

Component Search Service and Deployment of Distributed Applications

D. Kebbal and G. Bernard

Institut National des Télécommunications - Département Informatique

9, rue Charles Fourier 91011 Évry Cedex - FRANCE

{Djemai.Kebbal, Guy.Bernard}@int-evry.fr

Abstract

This paper presents and discusses some infrastructure services necessary for distributed applications development. The main service considered is the component search service which includes traditional naming and trading-like services. A number of studies have shown recently that current trading, based on interface type conformance, is insufficient for component-based applications. These studies have concluded that semantic information, describing especially the behavior of the component and its context dependencies, is required. This type of trading is called semantic trading. In this paper, we address this problem and propose a component search service model based on semantic trading. Furthermore, we discuss the integration of the service in a deployment tool, through an enhanced deployment process early proposed by Corba Component Model.

Keywords: *distributed applications, software components, trading, semantic trading, component search service, application deployment.*

1. Introduction

Developing scalable distributed applications requires the use of tools allowing inter-operability between applications or parts of applications. This cooperation scheme is usually achieved through the notion of services provided by some applications to other applications (client/server model). The Common Object Request Broker Architecture (CORBA) [8] is a well-known standard providing such cooperation model. Search (naming and trading) services provided by the majority of the implementations of the standard are tools for storing the characteristics of services provided by some suppliers and finding them to application clients wishing to use them dynamically. The search approach is based on the *conformance between the type and sometimes the properties of the supplied and the requested services.*

The emergence of component-based approaches renders

this type of trading inadequate [12]. Component-based models allow to build distributed applications by assembling some software components. Components are characterized by well-defined interfaces, context dependencies and total transparency of their implementation. However, though these interfaces are well-defined, this is insufficient for choosing the appropriate components to integrate during the application deployment. Component interface description must be complemented by semantic information describing the functioning of components. The so-called semantic trading is therefore required [12].

The remainder of this paper is organized as follows: in the next section, we introduce the component model. Section 3 discusses the trading service within the middleware framework. Section 4 presents the specificities of the component trading and introduces the semantic trading. In the section 5, we propose a tool for locating components, a component search service, based on semantic trading. Section 6 shows the utility of the proposed component search service as well as other system utilities in the deployment process of distributed applications. Finally, section 7 concludes this work and presents some open issues.

2. Component model

A software component is a unit of composition with explicit specified interfaces and context dependencies. A component may be deployed independently and is subject to composition by third parties [11]. Components are therefore basic units for building distributed applications. Component development is generally achieved at three different levels or steps: design, implementation and integration (assembly) in order to construct applications. Design step consists of describing the component's functionalities through the specification of the component's ports (interfaces, events, etc.). Implementation step, on the other hand, consists of writing the internal code of the component. Then, the component can be deployed (instantiated in a server) and may be assembled with other components

within the framework of a distributed application. These steps form the common application development process focusing on the *component reuse* aspect. Of course, this process can be more complex when we consider advanced features like packaging, installation, personalization (configuration), etc. Figure 1 presents the structure of a Corba Component Model (CCM) component [9], which is, at our consideration, the most complete component model specification until now. Indeed, CCM includes many concepts and features not supported by other component models such as EJB and COM+. These differences include, in particular, the *multiple interfaces* aspect which is not supported by the EJB model, and the *component categories* very restrictive in the two models, especially in COM+. Moreover, COM+ is a proprietary model (Microsoft) in which the application design follows a three-tier (client, server, database) architecture. Similarly, EJB is a language-dependent (Java) model. A major contribution of CCM derives from standardizing the component development cycle using CORBA as its middleware infrastructure [15]. CCM components provide some types of features called *ports* which express the integration (composition) capabilities of the component [6]. These features are:

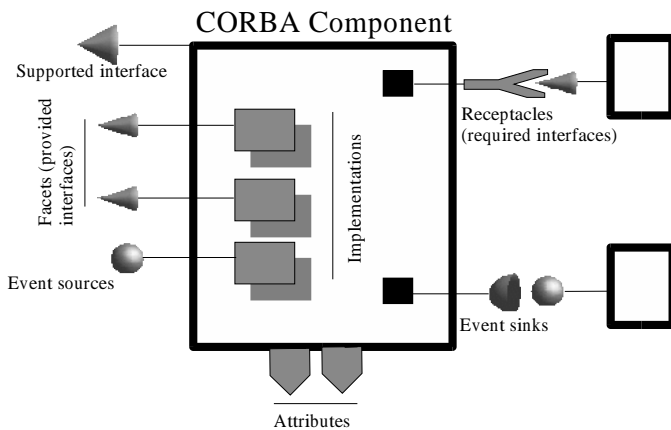


Figure 1. CCM component structure.

