Asynchronous, Hierarchical and Scalable Deployment of Component-Based Applications

Vivien Quéma[1], Roland Balter[2], Luc Bellissard[2],
David Féliot[2], André Freyssinet[2], Serge Lacourte[2]

[1]INPG - Laboratoire LSR-IMAG (CNRS, INPG, UJF) - projet Sardes
[2]ScalAgent Distributed Technologies
INRIA Rhône-Alpes, 655 av. de l'Europe, 38334 Saint-Ismier Cedex, France
Vivien.Quema@inrialpes.fr

Abstract. The deployment of distributed component-based applications is a complex task. Proposed solutions are often centralized, which excludes their use for the deployment of large-scale applications. Besides, these solutions do often not take into account the functional constraints, i.e. the dependences between component activations. Finally, most of them are not fault-tolerant. In this paper, we propose a deployment application that deals with these three problems. It is hierarchical, which is a necessary feature to guarantee scalability. Moreover, it is designed as a distributed workflow decomposed into tasks executing asynchronously, which allows an "as soon as possible" activation of deployed components. Finally, the proposed deployment application is fault-tolerant. This is achieved by the use of persistent agents with atomic execution. This deployment application has been tested and performance measurements show that it is scalable.

1 Introduction

1.1 Context and objectives

As underlined by Emmerich in [1], it was claimed for a long time that object-orientation was the solution to software reusability. Nevertheless, the large number of fine grained classes generated during object-oriented modelling induces a large number of dependencies between them, thus making it difficult to take classes out of the context in which they were developed. To overcome these problems, component models were elaborated. "A component is a unit of composition that can be deployed independently and is subject to composition by a third party." [2]

First component models like COM or JavaBeans had their execution limited to just one machine. These component models have been extended to allow for distributed execution across multiple machines: .NET or EJB. These distributed technologies have induced a new approach of software development that is often referred to as component-based development (CBD). CBD raises numerous challenges, such as distribution management, component discovery, integration of components with legacy software modules, etc. One of the major, not yet solved challenge is the deployment of distributed component-based systems.

According to Carzaniga and al. [3], deployment is a collection of interrelated activities that form the deployment life-cycle. In this paper, we focus on the steps

preliminary to the application's launching, that is: installation, instantiation, binding, and finally activation of the components. In existing component technologies, these steps are often reduced to software delivery: the component code located on one server node is first sent to n customer sites and then activated. Such a deployment process does take into account neither the *applicative logic*, nor the *physical constraints* of the physical environment where the application is to be deployed. A distributed application encompasses a set of components that interact with each other in a complex way, thus making them interdependent. To run consistently components require the availability of other (possibly legacy) components. These dependencies must be carefully taken into account to ensure a consistent activation of each of the components. This issue is referred to as the application logic enforcement. On the other hand the deployment process must be fault tolerant, even in the case of large-scale applications executing on numerous heterogeneous devices with varying operational conditions (i.e. network failures, disconnected mode, etc.). This issue is referred to as the physical constraints enforcement.

1.2 Approach

The paper addresses the two above-mentioned challenges. Application logic enforcement is achieved through the use of an architecture description language (ADL) to describe the distributed application to be deployed. This description relies on a hierarchical component model which is general enough to allow the description of most of component-based systems. Physical constraints enforcement is achieved by designing the deployment process as a scalable and fault tolerant distributed application. Scalability requires that the deployment process is distributed. This is achieved through a hierarchical structure of the deployment process in close connection with the component hierarchy of the application to be deployed. In addition, we propose to implement the deployment application on top of an asynchronous reliable runtime system, to meet the fault tolerance objective.

This paper is structured as follows: section 2 describes the hierarchical component model, and its associated architecture description language is introduced in section 3. Section 4 presents the deployment application. We describe its implementation in section 5 and performance figures are presented in section 6. Related work in addressed in section 7, before concluding the paper in section 8.

