A Platform Architecture to Support the Deployment of Distributed Applications

Tonghong Li, Andreas Hoffmann, Marc Born and Ina Schieferdecker
GMD FOKUS
Kaiserin Augusta Allee 31
D-10589 Berlin, Germany

Abstract-**Deploying a distributed application on the nodes of a network has undoubtedly been a daunting task, which encompasses the distribution as well as the configuration of its components. Deployment and configuration concerns the placement of software components onto the nodes of a target environment, their installation, configuration, and instantiation, as well as the distribution and configuration tasks during the service usage phase. However, they are not fully addressed by current CORBA Component Model (CCM). This paper elaborates on the platform architecture for deployment support and proposes the necessary extensions to current CCM.**

I. Introduction

Deploying a distributed application on the nodes of a network has undoubtedly been a daunting task, which is usually done by writing and manually executing installation and configuration scripts on hundreds or even thousands of nodes. Experience shows today that software support is needed to manage software distribution and configuration in heterogeneous environments featuring different middleware products, development tools, and methods. Automating the deployment process can reduce the software development effort and the dependencies on software vendor's middleware products, and shorten the time to market.

Proprietary solutions for this problem exist already in the tools market, such as IBM's Tivoli [1], Microsoft's Transaction Server [2], and Microsoft's Management Console [3]. Though these tools are relatively opened for Independent Software Vendors' (ISV) plug-ins, they are targeted to specific environments and therefore present limitations in heterogeneous environments. Another kind of proprietary solution for the problem of deploying distributed applications is represented by the tools coined as Application Servers, which are usually sold by the Java programming language Interactive Developing Environment's vendors. This kind of tools were born as supporters for the deployment of Enterprise Java Beans (EJB), thus being restricted to applications or components developed using Java and the EJB specification [4].

The CCM recently proposed by the Object Management Group (OMG) [5] is a comprehensive framework for deployment, assembling and packaging of CORBA-based distributed applications. The CORBA component specification comes with a deployment model that enables the partial automation of basic deployment tasks, such as installation, assemblage of components, etc. It includes a Deployment XML, which provides a couple of means for expressing configuration and deployment related issues. Furthermore, it defines a number of platform interfaces to support the deployment process in terms of Deployment XML. However, for comprehensive deployment and configuration support (including run-time reconfiguration), these are not sufficient.

The EURESCOM project P924 [6] targets concepts, methods, and notations for the deployment of distributed applications onto target middleware platforms. It has come up with a deployment and configuration language (DCL), which permits the description of configuration and distribution information for distributed applications. However, to be able to accomplish the deployment process, target platform support is necessary. From a more general view this comprises platform facilities supporting the initial installation, the configuration and the general management of components in an efficient, at best automatic way.

This paper elaborates on the platform architecture for deployment support. It is organized as follows. Section II gives an overview of our DCL language. Section III introduces the deployment procedure. Section IV elaborates our platform architecture. Section V gives some scenarios to explain how to deploy distributed applications under our platform architecture. Section VI summarizes our sample service. Section VII concludes the paper and presents the issue for further work.

II. DCL Language Overview

In DCL the following representation formats are proposed [7]:

- a graphical format (DCL/gr),
- a textual format (DCL/pr), and
- an interchange format (DCL/cif).

The semantics and the metamodel of DCL provide the common foundations for all three. While the graphical and the textual representation formats are intended to be used by humans for the creation and visualization of deployment specifications, the interchange format of DCL is mainly to be interpreted by tools and middleware platform during the entire deployment.

The graphical format (DCL/gr) is actually a specialisation of UML [8]. Therefore a special UML profile is defined for DCL, by specialising the UML metamodel.

The textual format allows for a user-friendly concrete syntax with no sophisticated editing tool requirements. This format might not be as intuitive as the graphical one, but it can be seen as a lighter means for the creation of DCL models.

The interchange format for DCL is actually an extended version of CCM's XML descriptors. It is intended to be the common interchange format for all kinds of deployment and configuration specifications between all phases of the entire deployment process independently from tool vendors and customer's middleware target environments. The interchange format of DCL carries all the information needed for

0-7803-7400-2/02/$17.00 © 2002 IEEE

deployment and configuration of distributed applications.

The DCL/gr and DCL/pr can be transformed to DCL/cif by appropriate translators. The deployment tool and middleware platform always use DCL/cif for the deployment of distributed applications.