- *provided and required interfaces*: determine the component context dependencies and provide a synchronous cooperation model between components;
- *supported interfaces*: reflect the inheritance aspect of components. They represent interfaces from which the

component's equivalent interface¹ inherits;

- *event sources and sinks*: provide an asynchronous interaction model between components;
- *attributes*: allow personalizing (configuration) of the component.

CCM inherits advantages of CORBA such as platform and language independence. Figure 2 presents a more detailed structure of Corba components.

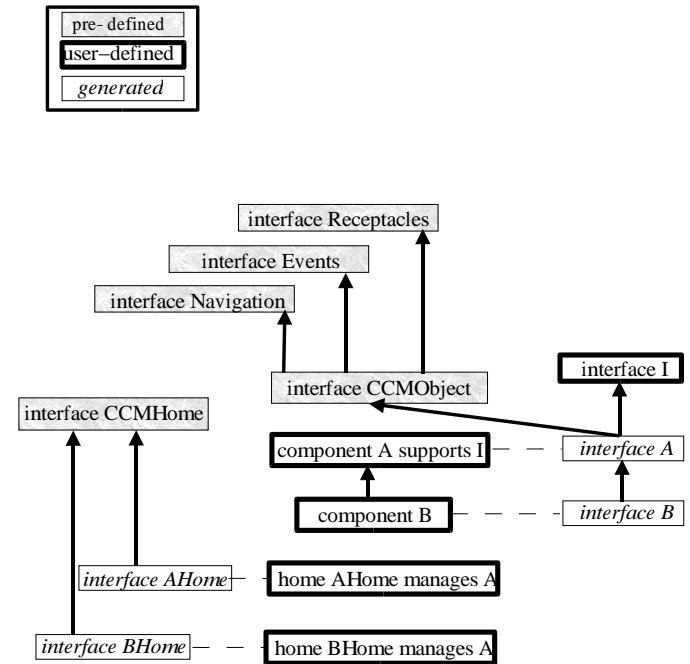


Figure 2. Component inheritance in CCM.

3. Trading

The aim of the trading function is sharing information and services in distributed systems. A trader is an object

¹The component's equivalent interface represents the base reference from which the component is accessed, especially from component-unaware customers [15].

that supports the trading service in a distributed environment. It can be viewed as an object through which other objects can advertise their capabilities and match their needs against advertised capabilities [10]. The trading allows customers to discover and select services at run time and bind to them dynamically. These services are early exported by suppliers. In the following, we present some trading systems and discuss some of their related aspects.

3.1. Trading in ANSA

The trading in the ANSA system is defined as follows: the activity of choosing services, such that they match some service requirement. The choice is based on the comparison of the specification of a service required (provided by a prospective consumer) and the service specifications supplied by service providers or their agents [2]. An ANSA trader is defined as an object performing the trading activities based on the entire or on a subset of the service offers. The trader has not a special status, it is considered like any other object [2]. A trading service is not the only service that can store knowledge about and references to other services; on the contrary, any object and service can do this.

Service offers selection is based on conformance of service type and a constraint expression which must characterize the required service.

3.2. Information exchange in UDDI

Universal Description, Discovery and Integration (UDDI) defines a framework for publishing and discovering information about web services [13]. Considered services consist of specific business functionalities exposed by companies in order to be used by other companies. UDDI adopts an approach based on the so-called *distributed registry* which includes businesses and their service descriptions. Built on SOAP (Simple Object Access Protocol) [14] and XML, UDDI intends to ensure interoperability through a cross-language and cross-platform approach. Service description in UDDI must provide three kinds of information: white, yellow and green (technical) pages-equivalent information, like the OMG specifications [7]. UDDI business registry provides a framework for exchanging service information on Internet. The UDDI registry is a distributed service with multiple root nodes that replicate data within each other. Service description consists of four types of information: business information, service information, binding information, and information about specifications for services. These types are structured in a hierarchy and correspond to the following entities:

- *businessEntity*: serves as the top-level information manager concerning a business unit. It includes sup-

port for yellow pages taxonomies (particular industry, product category, geographic situation).

- *businessService*: is a sub-structure of the businessEntity structure. Its role is to keep information related to either a business process (purchasing services, shipping services, etc.) or category of services.
- *bindingTemplate*: within each businessService live one or more technical web service descriptions in a bindingTemplate structure (connection address of the service, etc.).

