An Experience on CORBA Component Deployment

Maria Claudia B. Barros, Edmundo R. M. Madeira, Irineu Sotoma
mcbb@terra.com.br, {edmundo, isotoma}@ic.unicamp.br – Institute of Computing, UNICAMP
University of Campinas, Av. Albert Sabin, 1251 - CEP 13083-970 Campinas, SP, Brazil

ABSTRACT

The component technologies have been more and more developed and used to speed up the development of (distributed) applications. OMG has developed a model for CORBA components, which briefly describes the deployment of CORBA components without many details. This paper presents and discusses some implementation aspects and experimental results obtained by using an extended deployment model for CORBA Components previously proposed by the authors.

1. Introduction

A software component has well defined interfaces and properties, which facilitate the configuration and connection with other software components. By using components, the application developer only builds the business semantic and connects it to the existing components. The infrastructure to distributed communication, events, transactions, security, naming, and other services, are encapsulated in the components. By this way, it is easy to develop and maintain a business application and speed up the application development cycle.

An important part of the use of components is the deployment. The deployment of components consists in installation, instantiation and connection of components. The OMG (Object Management Group) has generated a component specification to CORBA (Common Object Broker Architecture) called CCM (CORBA Component Model) [4], which does not detail the CORBA Component deployment implementation.

This paper presents some implementation aspects and experimental results obtained through the implementation of an extended model for deployment of CORBA components. The model was detailed in [10] and this paper comments the model participants, an extension to the deployment process and a use case of the proposed model.

This paper is organized as follows. Section 2 presents some basic concepts about CORBA components, and discuss some related work. Section 3 shows the details of the proposed deployment model. Section 4 discusses some implementation issues of the proposed model on TAO ORB (Object Request Broker), shows a use case, describes experiments and discusses results of them. Finally, Section 5 offers some conclusions.

2. Basic Concepts

2.1. POA (Portable Object Adapter)

The POA (Portable Object Adapter) [5] was specified by the OMG for making the object implementations portable between different ORB products. Additionally, POAs have policies which allow the application developer to control threading management, *Object Id* management, and servant creation, destruction, activation and deactivation. The following policies are important to this paper understanding: *Request Processing Policy* (`USE_ACTIVE_OBJECT_MAP_ONLY`, `USE_DEFAULT_SERVANT`, `USE_SERVANT_MANAGER`), *Servant Retention Policy* (`RETAIN`, `NO_RETAIN`), *Lifespan Policy* (`TRANSIENT`, `PERSISTENT`) and *Id Assignment Policy* (`USER_ID`, `SYSTEM_ID`).

2.2. CCM specification

Component types are defined using the CIDL (Component Implementation Definition Language), which is an extension to the IDL (Interface Definition Language), and they are represented in the Interface Repository. A component is denoted by a component reference, which is represented by an object reference. Components are packaged in zip archive files, which have XML (Extensible Mark up Language) descriptors, which describe their contents. A component package comprises one or more implementations of a component. Each component package contains a component implementation. A component implementation is a DLL (Dynamic Linking Library), a shared library, or a Java .class file.

Deployment Architecture. CCM describes a deployment architecture, with a class diagram (showed in Fig. 1) and interactions between the pieces, which requires objects of the proposed interfaces `ComponentInstallation`,

AssemblyFactory and Assembly, and supposes the use of a deployment application. The other interfaces are included only for illustrative purposes. The steps in deploying and activating a component assembly could unfold as follows: 1) The deployment application interacts with the user to determine where each component is to be placed; 2) The deployment application asks the ComponentInstallation object, which retrieves the component file and optionally makes it available in the local environment; 3) The deployment application creates an Assembly object by calling an AssemblyFactory object on the host where the assembly object is to be created. Assembly objects coordinate the creation and destruction of component assemblies; 4) Based on the information in the assembly descriptor file, the Assembly object creates each component and component home and hooks-up the assembly; 5) In creating a Component, the Assembly object uses the ServerActivator, ComponentServer, Container, and the ComponentHome objects; 6) The ServerActivator object, on the target host, creates the ComponentServer object.; 7) The ComponentServer object, on the target host, creates containers within the ComponentServer. A container is referred by a reference to the Container interface; 8) The Container object on the target host installs the ComponentHome object into the container; 9) The Container object creates the ComponentHome object; 10) The ComponentHome object creates the Component instance; 11) A configurator could be applied to the component; 12) After the component installation, the Assembly object connects components in the assembly based on the information in the assembly descriptor file; and 13) After the successful connection among all components, the Assembly object signals each object that all of its initial connections have been fixed.