2 A hierarchical component model

In this section, we present the component model that is used to model applications to be deployed. For space limitations, we will not justify all the choices that have been made. Our main objective was that the component model be general enough to allow modeling most of the component-based systems. It is inspired by previous work made on distributed component-based applications [4]. It shares similarities with Fractal [5], a recent component model, which also aims at modeling component-based systems. An application is represented by a hierarchical assembly of components. Every component is made of two parts: a *functional part* and a *control part*. Moreover, components are bound using *connectors*.

The functional part The functional part of a component corresponds to the services it provides to other components and the services it requires from other components. These services are accessed via functional interfaces defined by their type, which consists of:

- An *identifier*: a name which is valid in the context of the component that owns the interface,
- A *role* specifying whether the considered interface is a client or a server interface,
- A *signature*: a collection of method signatures. Interface signatures are embodied as usual Java interfaces.
- A *contingency* used to specify whether a client interface is mandatory or optional (i.e. if it must be bound or not at runtime).

The model does not impose any implementation language: components may be developed in Java, C, etc.

The control part A component also owns control interfaces which are server interfaces that embody the control behavior associated with the component. In particular, the control part allows one to manage the component's lifecycle: start, stop, activate and deactivate. Decoupling the control part from the functional part is a necessary feature for administration applications (like deployment applications) to easily manage component-based applications. For instance, we will show later in the paper that the deployment application requires an activation interface.

Composite components The component model is hierarchical: a set of components can be manipulated as a component, called composite. Composite components are useful to represent hierarchical applications. A composite usually encapsulates components that cooperate to provide a functionality. Similarly to primitive components, composite components own functional and control interfaces. Nevertheless, we distinguish internal interfaces that can be bound to encapsulated components and external interfaces that can be bound to components outside the composite.

Connectors A functional interface can be bound to one (or several) other functional interface(s) using a connector. There exist several connectors: local and remote procedure calls, asynchronous message passing, etc. A connector is instantiated by a connector factory. Each factory must provide a `create` method, whose parameters are the identifiers of the interfaces to be bound. Note that binding two functional interfaces requires that the components owning these interfaces are encapsulated within the same composite.

Example Figure 1 represents the architecture of an application that conforms to the component model. For readability purpose, some connectors have been omitted. The application is represented by the `Application` composite. It encapsulates two primitive components `Client 1` and `Client 2`, and a composite component `Topic`. Each client owns two client interfaces — `subscribe` to subscribe to the topic, and `publish` to publish a message on the topic —, and a

server interface `receive` to receive messages broadcasted by the topic. The `Topic` composite component owns three external interfaces and three internal interfaces that are bound to two primitive components responsible for subscribers handling and message broadcasting.

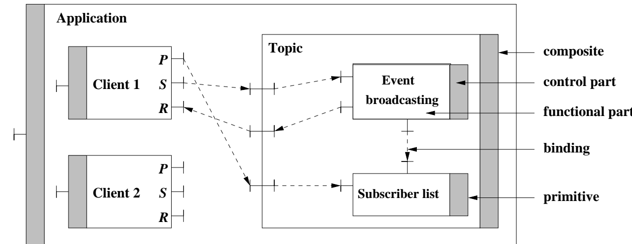

Fig. 1. Example of application following the model.

3 An architecture description language

To manipulate the above described component model, we have defined an architecture description language (ADL). In this paper, we will only show how this ADL is used to deploy applications. Note that it is also used for other purposes [6]. Like other ADLs [7–11], it aims at describing an application as an assembly of interacting components. It is inspired by previous work made on the Olan configuration language [12]. It allows the description of:

- the *functional* part of the application, i.e. the set of (possibly legacy) involved components and their interactions.
- the *non functional* part of the application, i.e. the location of components to be deployed and the order in which they must be activated.

Components and connectors involved in the application are described in a file called *description*. The description of a primitive component specifies its functional interfaces, its control interfaces and its implementation (Java class, binary file, etc.). Similarly, the description of a composite component gives its interfaces (internal and external) and its implementation. Moreover, it describes its internal architecture, that is the set of encapsulated components, their locations, their bindings using connectors and their dependences in term of activation.

As for components, two kinds of connectors may be used: *primitive* and *composite* connectors. Primitive connector descriptions only specify the connector factory to be used. Composite connectors are described as assemblies of interacting components bound by primitive connectors. This makes the description of a composite connector comparable to that of a composite component.