To discover a web service using UDDI, the first step is to locate the businessEntity information of the service. The programmer can drill down for more detail about a businessService or request a full *businessEntity* structure. The programmer then selects a particular bindingTemplate and uses it for contacting the service.

As we can see it, the trading in those systems is based on the notion of service represented by an interface. Searching for service offers requires the use of a matching approach which allows the trader to select the well-suited service offer corresponding to a requested one.

3.3. Service offers selection

An importer uses a trader to search for services matching some criteria. A service type, which represents the information needed to describe a service, is associated with each trader service [10]. It comprises an interface type and some named property types. Services types can be structured into a hierarchy reflecting interface type inheritance and determining the substitution rules between service types.

Service type conformance allows clients wishing to use a service which they can't find to substitute it for another service which conforms to the requested one.

Conformance in ANSA is based on the *no surprises rule* in which an interface is substituted for another supporting all its operations. Moreover, each operation of the second interface exists in the first one with the same name and its type conforms. Operation conformance means that they have the same lists of parameters and the same termination conditions [2].

4. Component Trading

Component-based application development requires a component description language like IDL3 [9] or CDL [5]. This kind of languages describes the context of components (required/provided interfaces, emitted/received events, etc.) and semantic relationships between component specifications and implementations (e.g. component *A* is equivalent

to B under condition C [5]). In conclusion, components are characterized by a number of properties making the classical interface type conformance inadequate. These properties can be summarized in the following:

- *Context dependencies*: as we have mentioned it, a component is characterized by its context dependencies expressed by ports (interfaces, events, etc.). Component selection must take into account these dependencies;
- *Complexity and transparency of component implementation*: a component is described by its ports which are interfaces. A user who wants to use the component must give a precise description of the service on an interface type. The component selection is therefore based on a syntactic information. Due to the complexity of components and the diversity of domains, this type of information becomes insufficient to characterize a component. Semantic information, describing the component functionalities and its behavior, must be then used. Component behavior includes pre/post conditions, invariants, etc. which may be specified in the search process. The behavioral information is generally extracted at the design phase of components;
- *Component conformance or compatibility*: the substitution of services in application-based components is therefore based on the component compatibility. That is all conditions under which a component may be substituted for another. This aspect will be developed below.

These differences, especially the consideration of the semantic aspect of components in the search phase, have motivated the development of the so-called semantic trading [12].

4.1. Component compatibility and semantic trading

In component-based approaches, the matching is based on component compatibility rather than interface type conformance. Component compatibility includes conditions under which a component may be substituted for another. These conditions can be expressed using *reuse patterns* like those based on contracts. De Hondt [4] states that beyond the required and provided interfaces compatibility, the correct interaction behavior must be considered as well. Reuse contracts express the assumptions each participant makes about its *acquaintance* which document all inter-operation dependencies and emphasizes especially on call-backs of operations to original component. For example, if component A invokes an operation o_1 on a required interface of a component B , and o_1 calls in turn an operation o_2 on A ,

then this dependency must be respected in any composition of A with another component providing that required interface. Reuse contracts track such dependencies and allow components to deviate from the reuse contract based on a developer documentation which specifies how they deviate exactly. Reuse contracts are then subject to reuse operators (modifiers, like extension, refinement, ...) having their applicability rules [4]. Reuse operators allow the detection of conflicts at the composition phase and ensure an appropriate composition of components.

Terzis suggests that behavioral information of components extracted generally at system design must be used to construct an *ontology of the system's design* [12]. Generally, the concepts of the ontology are the concepts (types) identified during the system design and their relationships can be directly mapped into the ontology. The ontology provides a basis for a knowledge base which encodes the system design and ensures a *common vocabulary* between the components. Design knowledge, combined to a number of rules which capture composition patterns, determines the *semantic inter-operability constraints* [12].

The ontology is a number of terms with their definitions and a specification of their relationships [17]. The relationships are a formal specification between the terms of the domain (part-of, subclass-of, etc.). Ontologies provide a framework for expressing design concepts of a system. These concepts are generally stored in a knowledge base. To permit interaction and information exchange between different domains, a standardization of terminology must be done. This can be achieved by integrating ontologies of different domains. People from different domains can thus interact and interoperate without being forced to be consistent with the concepts of others.