DCL/cif extends existing CCM XML descriptors, in order to enable:

- the precise definition of the installation map,
- the embedding of notations for constraints and actions,
- a more general approach for defining properties as well as requirements on the target environment using the embedded constraint language,
- means for the definition of dynamic properties to be monitored at run-time,
- the definition of rules for run-time re-configuration.

DCL/cif defines two additional descriptors: environment DTD and node properties DTD. The environment DTD describes the overall structure of a target environment in terms of nodes and links between nodes as well as overall capabilities of the platform, while the node properties DTD focuses on the description of the properties and capabilities of a certain node of the target environment.

III. Deployment Overview

Because a distributed application that is ready for deployment is in the form of component assembly archive [5], we use the term "assembly type" to address a type of distributed application. An assembly package consists of a component assembly descriptor (CAD) and a set of component packages and property files. The CAD contains information about which components make up the assembly, how those components are partitioned, how they are connected to each other. A component package is a specialization of a general software package, which maintains one or more implementations of a component.

During the deployment procedure, the deployment tool has to open the assembly package and read and parse all the XML descriptors provided by the vendor of the shipped service. Furthermore, it needs to interact with the middleware platform to get its required information like target environment and use platform specific services to accomplish the deployment process. In summary, it has to perform the following steps:

1. Identification on which hosts the components of the distributed application are to be installed (installation map) as well as definition of reconfiguration rules. This information will be added to the CAD of assembly package.

2. Installation of component implementations on each node of the platform according to the installation map.

3. Instantiation of components on particular nodes.

4. Connection of component instances and performance of an initial configuration as specified in the CAD.

5. Supervision of the complete application execution according to rules contained in the CAD and potential runtime re-configuration of single component or the entire application topology.

From the above description, it is obvious that middleware platform should provide the capabilities of installation and instantiation of components, management of properties of platform and components, and reconfiguration of deployed application according to the included rules if the whole deployment procedure can be automated.

IV. Platform Architecture

In order to support deployment and configuration of distributed applications specific platform interfaces are needed. A first step towards this deployment support is proposed by CCM, which defines a number of platform and application interfaces. However, it only supports the partial automation of basic deployment tasks, such as installation, assemblage of components, etc. Thus, we define a more sophisticated architecture by extending the interfaces proposed by CCM. In the following interface description, we will point out if the interface comes from CCM.

A. Description of the Interfaces

- **NodeManagement**

This interface is used to obtain a node's properties in terms of node properties DTD. It also provides a set of operations to allow users to get and set a property's value. A property is two tuples of <property_name, property_value>. The property_name is a string that names the property, while the property_value is of type any and carries the value assigned to the property.

Moreover, it provides a operation to execute all activities related to loading a component, for example, creating a component server, creating a container within the server and installing a home object within the container. In CCM specification, these works are completed with the help of a set of objects on each node, which is rather complicated [5]. In our implementation, each component provides an executable program to finish this work. What this operation has to do is to create a new thread or process and launch the execution of the program in this thread or process.

- **ExtComponentInstallation**

In order to support an installation capability CCM introduces the specification of the interface **ComponentInstallation**, which is used to install, query, and remove component implementations on a single node. However, it is not specified how the component package is uploaded. The **ExtComponentInstallation** interface extends the **ComponentInstallation** interface by adding an upload operation.

- **ConstraintMonitor**

This interface represents a run-time configuration rule. A rule is composed of two parts: a run-time constraint as the precondition for an action, an action to be automatically performed if the precondition of the action is violated. This interface performs the tasks of evaluating the constraint and executing the corresponding action when the constraint is violated.

Since there are a couple of specification and scripting notations, which seem to be suitable for specifying constraints and actions, it has been decided to embed such an existing notation rather than to develop a new one in DCL. The advantage of this open approach is that the implementation of a proper tool support is easy to provide, because the interpreters for these notations can be freely obtained. During the computation of DCL/cif specification, the appropriate external interpreter will be invoked for the evaluation of constraints and the execution of actions.

Currently, the Object Constraint Language (OCL) [8], Perl [9] and Python [10] are regarded to be suitable for embedding into DCL/cif. In our opinion, the most preferred language is Python [10]. This is due to the following reasons: (1) Python is a powerful scripting language for nearly all object-oriented concerns, it provides proper means for describing properties and constraints as well as user-defined actions. (2) OMG has defined and already standardised a mapping from Python to CORBA. Hence, it is possible to refer to CORBA-interfaces within user-defined actions.

