evaluation of the duration of a decision making on the placement of components

5 Related Work

The main aspects developed in this paper are related to a distribution scheme for hierarchical components on dynamic networks and to an automatic management of their deployment which is driven by constraints on resources that the machines of the network have to satisfy.

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 [12] and a set of algorithms that improve mobile system's availability is presented. In [13] 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 [9], 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 [14] 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 work presented in [15] deals with the deployment of hierarchical component-based applications. 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. Besides, the approach proposed by the authors focuses on functional constraints and thus resource requirements have not been taken into account.

Among the works on autonomic computing, [16] and [17] are based on autonomic entities—the components—to define autonomic systems. Changes in the environment are performed locally by every component that is responsible for its own reconfiguration, update, migration etc. However, the deployment of autonomic systems and the management of architectural consistency are not explicit.

6 Conclusion

This paper has presented a middleware support for deploying and executing an application built with ubiquitous 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 ubiquitous in the sense that they are made available 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.

Our proposal for supporting the deployment covers the last phases of the deployment process, namely the instantiation of the components' instances and their activation, which are handled through the individual activation of their interfaces. 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.

An algorithm allowing an autonomic deployment of a component-based application has been proposed. 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 because some of its interfaces can remain inactive (the component then executes in a degraded mode) and defines the propagative deployment phase. When a constraint attached to a component becomes inconsistent, its redeployment is performed automatically by going back to the propagative deployment phase. The autonomic deployment is based on a consensus algorithm in order to guarantee the consistency (in terms of components' instances) of the deployed architecture even in the context of a partitioned network.

References

[1] H. Roussain and F. Guidec. Cooperative Component-Based Software Deployment in Wireless Ad Hoc Networks. In *Proceedings of the 3rd International Working Conference on Component Deployment (CD 2005)*, volume 3798 of *LNCS*, pages 1–16, Grenoble, France, November 2005. Springer.

[2] D. Hoareau and Y. Mahéo. Distribution of a Hierarchical Component in a Non-Connected Environment. In *Proceedings of the 31th Euromicro Conference - Component-Based Software Engineering Track*, pages 143–150, Porto, Portugal, September 2005. IEEE CS Press.

[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 7th International Symposium on Component-based Software Engineering (CBSE7)*, volume 3054 of *LNCS*, pages 7–22, Edinburgh, UK, May 2004. Springer.

[4] R.C. van Ommering. Koala, a Component Model for Consumer Electronics Product Software. In *Proceedings of the 2nd International ESPRIT ARES Workshop*, volume 1429 of *LNCS*, pages 76–86, Las Palmas de Gran Canaria, Spain, February 1998. Springer.

[5] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, volume 989 of *LNCS*, pages 137–153, Sitges, Spain, September 1995. Springer.

[6] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDS'98)*, pages 43–51, Annapolis, Maryland, USA, May 1998. IEEE CS Press.

[7] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[8] E.M. Dashofy, A. van der Hoek, and R.N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 266–276, Orlando, Florida, USA, May 2002. IEEE CS Press.

[9] A. Dearle, G. N. C. Kirby, and A. J. McCarthy. A framework for constraint-based deployment and autonomic management of distributed applications. In *Proceedings of the International Conference on Autonomic Computing (ICAC'04)*, pages 300–301, New York, USA, May 2004. IEEE CS Press.

[10] A. Mostéfaoui, S. Rajsbaum, M. Raynal, and M. Roy. Condition-based consensus solvability: a hierarchy of conditions and efficient protocols. *Distributed Computing*, 17(1):1–20, 2004.

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

[12] M. Mikic-Rakic and N. Medvidovic. Software architectural support for disconnected operation in highly distributed environments. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE7)*, volume 3054 of *LNCS*, pages 23–39, Edinburgh, UK, May 2004. Springer.

[13] T. Li, A. Hoffmann, M. Born, and I. Schieferdecker. A platform architecture to support the deployment of distributed applications. In *Proceedings of the IEEE International Conference on Communications (ICC'02)*, volume 4, pages 2592–2596, New York, USA, April 2002. IEEE CS Press.

[14] M. Mikic-Rakic and N. Medvidovic. Architecture-level support for software component deployment in resource constrained environments. In *Proceedings of the 1st Working Conference on Component Deployment (CD 2002)*, volume 2370 of *LNCS*, pages 15–30, Berlin, Germany, June 2002. Springer.

[15] V. Quéma, R. Balter, L. Bellissard, D. Féliot, A. Freyssinet, and S. Lacourte. Asynchronous, hierarchical and scalable deployment of component-based applications. In *Proceedings of the 2nd International Working Conference on Component Deployment (CD'2004)*, volume 3083 of *LNCS*, pages 50–64, Edinburgh, UK, May 2004. Springer.

[16] H. Liu, M. Parashar, and S. Hariri. A component-based programming model for autonomic applications. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)*, pages 10–17, New York, USA, May 2004. IEEE CS Press.

[17] S.R. White, J.E. Hanson, I. Whalley, D.M. Chess, and J.O. Kephart. An architectural approach to autonomic computing. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)*, pages 2–9, New York, USA, May 2004. IEEE CS Press.