Based on this kind of trading, Terzis proposes a trader model associated with a composition service. The trader receives requests and replies by a composition of the retrieved components, which is performed by the composition service. A number of pre-processing components are provided in order to support multiple request expression languages and to transform requests in the basic format. The transformed request goes then from a translator which transforms it to a number of alternative composition strategies (assemblies). For each strategy, a number of queries for the trading space is formed. The query formation process is supported by the terms of the ontology which must ensure semantic inter-operability through query expansion or contraction.

[3] discusses some propositions on reuse in knowledge-based systems using CORBA. Again, they bring out the syntactic aspect of CORBA and argue that knowledge-based systems have addressed the component design, integration and semantics. They focus on the fact that reuse

implies that components must be described in an abstract fashion in order to allow developers to find and integrate the appropriate components. They argue that these abstractions must be represented in ontologies. Furthermore, they discuss interface adapting if the specifications do not match exactly. Interface adapting consists of adapting the outputs of one method to inputs of another method (adapting one IDL specification to another).

In conclusion, for a component to be reusable easily, it must be quickly found and understood by developers. Semantic trading which emphasizes on the functionality of components contributes significantly to achieve these objectives.

5. Component search service

In this section, we describe a search service we are currently developing within the framework of a project called CÉSURE. CÉSURE is a project where the objective is to conceive a platform for *applications of service* provided to mobile users [1]. CÉSURE applications are distributed applications focusing on the mobility of users through the smart card support and developed using a component-based model. CÉSURE is intended to provide tools for development, deployment and maintenance of applications. A number of system utilities are developed in this scope. This includes component search, monitoring, fault tolerance (component replication, failure detection, application reconfiguration, etc.) and deployment tools.

In CÉSURE, the deployment of an application consists of searching component packages, active instances and hosts on which component implementations must be instantiated. For example, a user which has subscribed to a weather service, may wish to use his service from a mobile terminal. When it connects, the search tool must search for a component corresponding to a man/machine interface which must be instantiated on the user's terminal. Moreover, an active component corresponding to a weather server must be found by the search tool. Therefore, the search tool must be able to search for component packages, instances as well as hosts on which components will be instantiated.

5.1. Component search service architecture

In this subsection, we present the architecture of the component search service which we propose. The aim of the service is to respond to component search requests whatever their form. In other words, the tool serves as naming and trading services at the same time. The semantic trading is used. Furthermore, since distributed application deployment may use already activated component instances, the

service must be able to distinguish these requests and search for active component instances rather than for implementation packages. Figure 3 shows the main components of the tool:

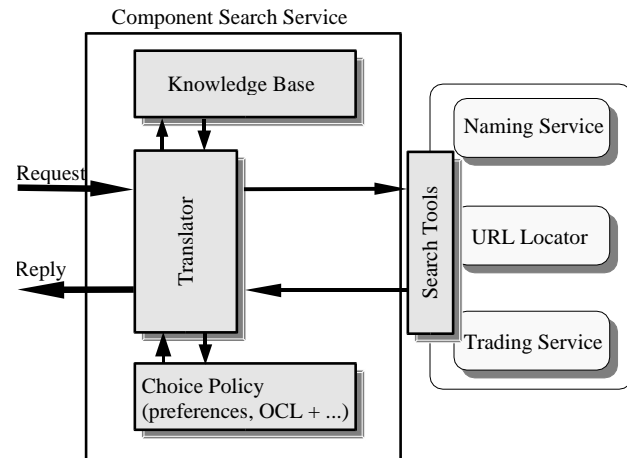


Figure 3. Component search service architecture.

- *Translator*: this is the main component of the service. It is intended to receive requests and reply to the client (a deployment tool in general). The translator uses the knowledge base to transform original requests into well-suited, easily understandable and complete requests using the ontology stored in the knowledge base. Note that, syntactic and semantic informations are combined in the search process.
- *Search Tools*: they represent the search space. They consist of some classical search tools in which service matching is based on interface type compatibility. This includes CORBA trading and naming services, as well as a tool for locating component packages through URLs. So, component packages, instances or interfaces can be searched by the same tool. Moreover, the search criteria may be a description of the entity (component package, service), a symbolic name or an absolute location of the entity. Following the case, the trading, the naming or the URL locator is used.
- *Knowledge base*: the Knowledge-base constitutes a base in which all informations (ontologies) are kept. In addition, it includes the matching rules between terms from different domains. The matching rules constitute thus an integration of ontologies encoding the concepts

of those domains. Moreover, domains may be hierarchically structured and refined following inheritance relationships.