2.3. Related work

The literature has several works on component models. However, works on CCM and CCM deployment are minimum yet. Wang et al. [8] suggested applying reflective middleware techniques by using several design patterns to resolve some key design challenges for QoS-enabled (Quality of Service-enabled) CCM implementations. Kebbal and Bernard [2] extended the CCM deployment process because they have noted mainly two drawbacks in the CCM specification: it does not address issues like localization of component implementations and the management of host configuration; and it does not fix a number of details such as the localization of helper objects (Assembly, ComponentInstallation and so on).

Rudkin and Smith [6] defined a service creation and deployment process of Java applications that use Java Beans components. Mikic-Rakic and Medvidovic [3] dealt with several deployment problems which they had identified. They used a software architecture called Prism. The authors did not explicitly mentioned what type of standardized components (COM, Java Beans or CCM) Prism supports. However, it seems each of these could be supported if the Prism style is followed. Hissam et al. [1] suggests a new approach, called PECT (Prediction-Enabled Component Technology), to technology components, which integrates component technology with analysis model. By using PECT, consumers may obtain measurably bounded trust in both components and design-time predictions based on the use of these components.

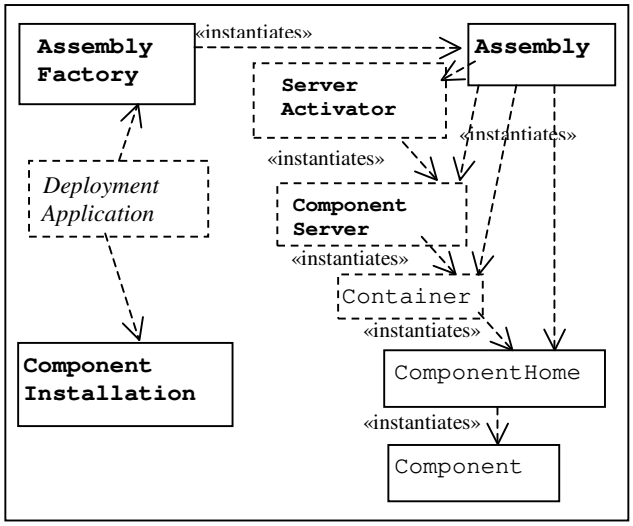

Fig. 1. Dependencies among the *deployment architecture* pieces proposed in CCM

3. The Proposed Deployment Model

This section details the proposed model for deployment and presents a use case.

3.1. The Proposed Model

Our deployment model proposes a framework for component installation and activation [10]. The main differences between our proposed model and the CCM deployment specification are: a class diagram which details the parties involved in the deployment process, a new repository called *Component Assembly Instance Repository* (CAI) and a modified deployment process. Our proposed model has the following participants: ***repositories***, ***user information***, and ***processes***.

Repositories of the proposed model. The repositories of the proposed model are used by the processes and they form the user information. The *Interface Repository in CIDL* is formed by interface repository objects organized hierarchically as a containment tree. The Component Repository is a type that derives from Repository and it is used as a top-level container for the IRs that store component information. The *Implementation Repository* contains all the information necessary to locate and activate component implementations. Since the components will be accessed via their equivalent interfaces, the Implementation Repository for Components is very similar to the Implementation Repository used to locate/activate object implementations. The main difference is instead of storing the association between an object type and a server executable, the Implementation Repository stores the association between a component definition and a dynamic library, which will be placed into memory. The *Assembly Archive (AAR)* is composed of files CCD (Corba Component Descriptor), CAD (Component Assembly Descriptor) and CPF (Component Property File). The content and format of those files were defined at the CCM specification. The *Component Assembly Instance Repository (CAI)* stores the information of each component assembly instance deployed: component assembly identification, features of each component instance that are part of the assembly, the connections between components through events, and the connections between components through ports. Each assembly instance must have its own component instances. Component instances are not shareable among assembly instances. Each CAI Repository entry corresponds to a component assembly instance deployed.