The following rule is excerpted from our sample service discussed in section VI. This rule will be evaluated every 10 seconds. In case that a node where one of the philosophers runs has a load higher than 90 percent, then an action is performed which writes a message to a control window.

```
  <rule name="PhilosopherRule">
       <condition language="python" interval="10">
              ::$node.get_property_value("CPULoad"
              ).value() > 90
        </condition>
        <action language="python">
              ::DemoReport.notice("$instance:
              critical load on $node")
       </action>
</rule>
```

There are two approaches to evaluate constraints. One adopts the push model: first setting the trigger events, evaluating the constraint only when the events happen. This needs the support from the platform, which should provide a mechnism to notify the client in case a property value has changed. The other uses the pull model : evaluating the constraints in the specified time, requesting the needed property values from the platform during the process of constraint evaluation. For simplicity, we only use pull model for contraint evaluation in our prototype.

Before the evaluator of the selected scripting language is invoked, the constraints and actions have to be translated to an expression that is interpretable by that evaluator. The transformation is done in two steps: (1) replacement of the keywords: **$instance** and **$node** by the instance name and the name of the node the instance is assigned to, (2) translation of scopes to CORBA interface via the name service.

- **ConstraintMonitorFactory**

The interface is a factory for **ConstraintMonitor** interfaces. It is used to create **ConstraintMonitor** instances.

- **DPEManagement**

This interface is used to manage **AssemblyTypeManagement** interfaces, including creation, registration, finding and listing.

- **AssemblyTypeManagement**

This interface is the management interface for a certain assembly type. It comprises the installation and deinstallation of implementation codes according to the assembly description of that assembly type, and the management of real assembly instances.

- **Assembly**

The interface is taken from the CCM specification. It represents an assembly instance. It is used to build up and tear down a distributed application. Building the assembly means that it is going to instantiate all of the components in the assembly and create connections between them. Tearing the assembly down means removing all connections and destroying the components in the assembly.

- **AssemblyFactory**

The interface is taken from CCM, which is used to create Assembly instances.

B. Description of the Components

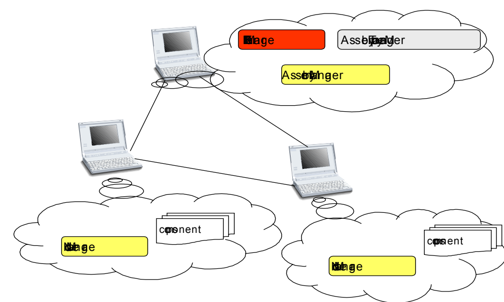

Fig. 1. One scenario of platform architecture.

In our platform architecture there are four components: **DPEManager**, **NodeManager**, **AssemblyTypeManager**, and **AssemblyManager**. **NodeManager** has to be installed in all nodes of middleware platform. Depending on the customer’s requirements and network status, **DPEManager** can be placed either in a specific node that is responsible for the management of the entire middleware platform or in any node of the system. **AssemblyTypeManager** and **AssemblyManager** are created dynamically. During the deployment procedure, only one **AssemblyTypeManager** is created for one assembly type, which is responsible for installing and uninstalling the software package of this assembly type. In order to make this type of assembly running, the **AssemblyTypeManager** has to instantiate a **AssemblyManager**, which is responsible for the management of the running assembly instance. An **AssemblyTypeManager** can instantiate one or more **AssemblyManagers**. Figure 1 shows one scenario of our platform architecture.

- **DPEManager**

The **DPEManager** is a component responsible for the management of the platform. There is exactly one entity of **DPEManager** running on a platform each time. A DPEManager supports the interfaces **DPEManagement**, **ConstraintMonitorFactory**, and **AssemblyFactory**.

The **DPEManager** is the central (initial) access point for all applications and external tools to the platform. It provides functionalities of automated environment detection and code upload. In addition, it is responsible for instantiation of the **AssemblyTypeManagers** and transmission of component assembly package to the **AssemblyTypeManager**.

- **NodeManager**

On each node of the platform a **NodeManager** is running. Its supported interfaces are **ExtComponentInstallation** and **NodeManagement**. It provides access to node specific information and services. In addition, it is responsible for code uploading, controlling and accessing properties of that node.

- **AssemblyTypeManager**

The **AssemblyTypeManager** is responsible for the installation and de-installation of an assembly type. It supports the interface **AssemblyTypeManagement** and uses the interface **ExtComponentInstallation**.