- *Choice policy*: this component allows the search service to choose among many matching responses following the request preferences. These preferences can be expressed using an Object Constraint Language (OCL) [16]. The choice may involve the component reuse constraints based on some formalism like the reuse patterns model. Finally, the matching offers found can be stored by this component. So, the application may choose to use another found component by changing the preferences or the reuse constraints. This constitutes a framework for a *caching mechanism* in case of large scale environments.

5.2. Registering and searching for components

The component search service provides a number of service interfaces like an OMG trader. These interfaces are intended in particular to register and retrieve component offers and to manage the search server. As in the traditional traders, a component must initially be registered within the search service before being searched and used by clients. During the registration phase, a component description is provided. The component description consists of its context dependencies in a syntactic form as well as its semantic description. The search phase uses the entire or a part of the previous information to retrieve the component. The following IDL describes a prototype for a component search service. The main search criteria structures are shown.

```
/* Component's context dependencies */
struct ComponentFeatures {
    ComponentName name;
    ComponentNameSeq inherits_components;
    InterfaceFeaturesSeq supported_interfs;
    Ports ports_descr;
    AttributeNameSeq attributes;
    CosTrading::PropertySeq component_props;
};

/* Component's semantic information */
struct ComponentSemantics {
    Domain ont_domain; /* ontology */
    ComponentCharacteristics component_descr;
    ComponentBehavior component_behav;
    ComponentReuseConditions reuse_conds;
    ComponentFunctionalities component_funcs;
};
```

The description is well-adapted to the CCM model. The component is described by two structures: component features and component semantics.

5.2.1 Component features

Component features structure includes a syntactic description of the component, in particular its context dependencies. It consists of the following fields:

- *name*: the name of the component. It indicates in some way the type of the component. Components can then be structured hierarchically following the inheritance property, like service types in OMG traders.
- *inherits_components*: a sequence of component names (types) from which the component inherits.
- *supported_interfs*: a sequence of interface descriptions concerning the supported interfaces. The structure `InterfaceFeatures` comprises a syntactic description of an interface, in a traditional-like description manner. The syntactic description includes the interface type and some named properties.
- *ports_descr*: a syntactic description of all component's ports (required, provided interfaces as well as emitted, published and consumed events). Each port, materialized by an interface, is described by an `InterfaceFeatures` structure similarly to the supported interfaces.
- *attributes*: component's attributes names, which express the component configuration and personalization capabilities.
- *component_props*: a component may be characterized by a set of properties which correspond to the component equivalent interface's properties. During the search phase, a constraint, applied to those properties and expressed in a constraint language like OCL, may be specified.

5.2.2 Component semantics

Component semantics structure includes a description of all semantic aspects related to the component. This comprises, in particular, behavioral description of its ports and a description of its functionalities. It consists of the following fields:

- *ont_domain*: determines the ontology by identifying the domain from which the request is originated.
- *component_descr*: It describes the component characteristics: the component nature and its technical properties in particular. For example, a particular component implementation may be described by a number of properties and utilization constraints: operating system, runtime, compilers, etc.

- *component_behav*: expresses the behavior of the component. Behavioral information consists, in particular, of invariants and pre/post conditions. For each component port (interface), a number of invariants may be specified and for each operation of the interface, a number of pre/post conditions can be defined. The behavioral information specification may be expressed in an Eiffel-like language.
- *reuse_conds*: this structure is kept for eventually expressing the reuse conditions of a component. These conditions may be inspired from the reuse contracts studied by De Hondt [4].
- *component_funcs*: describes component functionalities in a textual or other form. Component importers may use this possibility to search for some component by describing its functionalities. When a component is obtained, its context dependencies including interface description, syntax and behavioral conditions are known. So, the component can be safely used.

5.2.3 Component search scenario

When a request is submitted to the search server, it goes through the translator, the heart of the model. The translator transforms the request, rich semantically, to a syntactic request, using the matching rules of ontologies stored in the knowledge base. The resulting request is then likely to be understood in other domains and adapted to the traditional search forms.

Matching rules stored in the knowledge base and resulting from ontology integration are used in a dictionary usage manner. This allows translating terms originating from a domain in another domain. So, requests initiated from a domain can be understood and satisfied in another different domain, without user intervention.

All the search criteria of components in the search request are optional. The search may concern a part of a component description. So, clients may provide only a description of the component's functionalities to access the full semantic and syntactic characteristics of the component. This opportunity may be very useful in the development phase of applications.