Figure 2 depicts the description of the `Application` composite. It gives its control interfaces and its implementation. For legacy components, the implementation field is replaced by a description of the component location. Furthermore, encapsulated components and connector factories are specified with the keyword `new`, with in parameter the site where they have to be deployed. Bindings between components are specified with the symbol => and with the keyword `using`

```
Composite Application {

  *** No functional interfaces ***
  *** Control interfaces ***
  Activation controller {
    signature = fr.application.Activation ;
  }
  ...

  *** Implementation ***
  Implementation Application {
    type = java ;
    funct_part = void ;
    contr_part = fr.application.ApplicationControl ;
  }

  *** Encapsulated components ***
  Client client1 = new Client (zirconium.inria.fr);
  Client client2 = new Client (argent.inria.fr);
  Topic topic = new Topic (strontium.inria.fr);

  *** Connector factory ***
  Rmi rmiF = new Rmi (zirconium.inria.fr) ;

  *** Component bindings ***
  client1.abonnement => topic.abonnement using rmiF ;
  ...

  *** Activation dependencies ***
  (client1, client2) depend on topic ;

}
```

Fig. 2. Description of the `Application` composite.

that is used to indicate the connector factory to be used. Finally, the order of component activations is specified: it is mentioned that the two client component activations depend on the topic activation. Note that when no activation order is given, a component is considered to be activable as soon as its client functional interfaces are bound, even though the components owning the server interfaces are not activated.

4 The deployment application

The deployment application uses the ADL description to instantiate and bind application components. Our goal is twofold: (1) to use the ADL description to ensure the respect of functional constraints and (2) to exploit the hierarchical structure of the application to distribute the deployment intelligence. To reach our goal, the deployment application is implemented as a *distributed workflow* decomposed in *tasks*. Tasks execute in parallel or sequentially. They are responsible for the various deployment operations: instantiation, binding, activation. These tasks execute within hierarchically organized entities called *deployment controllers*. We first show how the controller hierarchy is built. We then describe the architecture of controllers.

4.1 Deployment controller hierarchy

Recall that the component model requires that two bound components be encapsulated in the same composite. As a consequence, an application built using

this model has always a tree-like architecture: nodes of the tree are composite components, whereas leafs are primitive components. The deployment controllers' hierarchy follows this treelike hierarchy: each composite is associated with a controller. Figure 3 illustrates this concept: a deployment controller is associated to each of the four composites C_1, C_2, C_3 and C_4.

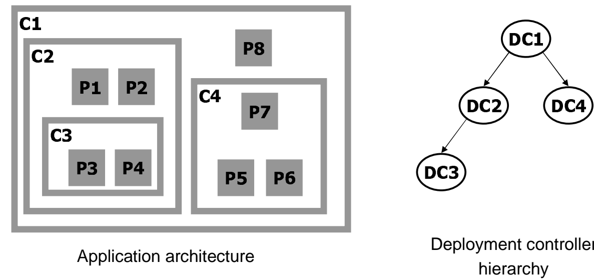

Fig. 3. Applicative hierarchy.

However, it is also possible to extend this hierarchy by creating controllers in charge of a subset of components. Such an example is depicted on figure 4: components P_1 , P_2 and P_5, P_6 are associated to controllers $DCBis_1$ and $DCBis_2$, respectively. This consists in creating "virtual" composites, without functional and control code. This extension possibility is interesting for applications built using flat component models such as the Corba Component Model (CCM [13]), or Sun'S EJB model [14].

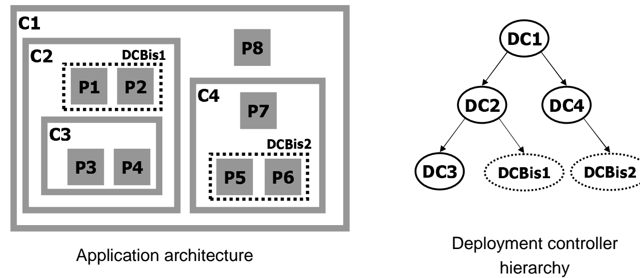

Fig. 4. Extended hierarchy.