Since it controls code upload to nodes, it needs to have XML parsing facilities to read the installation map contained in the XML descriptors. Furthermore, it stores all rules for run-time re-configuration.

- **AssemblyManager**

It supports the interface **Assembly** and requires a set of interfaces for component instantiation and constraint evaluation.

The **AssemblyManager** manages a single running instance of an assembly type. It is responsible for the instantiation and initial configuration set-up for all components of that assembly. Furthermore, it starts the permanent evaluation of run-time re-configuration rules.

The management of a running assembly instance encompasses: (1) registration of dynamically created components, (2) migration control of single components, (3) tearing down, stopping and resuming of running Assemblies.

V. Common Scenarios

To facilitate better understanding of the interaction between those interfaces specified in the section IV, the subsequent sections contain diagrams illustrating important processes concerning deployment and configuration.

A. *Component Installation*

Figure 2 is the diagram of component installation, which is explains as follows:

1. The **AssemblyTypeManager** calls the upload operation on the **ExtComponentInstallation** interface provided by the **NodeManager**. This operation results in uploading the software package containing the component implementations to the node where the **NodeManager** instance belongs. After this call the returned identifier can be used to identify this package for the purpose of installation of component implementations.

2. The **AssemblyTypeManager** calls the install operation on the **ExtComponentInstallation** interface. Within this operation the previously uploaded package is referred and a certain implementation specified by its id is installed on the node.

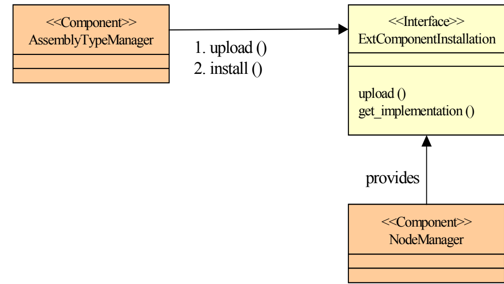

Fig. 2. Component installation.

B. *Component Instantiation*

Figure 3 is the diagram of component instantiation, which is explained as follows:

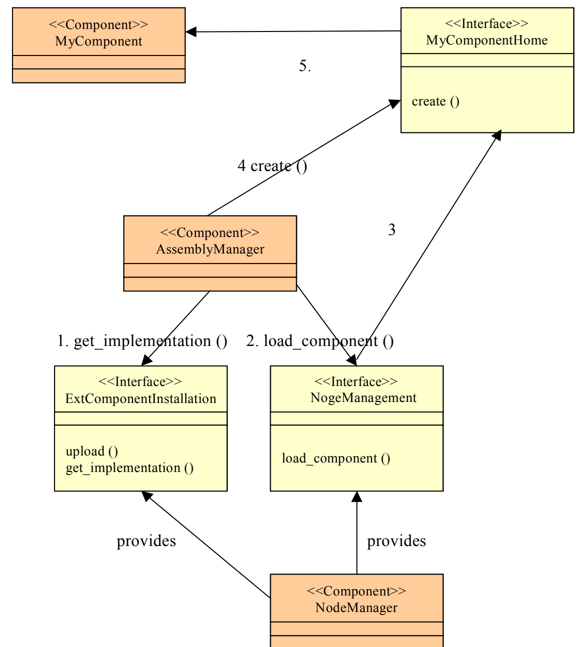

Fig. 3. Component instantiation

1. The **AssemblyManager** calls get_implementation on the **ExtComponentInstallation** interface of the target node. It gives the id of the installed implementation according to the assembly description and obtains the location of installed implementation.

2. The **NodeManagement** interface of the target node is called.

3. The component implementation is loaded into the container and a **Home** instance for this component is created.

4. The **AssemblyManager** calls the create operation on the **Home** interface.

5. A component instance is created and its reference returned to the **AssemblyManager**.

C. Deployment Scenario

Figure 4 is diagram of deployment scenario, which is explained as follows:

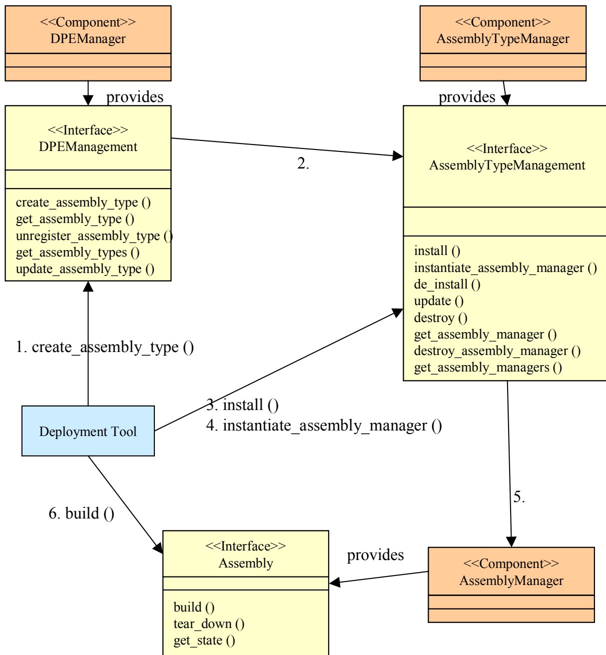

Fig. 4. Deployment scenario.

1.The deployment tool calls create_assembly_type on the **DPEManagement** interface of the actual platform (**DPEManager**). It provides the XML description of the assembly type and obtains a reference of the **AssemblyTypeManagement** interface of that type.

2.An **AssemblyTypeManager** instance is created providing the implementation of **AssemblyTypeManagement**. In the process of create_assembly_type operation, reconfiguration rules for this assembly type are parsed from the XML file and are stored in a local repository.

3. The deployment tool calls install on the **AssemblyTypeManagement** to install the proper implementation codes on the platform. The installation details are described in the component installation scenario.

4. The deployment tool calls instantiate_assembly_manager on the **AssemblyTypeManagement** object.

5. An **AssemblyManager** instance is created having access to the XML description of the assembly type, which serves as a recipe for the actual assembly instance creation.

6. An assembly instance is created by calling the build operation on the **AssemblyManager** object. The component instantiation is carried out in the build operation. In the process of build operation, the **AssemblyManager** also looks up the repository in the **AssemblyTypeManager** in order to obtain all re-configuration rules for this assembly type. For each rule, it calls **ConstraintMonitorFactory** once to generate the corresponding **ConstraintMonitor** object. After that, the assembly instance is running. The platform continually monitors its execution, determining whether or not it should be reconfigured based on the evaluation of its constraints.

VI. Sample Service

This sample service that has been selected to demonstrate our prototype implementation (deployment tool and platform support) is the "classical" dining philosophers' problem [11]. The example scenario includes three different components Philosopher, Fork and Observer, which can be distributed across the target network. For each component, we provide two implementation versions: Window NT and Linux, which are packed in a component package.

A configurable number of philosophers are sitting on a round table; on the table are a finite number of forks. Philosophers perform actions: thinking, eating and sleeping. They do not need any resources in order to think or sleep, but they need two forks for eating, one for the left hand and one for the right hand. Therefore, before starting to eat, a philosopher tries to get the two forks, which are configured to be next to him. An observer will be notified by all philosophers in the case of an activity change, i.e., at the time a philosopher starts eating, starts thinking or starts sleeping. Furthermore, the critical state of getting hungry is notified to an observer as well.

VII. Conclusion

With the increase of the complexity of today's distributed applications, it is of paramount importance to enable the automation of the entire deployment process of distributed applications. In this paper we extend platform interfaces proposed by CCM to achieve this target. The work here has been influenced, and, consequently, evolved around the CORBA Component Model.

References

[1] IBM, "A Project Guide for Deploying Tivoli Solutions", http://www.redbooks.ibm.com.
[2] Jennings, R., "Microsoft Transaction Server 2.0", SAMS Publishing, 1997.
[3] Microsoft Corp., "Microsoft Management Console Overview", 1999.
[4] Sun Microsystems Inc., "Enterprise JavaBeans™ Specification", v1.1, Preliminary version, 1999, http://www.javasoft.com.
[5] OMG, "CORBA components Volume 1", June 1999, orbos/99-07-01.
[6] EURESCOM P924, http://www.eurescom.de.
[7] EURESCOM project P924, Deliverable D2, "Notation and Semantics for Deployment and Configuration", June, 2001.
[8] OCL: The Object Constraint Language, http://www-4.ibm.com/software/ad/standards/ocl.html.
[9] Perl Mongers, http://www.perl.org/.
[10] PYTHON, http://www.python.org/.
[11] EURESCOM project P924, Deliverable D5, "Process Evaluation Results, Achievements and Project Conclusions", July 2001.