6. Component search service and deployment

The deployment phase is the last step in the development process of applications. The deployment of a distributed application consists of installing and instantiating its components on the hosting servers, configuring them and assembling them following the application assembly specification. In many cases, some components used in the assembly are already active, their instantiation is therefore

unnecessary. The component search service is responsible for identifying and locating these components. The search service is typically used by a deployment tool in order to search for appropriate components and sometimes to optimize their deployment. Deployment tools, provided by an ORB or other vendors, vary from simple tools which just guarantee the deployment constraints, to more elaborated tools which ensure more advanced properties, like component location services and load balancing.

6.1. Deployment process in CCM

In CCM, the deployment is intended to hook-up a so-called logical component topology to a physical computing environment [9]. In CCM, the deployment is triggered by logical component assembly specified in an assembly file, and concerns components and component homes.

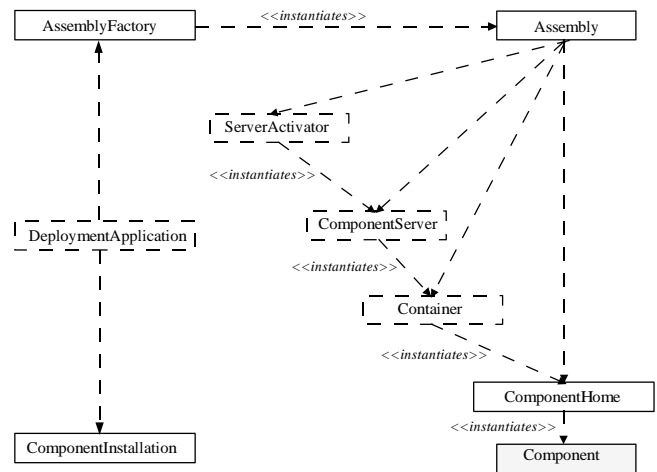


Figure 4. CCM deployment architecture.

In the following, we summarize the basic steps which contribute in the deployment process in CCM [9]:

1. Determine the target host configuration, on which components will be installed and instantiated. This is done most likely in collaboration with the user. Components are deployed in some cases following a process or host collocation constraints;
2. Install non-already installed component implementations on appropriate platforms;
3. Instantiate components and component homes on particular hosts. This can take into account the process and host collocation constraints;

4. Connect components as specified in the application's assembly file.

Figure 4 illustrates the architecture of the deployment tool specified by CCM and the relationships between the objects participating in the application deployment process. The deployment is carried out by a *deployment application* and some helper objects (component repositories, assembly and component factories, containers, etc.).

6.2. Deployment process in CÉSURE

In this subsection, we will show how the infrastructure services are used at the deployment phase of the distributed application in CÉSURE.

Although the deployment process is well specified and standardized in CCM, this specification presents a number of drawbacks including:

- poorness of the mechanism which does not address some issues like the localization of component implementations and the management of host configuration;
- vagueness of the deployment process which does not fix a number of details such as the localization of helper objects (Assembly, ComponentInstallation, etc.). This is caused, in our opinion, by the fact that the process is not implemented and tested.

The additional services (component search and monitoring services) can be considered as Common Object Services (COS) and can be retrieved by the ORB, like in standard CORBA programming. Therefore, these services may be centralized, distributed or semi-distributed. The deployment tool may be unaware of their localization. However, an important issue is that the infrastructure must optimize the use of these services, by finding the nearest service agent in case of a distributed architecture for example.

Figure 5 depicts out the enhanced deployment process which we propose for CÉSURE applications. The main steps of CCM deployment process are adopted and adapted. Furthermore, some additional steps are included following the contribution of the additional provided services. The deployment process proceeds as follows:

1. The **DeploymentApplication** is started. It begins by locating the component search and monitoring servers. Then, it addresses the search service in order to get a host configuration on which components will be instantiated. The location of components is either specified by the application in the assembly descriptor file, or searched by the component search server. The location of hosts is determined:

- by the application in the assembly file,
- or by the search service in conjunction with the monitoring server following some constraints.

Component search and monitoring servers cooperate in order to keep an up-to-date state information on hosts. The host configuration management is ensured by the monitoring server which supervises host activities (load state, etc.) and informs systematically the search server of the state evolution of the configuration. So, the search process for hosts may involve precise load information in the selection criteria.