4.2 Deployment controller architecture

Principle Except for the root deployment controller, each controller is created by its parent controller. To communicate, two controllers must establish a session. This session is used to exchange control messages.

As depicted on figure 5, each controller hosts a set of tasks responsible for various aspects of the deployment. These tasks can either be created by other tasks hosted by the controller or by tasks executing within the parent controller

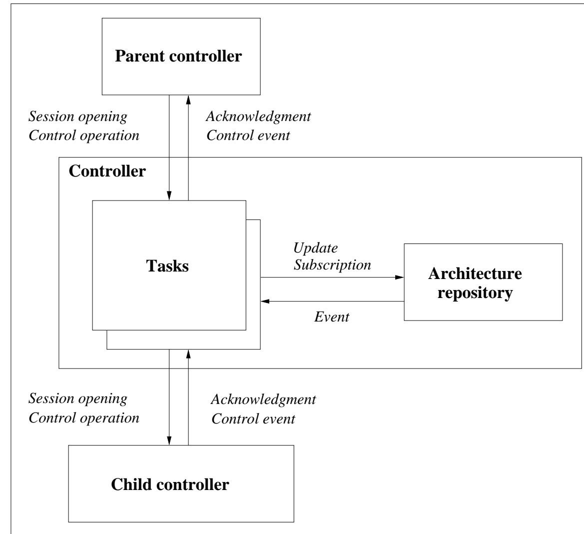

Fig. 5. Architecture of a controller.

(creation orders are propagated using the session established between the two controllers). Tasks store their results in an architecture repository. Each time the repository is updated, it sends an event reporting the update to interested tasks, which react by either executing local operations, or creating other tasks.

Tasks We distinguish four kinds of tasks:

- *Creation tasks* are in charge of creating components. Their code depends on the type of component to be instantiated (Java, C, etc.). Moreover, in the case of composite components, the creation task creates the controller associated with the composite and opens a session that takes as parameter the ADL description of the composite to be deployed. This session is used by other tasks to send control orders. At the end of this task, the architecture repository is updated with references to the created component and its functional interfaces. Note that in the case of a "virtual" composite, stored references are those of its encapsulated components.
- *Integration tasks* are in charge of integrating legacy components. These are retrieved using information stored in the ADL description. The architecture repository is then updated with information about the component references.
- *Binding tasks* are in charge of binding components. Their code depends on the type of connector to be instantiated. Nevertheless they follow the same pattern: they first create the connector factory using information stored in the ADL description; then, they create the connector by calling the `create` method with appropriate parameters. These parameters are retrieved from the architecture repository.
- *Activation tasks* are in charge of activating components. For primitive components, this only consists in calling the `activate` method provided by the `Activation` control interface. For composite components, this task uses the session established with the child controller to send an activation order. Once the order is completed, the session with the child controller is closed.

Organization of tasks Tasks are created by a controller according to the ADL description of the composite to be deployed. One creation (or integration) task and one activation task are created for each encapsulated component. Moreover, one binding task is created for each connector to be built. Note that in the case of a "virtual" composite, only connectors between encapsulated components are managed by the controller. Indeed, the bindings between encapsulated and other components are done by the parent controller.

Tasks within a controller execute independently. They synchronize through events generated by the repository upon updates. Tasks execute either in parallel or sequentially according to their type and the components they work on: all the creation and integration tasks execute in parallel. On the other hand, a binding task depends on the creation and integration tasks in charge of the components to be bound. Finally, an activation task executes after both all the component's required services have been bound and all the components it depends on have been activated. This organization of tasks guarantees an "as soon as possible" activation of each component. This would not be the case if all the creation tasks were delayed until all the binding tasks complete, themselves being delayed until the completion of all the creation tasks.

4.3 Fault tolerance

Like every large-scale distributed application, the deployment application can be subject to node crashes, network breakdowns, disconnected sites, etc. It is thus necessary to discover and handle these faults.