User information in the proposed model. The user information has two pieces. The *User input* corresponds to: *Components* chosen by the user to form the whole application; *User input for Assembly*, which describes how the chosen components will be "hooked-up"; and *User input for Deployment*, which informs on which host each component will be deployed. The *User output* corresponds to the `AssemblyId` (unique name of the deployed assembly) and `ComponentIds` (component instance identifier of each deployed component). The user can use this information to access/localize the component instances that belong to the assembly.

Processes of the proposed model. Fig. 2 shows a class diagram which shows the dependencies between the classes that implement the processes of the proposed model. The *Builder Tool* shows a list of Components (based on Interface and Implementation Repositories) and gets the User input information. Afterwards, it generates an Assembly Archive (AAR). The *Populator of CAI* reads the AAR file and creates an entry in the CAI that represents the assembly being deployed. The *Deployment Manager* coordinates the deployment process. It reads the necessary information from the CAI Repository, and interacts with the other managers and servers in order to create the deployment infrastructure in each machine. Additionally, it stores in the CAI the information returned by the manager servers. The *Installation Manager* distributes the libraries that implement the components and their property files (if exist) to the machines where the components will be deployed. The *Activation Manager* asks the Activation Server in each host requesting the creation of the references for the component instances being deployed in the host. The *Activation Server* creates and maintains the POAs that generate the component instance references, and manages the lifetime of the servants that incarnate those component instances. The *EventServer* creates the eventChannels for the component assembly, and "hooks-up" the producer(s) and consumer(s). Components can receive references to other objects and use those references to invoke operations. The Components can provide internal object references which are called facets, and can use (get connected to) the references provided by other components via receptacle ports. The *Linking Manager* "hooks-up" components that are part of facet X receptacle connections. The user can consult the *Home Server* in order to obtain a reference to any component. The user calls the method `Home.resolve (assemblyId, componentId)`, which just gets previously created references from the CAI Repository that were created during the deployment process.

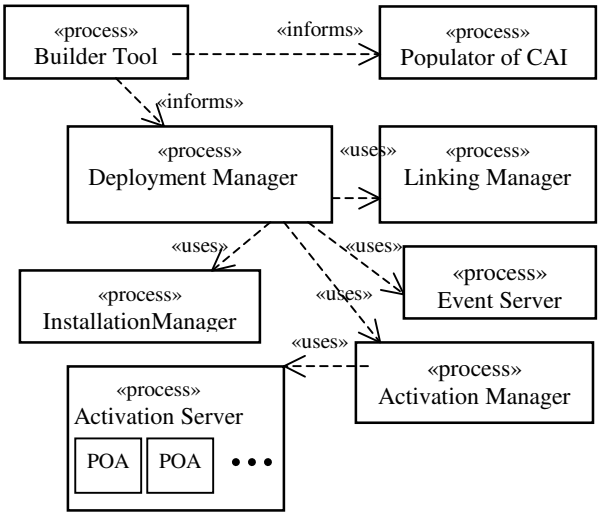

Fig. 2. Dependencies among the pieces of our deployment architecture

The POAs for components. For each component category and activation model a specific POA is created by the Activation Server. The activation model that better fits with components is On Demand: the POA creates the

reference to the component but the servant will be created only at the first time this reference is used by a client. We use a Servant Manager to create the servants. This Servant Manager acts as a container, providing for the component servant access to the CORBA Services. The Servant Retention POA policy determines the Servant Manager behavior: Servant Location (`NON_RETAIN`) or Servant Activation (`RETAIN`). Another activation model useful is the Explicit, so at deployment time both the component reference and servant are created. This is an advantage for applications that get few components and/or need that the components are ready to be used immediately. Table 1 summarizes the POAs created by the Activation Server. At deployment time the user should choose an activation model (OnDemand-`RETAIN`, OnDemand-`NON_RETAIN` or Explicit) for each component instance. This information will be stored in the CAI Repository.