2. The second phase in the deployment process is the installation of the retrieved and non-installed component implementations on the appropriate platforms. This is done by calling the **install** operation on the **ComponentInstallation** object. The parameters of this operation are the implementation identifier (*id*) and a string denoting the component file location.
3. After the component installation phase is complete, the deployment application creates an **Assembly** object. This is done by calling an **AssemblyFactory** object on the host where the assembly object must be created. The parameters of this operation are especially the assembly descriptor file location (steps 3 and 3.1 on the figure). The role of the assembly objects is to coordinate the creation and destruction of component assemblies.
4. The next step consists of creating the assembly by connecting component instances. Before connecting the components, component and home instances must be first created. Component instances creation is orchestrated by the *Assembly* object on the basis of collocation constraints and on host configuration obtained at the first phase.
5. In order to create a component, the *Assembly* object must create a component server, create a container within the server, install a home object within the server and then use the home to create the component [9]. This procedure is carried by a **ServerActivator** object which resides on the host where the component instance is to be created². Thus, the component creation is achieved in the following steps:
 - (a) The *Assembly* object calls the **create_component_server** operation on the *ServerActivator* of the host on which the component is to be created. An empty server is

²There is one *ServerActivator* instance on each host. Its reference must be known by the *Assembly* object.

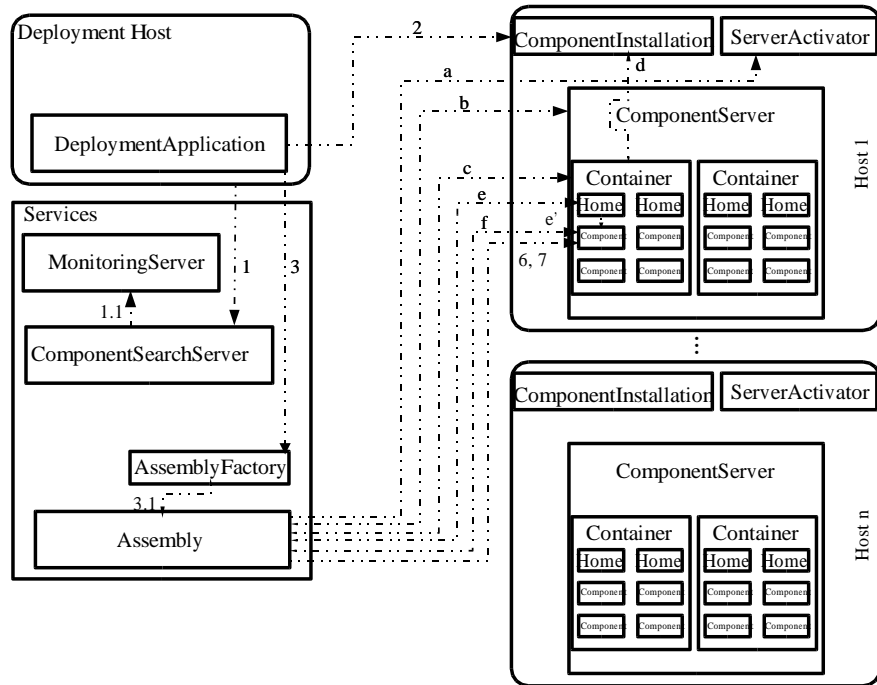


Figure 5. Deployment steps in CÉSURE.

then created and the **ComponentServer** object's reference is returned.

- (b) When the *Assembly* object obtains the reference of the created *ComponentServer*, it uses it to create a **Container** within the server. This is achieved by calling the **create_container** operation on the *ComponentServer* object. The container identifier is passed to the operation which creates the container and returns a reference to its interface.
- (c) The next step consists of installing the **ComponentHome** within the container. The *Assembly* object calls then the **install_home** operation on the Container interface, giving it the identifier of the component. The operation returns a reference to the created component home interface.
- (d) In the *ComponentHome* creation operation, the container calls the **get_implementation** operation of the *ComponentInstallation* object. This operation gets a component implementation identifier and returns the absolute location of this implementation. The container then loads the implementation and instantiates the **ComponentHome** object.
- (e) The next step consists of creating the **Component**. This is accomplished by the *Component-*

tHome object, when the *Assembly* object calls the **create_component** operation provided by its interface. The return value of the operation is a reference to the created component (of type **CCObject**).

- (f) The component is then configured, if required.
6. When all components are created, the *Assembly* object connects them based on the information specified in the connect block of the assembly descriptor. The component connection is accomplished by calling the receptacle connect operation of the **CCObject** reference.
7. After that, the *Assembly* object calls **configuration_complete** on each object in the assembly to signal that all its initial connections have been fixed.