Fault detection Two kinds of faults may happen: either a task is blocked — for example, a component creation does not complete —, or a network or a machine crashes. In the latter case, the tasks interacting with this machine will block. Thus, each fault causes one or several tasks to block. Two fault discovering strategies are possible: the first one consists in setting bounds to the execution times of the different tasks. Once a bound has been raised, an error message is propagated. This method is not viable for the deployment of large-scale applications, since it is very difficult to determine realistic bounds.

The strategy we have adopted is "optimistic": no error message is propagated. Instead, every controller owns a supplementary task, called monitoring task whose role is to observe other tasks' progression. This task collects events produced by the repository, filters them and forwards interesting events to the monitoring task executing within the parent controller. All these events are received by the monitoring task executing within the root controller. This task is used by the application administrator to check the deployment progression and to detect faults.

Fault handling Faults are handled following a two steps process: all or part of the controllers are stopped. Then, a new deployment order is given. Between these two steps, a site can be restarted, the ADL description can be modified, etc. Stopping a deployment controller is made possible by sending a stop order using the session established with its parent controller. This causes the different tasks executing within the controller to stop. Redeploying the application is made possible by opening new sessions along with the (possibly modified) ADL

description. Stopped tasks are restarted (sometimes recreated). The repository is also restarted and used to determine the operations that remain to be done. It is important to note that for this mechanism to work correctly, it is required (1) that repositories and tasks have persistent states, and (2) that communications between them be reliable.

5 Implementation

The deployment application has been implemented using the ScalAgent middleware [15]. It is a fault-tolerant platform that combines asynchronous communications with a programming model using distributed, persistent software entities called *agents*.

The agent paradigm Agents are autonomous reactive objects executing concurrently, and communicating through an event-reaction pattern, thus following the *actor* paradigm [16]. An event is a typed data structure used for exchanging information with other agents. Once an agent receives an event, it reacts accordingly, thus changing its state and/or communicating with other agents. Agents are persistent, which means that the agent lifetime is not bounded to the duration of the execution. However, persistence is not sufficient for retrieving a consistent state after failure. Also, agent reactions are atomic. This property ensures that a reaction is either fully executed or not executed at all.

The execution infrastructure This event-reaction model is based on a MOM which guarantees the reliable, causal delivery of messages. The MOM is represented by a set of agent servers organized in a bus architecture. Each server is made up of two components, the *local bus* and the *engine* (see figure 6). The local bus conveys messages. It is made of a *channel* in charge of routing messages and several networks that implement the basic message-based communication layers. The engine is responsible for the creation and execution of agents. It behaves according to their event-reaction model. It performs a set of instructions in a loop, getting the next message from the channel and making the proper agent react.

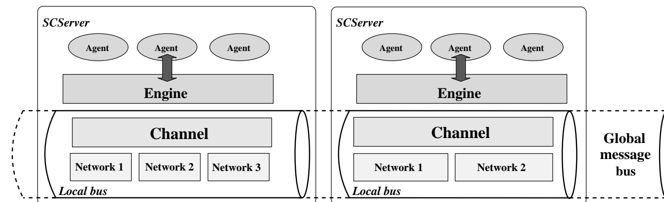

Fig. 6. Two interconnected agent servers.

Implementing the deployment application Controllers, tasks, as well as repositories are implemented using agents. At session creation time, the agent

implementing the child controller is (possibly remotely) created. This controller agent creates the agent responsible for the repository and uses the ADL description to create the agents responsible for the various tasks: a creation and an activation task by encapsulated component, as well as a binding task for each binding to be established. Fault tolerance is made possible by the atomic execution of agents, which guarantees that restarted tasks and architecture repositories are in a consistent state.

6 Evaluation

Performance measurements have been done to validate the proposed deployment application. They have been performed on a 216 PCs cluster equipped with 733MHz Intel Pentium III processor and 256Mo RAM. Tests consisted in deploying applications, whose architecture follows the pattern represented on figure 7. Application components are implemented using Java objects communicating using the ScalAgent middleware.

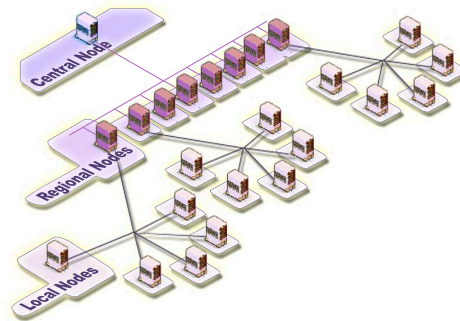

Fig. 7. Architecture of the test application.

Recall that deployed applications have always a tree-like hierarchy. The test application hierarchy is made of three levels: a central site connected to n composite components, each one encapsulating a regional composite connected to m local composites. Every local composite encapsulates 10 primitive components. Every regional composite encapsulates 10 primitive components as well as local composites. The measured metric is the average deployment completion time. We varied several parameters:

- The number of physical machines that host application components. This number ranges from 11 to 151.
- The number of local composites. This number ranges from 1 to 7000. In the case of the test application, each local composite execute within a Java process. As a matter of fact, the more the number of local composites increases, the more the required power on the machine is significant.
- The number of regional composites. This number ranges from 1 to 50. Note that increasing the number of regional composites decreases the number of local composites that each one encapsulates.

6.1 Distribution impact evaluation

To evaluate the distribution impact, the number of regional and local composites remains constant ($m = 10$ and $n = 100$), whereas the number of machines that host application components varies (from 20 to 111). Such applications involve 110 Java processes, each one hosting approximately 10 components. Obtained results are presented on figure 8. The average completion time ranges from 45 to 55. We can see that up to a certain number of machines, the decentralization increases the deployment performances. This is mainly due to the fact that each machine hosts less Java processes. Nevertheless, a too large number of machines increases the use of remote communications, thus reducing the deployment performances.

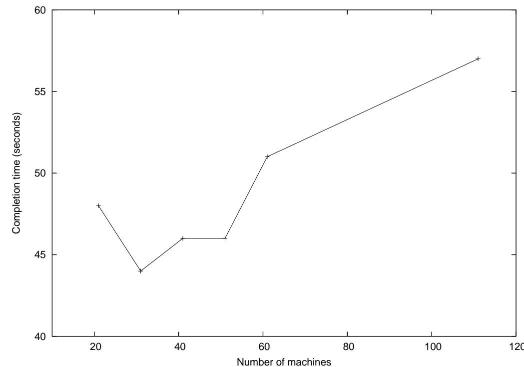

Fig. 8. Distribution impact evaluation.

6.2 Architectural impact evaluation

This test aims at evaluating the impact of the composite hierarchy. Recall that the application is represented by a central composite encapsulating m regional composites, each one encapsulating — besides its own primitive components —, $n = m$ local composites. Figure 9 shows that, for both a fixed number of machines (= 60) and a fixed number of local composites ($n = 500$), the deployment completion time is minimum for $m = 10$ regional composites (i.e. $n = m = 50$). This arises from the deployment application architecture which associates a controller to each composite. An increase of the number of controllers induces a parallelization of the deployment. Nevertheless, this number must be kept reasonable, since too large of a number causes each controller to handle a large number of sessions, which slows down the deployment process. This experience shows that the hierarchical structure of the application architecture has an impact on the deployment performances, which opens research perspectives towards tools to help determining an application's optimal architecture with regard to its deployment.

6.3 Scalability evaluation

This test is the most important. It aims at verifying that the proportional increase of parameters (n, m and the number of machines) does not cause an

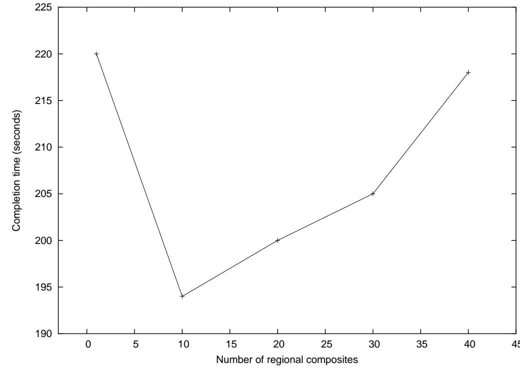

Fig. 9. Architectural impact evaluation.

exponential increase of the deployment completion time. Figure 10 shows that this time remains linear, with n ranging from 10 to 450, m from 1 to 45, and the number of machines from 3 to 91.