7. Conclusion

The development of large scale distributed applications requires some interoperability level, in order to allow heterogeneous applications to cooperate. Search tools like naming and trading services provide a framework facilitating this interoperability, through the notion of service offers exported by some suppliers and made at the disposal of

service importers or clients. Search approaches are generally based on the conformance of the type and sometimes the properties of services. However, in component-based models, this type of search becomes inadequate. Indeed, component-based distributed applications are constructed by assembling some software components. Matching is therefore based on component substitutability which express the conditions under which a component may be substituted for another. These conditions include syntactic properties characterizing component dependencies (interfaces, events, etc.) as well as semantic properties, including information on component functionalities and reuse conditions. This is even more significant in large scale environments, in which terms from different domains may be subject to ambiguities and misinterpretation.

Search service is an important tool facilitating the distributed applications development, deployment and interoperability. Therefore, developing and experimenting search services based on semantic trading is very important, in order to measure the efficiency, feasibility and performance of the approach. Indeed, semantic trading constitutes an essential framework for developing and supervising distributed applications especially in large scale environments

The assembly, created in CCM deployment steps, specifies an initial configuration and does not address the evolution of the component connections. The component connections may change following the application evolution and underlying environment state changes. These changes include especially failures, load unbalance, etc. The application must be able to adapt to these situations and to continue to evolve. Therefore, tools allowing to keep these properties must be considered in the deployment and supervision process of applications.

Within the framework of the component search service, other issues and problems must be addressed. This concerns many aspects and in particular the versioning problem. Indeed, when multiple versions of the same component may co-exist, the search server must be able to choose the appropriate versions among all the available ones. A solution to this problem may be the consideration of the version property, which must be specified by the client, in the search criteria. Another solution may be a policy used by the search service which allows the automation of this task, such as the choice of the most recent version systematically.

References

- [1] Cesure. Infrastructure système. D5, Novembre 2000.
- [2] J. P. Deschrevel. The ANSA model for trading and federation. Technical Report ANSA phase III APM.1005.01, Architecture Projects Management (APM) Ltd., Cambridge, UK, July 1993.
- [3] J. H. Gennari, A. R. Stein, and M. A. Musen. Reuse for knowledge-based systems and Corba components. In *10th Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW'96)*, Banff Canada, November 1996.
- [4] K. D. Hondt, C. Lucas, and P. Steyaert. Reuse contracts as component interface descriptions. In *WCOOP'97 - 2nd International Workshop on Component-Oriented Programming*, pages 43–49, Turku Centre for Computer Science, 1997.
- [5] J. Kiniry. CDL: A component description language. In *submitted to the Advanced Topics Workshop (ATW) of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, March 1999.
- [6] R. Marvie and P. Merle. CORBA: From objects to components. In *Tutorial for the 14th European Conference on Object Oriented Programming (ECOOP'2000)*, Nice & Cannes, France, June, 2000.
- [7] The Common Object Services Specifications, Volumes 1 & 2: Architecture and specification - revision 2.0. John Wiley and Sons Inc., 1994.
- [8] The Common Object Request Broker: Architecture and specification - revision 2.0. OMG TC Document 97-02-25, July 1996.
- [9] Corba Component. OMG Documents orbos/99-07-01 orbos/99-07-02 orbos/99-07-03 ptc/99-10-04 Joint Revised Submission, 1999.
- [10] Trading object service specification. Associated OMG Documents - Trader Service V1.0, May 2000.
- [11] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
- [12] S. Terzis and P. Nixon. Component trading: The basis for a component-oriented development framework. In *Workshop on Component-Oriented Programming (WCOP'99), European Conference for Object-Oriented Programming (ECOOP'99)*, Lisbonne, Portugal, June 1999.
- [13] Universal Description, Discovery and Integration. UDDI technical white paper. UDDI white papers, <http://www.uddi.org>, September 2000.
- [14] W3C. The simple object access protocol (SOAP) 1.1. W3C Note 08 May 2000 <http://www.w3.org/TR/SOAP>.
- [15] N. Wang, D. C. Schmidt, and C. O'Ryan. *Component-Based Software Engineering: Putting the Pieces Together*, chapter An Overview of the CORBA Component Model. Addison-Wesley, 2001 (still).
- [16] J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 13, May 1999.
- [17] G. Wiederhold. Interoperation, mediation and ontologies. In *International Symposium on Fifth Generation Computer Systems (FGCS94), Workshop HCKB*, volume W3, pages 33–48, Tokyo, Japan, December 1994.