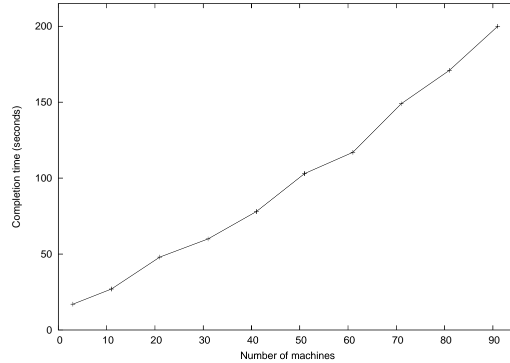

Fig. 10. Scalability assesment.

7 Related work

Deployment has been an increasing area of interest this last 10 years. Work has been done on the SOFA component model [17]. Similarly to our approach, they use an ADL (CDL for Component Definition Language) which allows the description of the application to be deployed as a hierarchical assembly of components. Nevertheless, the proposed deployment process is centralized and does not take advantage from the hierarchical structure of the application. Indeed, a centralized server uses the ADL to propagate creation and binding orders.

[18] focuses on a deployment specific problem: the one of component discovery. The authors propose a trading service based both on the type of the component interfaces and on semantic information about the component. This tool has been integrated into the Corba component model (CCM) deployment

process. The proposed protocol is synchronous and centralized, which we think is not a viable approach for large-scale applications. Such a tool could be integrated into our proposition as complementary to the ADL. It would be used by the different tasks to retrieve information they need, thus allowing dynamic evolution of the ADL description.

Researchers at University of Colorado have proposed a reconfiguration system for the Sun's EJB component model [19]. BARK (*Bean Automatic Reconfiguration FrameworK*) is designed to facilitate all the tasks in the deployment lifecycle for EJBs. It provides functions to download component packages over the Internet and load them into the EJB container. It also manipulates the component package descriptors to provide control over component bindings. In that sense, BARK may be compared to an ADL. BARK is composed of a set of Application Server Modules (ASM) that work in cooperation with EJB application servers. A central Repository is used for storing component software packages. Finally, a tool determines the scripts to be executed by each ASMs. The proposed deployment process shares similarity with ours since it uses an ADL-like description of EJBs. However, deployment orders are centralized and synchronous.

Hall et al. have worked on software deployment [20]. The architecture they propose, called *Software Dock*, is composed of a set of servers, or *docks*. These *docks* help registering software releases and software consumer configurations. They are used by *agents* that are in charge of a specific deployment task. Agents can migrate from *dock* to *dock* and communicate using a wide-area event system. Similarly to our approach, this system uses asynchronous communications and agents. However, this system is not dedicated to component deployment; it is agnostic to development approaches. As a matter of fact, the deployment is not hierarchical and it does not allow an "as soon as possible" activation of part of the components. Finally, agents do not execute atomically, which precludes fault tolerance.

We conclude this related work survey with work conducted by the Grid community. In particular, *proactive* [21] is an active object-based distributed programming model. One of the designers' goal is to remove any reference to physical machines, repositories, etc. from the code of objects. This is achieved by using "virtual structures". They are mapped onto physical structures using XML descriptors. Proactive shares our goal of separating functional code from deployment-related data. Nevertheless, we do not aim at the same type of deployment: this system targets lowlevel software deployment like Java virtual machines. We are focused on component-based applications.

8 Conclusion and future work

The deployment of distributed component-based applications is one of the challenges raised by the *component-based development*. Existing solutions are often centralized and only consist in software delivery. Such deployment processes have several drawbacks: their centralized control excludes the deployment of large-scale distributed applications. Moreover, these processes do not take into account the *applicative logic*: the activation is often unnecessarily delayed and they do not propose to integrate legacy components with deployed applications. Finally, they do not take into account the *physical constraints* of the system on

which the application is to be deployed: as a consequence, most of them do not tolerate faults.

In this paper, we have proposed a deployment application that solves these three challenges. (1) There is no centralized control: the deployment application is hierarchically organized according to the application's architecture. This hierarchical organization avoids bottlenecks and makes the deployment scalable. (2) The asynchronous and parallel execution of tasks allows an "as soon as possible" activation of deployed components. Indeed, every task synchronizes (using the architecture repository) only with tasks it depends on. Dependency knowledge is made available by the ADL description of the application. (3) Finally, the proposed deployment application is fault tolerant. This is achieved by the use of persistent agents with atomic execution.

Future work: first performance measurements are an excellent encouragement to pursue our work. First, we plan to refine the experiments we have done in order to evaluate the computational and memory overheads of the deployment controllers, and the number of concurrent tasks a controller can support. Second, we plan to provide the deployment application with support for high availability. As a matter of fact, while allowing a great deal of parallelism, the hierarchy of deployment controllers is still amenable to bottlenecks and single points of failure. We are currently adding replication capabilities to the ScalAgent MOM; this will allow duplicating the deployment controllers, thus improving the availability of the deployment application.

References

1. W. Emmerich. Distributed Component Technologies and their Software Engineering Implications. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 537 – 546, Orlando, Florida, May 2002.
2. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
3. A. Carzaniga, A. Fuggetta, R. Hall, A. van der Hoek, D. Heimbigner, and A. Wolf. A Characterization Framework for Software Deployment Technologies. Technical Report 857-98, Department of Computer Science, University of Colorado, 1998.
4. L. Bellissard, S. Ben Atallah, F. Boyer, and M. Riveill. Distributed Application Configuration. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'96)*, pages 579–585, Hong-Kong, May 1996.
5. E. Bruneton, T. Coupaye, and J.-B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga, Spain, June 10th-14th 2002.
6. Vivien Quéma and Emmanuel Cecchet. The Role of Software Architecture in Configuring Middleware: the ScalAgent Experience. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS'2003)*, La Martinique, France, 2003.
7. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Software Engineering*, 21(4):314–335, 1995.
8. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 44–53, 1999.

9. V. Issarny, C. Bidan, and T. Saridakis. Achieving Middleware Customization in a Configuration-Based Development Environment : Experience with the Aster Prototype. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 207–214, Annapolis, Maryland, USA, May 1998.
10. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. In *IEEE Transactions on Software Engineering, Special Issue on Software Architecture, Vol. 21, No. 4*, pages 336–355, April 1995.
11. R. Allen, D. Garlan, and R. Douence. Specifying Dynamism in Software Architectures. In *Proceedings of the Workshop on Foundations of Component-Based Software Engineering*, Zurich, Switzerland, September 1997.
12. R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.Y. Vion-Dury. Architecturing and Configuring Distributed Applications with Olan. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, UK, September 1998.
13. Philippe Merle, editor. *CORBA 3.0 New Components Chapters*. OMG TC Document ptc/2001-11-03, November 2001.
14. Enterprise JavaBeansTM Specification, Version 2.1, August 2002. Sun Microsystems, `http://java.sun.com/products/ejb/`.
15. L. Bellissard, N. de Palma, A. Freyssinet, M. Herrmann, and S. Lacourte. An Agent Plateform for Reliable Asynchronous Distributed Programming. In *Symposium on Reliable Distributed Systems (SRDS'99)*, Lausanne, Switzerland, October 1999.
16. G. A. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. In *The MIT Press, ISBN 0-262-01092-5*, Cambridge, MA, 1986.
17. T. Kalibera and P. Tuma. Distributed Component System Based On Architecture Description: the SOFA Experience. In *Proceedings of the 4th International Symposium on Distributed Objects and Applications (DOA'02)*, Irvine, CA, October 2002.
18. D. Kebbal and G. Bernard. Component Search Service and Deployment of Distributed Applications. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01)*, Roma, Italy, September 2001.
19. M. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and A. Wolf. Reconfiguration in the Enterprise JavaBean Component Model. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD'02)*, pages 67–81, Berlin, Germany, June 2002.
20. R. Hall, D. Heimbigner, and A. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 174–183, Los Angeles, CA, May 1999.
21. F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssière. Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications. In *Proceedings of the 11th International Symposium on High Performance Distributed Computing (HPDC'02)*, pages 93–102, Edinburgh, Scottland, July 2002.