	Activation Model	NON_-RETAIN	RETAIN	EXPLICIT
Component Category	Service	POAServiceNR	POAServiceR	POAService-Exp
	Session	POASessionNR	POASessionR	POASession-Exp
	Process	POAProcessNR	POAProcessR	POAProces-sExp
	Entity	POAEntityNR	POAEntityR	POAEntityExp

Table 1. POAs created by the Activation Server

CCM Deployment Architecture versus the proposed model. The pieces of our model are related with the CCM deployment architecture, shown in Fig. 1, in the following way. The `Assembly` class functionalities were distributed in our model (Fig. 2). The POA Servant Manager, in our model, acts similar to the `Container` class of CCM, by providing for component servants the access to CORBA services. The POA in our model acts similar to the Component Server of CCM in the sense it calls the Servant Manager according to the Servant Retention Policy. Unlike the `ComponentHome` in CCM, our Home Server only consults the reference to the component. Unlike the CCM `Component`, our model creates the components according the activation model. An important achievement of our proposed model is the possibility to provide different life-cycle policies according to the component categories. This allows performance enhancements, especially if the user chooses the component category according to the component demand. Another advantage is that in case of failure, the proposed model helps the recovery: when the Activation Server comes up, it recreates all the POAs, and when the Event Server comes up it reads the CAI repository in order to recreate and reconnect the necessary event channels. Also, the CAI Repository contains all the necessary information to undo a deployment; in which case, only the necessary processes at the correct machines are notified of the deployment undoing. A disadvantage is that the Proposed Model depends on the CAI repository: if new component categories with new life-cycle behaviors are added to the system, our model must be updated. In this sense, the CCM deployment architecture is more generic and, therefore, more flexible to changes.

Deployment Process. Fig. 3 represents our deployment process, which is divided in *assembly steps* (**a** to **d**) and *deployment steps* (**e** to **n**). This deployment process model describes how to gather the necessary information for the deployment, and how to set up the deployment infrastructure. We assume all the processes of our proposed model are started before the use of the Builder Tool. In the *Assembly Step*, the user activates the Builder Tool (dotted line **1**). The Builder Tool gets from the user: the chosen component implementations (**a**), the assembly name that represents the application, the identification tag for each component (optional) (**b**), and the hosts in which each component will be installed (**c**). Afterwards, by using this user information, the Builder Tool generates an Assembly Archive (AAR) (**d**). In the *Deployment Step*, the Builder Tool informs to the Populator of CAI, the file name of the AAR (Assembly Archive) just created (dotted line **2**). The Populator of CAI reads the AAR (**e**), creates an entry in the CAI and generates `AssemblyId`, `ComponentIds` and `ConnectionIds` (**f**). The Builder Tool calls the Deployment Manager and informs the `AssemblyId` (dotted line **3**). The Deployment Manager, by using an `AssemblyId` and the CAI (**g**), interacts to the: Installation Manager (**h**), which installs the libraries and their property files (if exist) in the target machines; Activation Manager (**i**), which calls the Activation Server (**j**) at each host to create references to the component instances; Event Server (**l**), which creates event channels and links event consumers and producers; and Linking Manager (**m**), which makes the connection between the components. The user can now use the `AssemblyId` and `ComponentIds` available in the CAI to invoke the required components (**n**). The step (**j**) returns a component reference to the Activation Manager, that forwards it to the Deployment Manager (**i'**), which stores it into the CAI (**i''**). The step (**l'**) returns an event channel reference to the Deployment Manager that stores it into the CAI (**l''**).

4. Implementation and Tests

The main parts of the proposed model were implemented in order to validate the model and also to evaluate implementation aspects. A use case is shown and the performance of the POAs for components is evaluated.

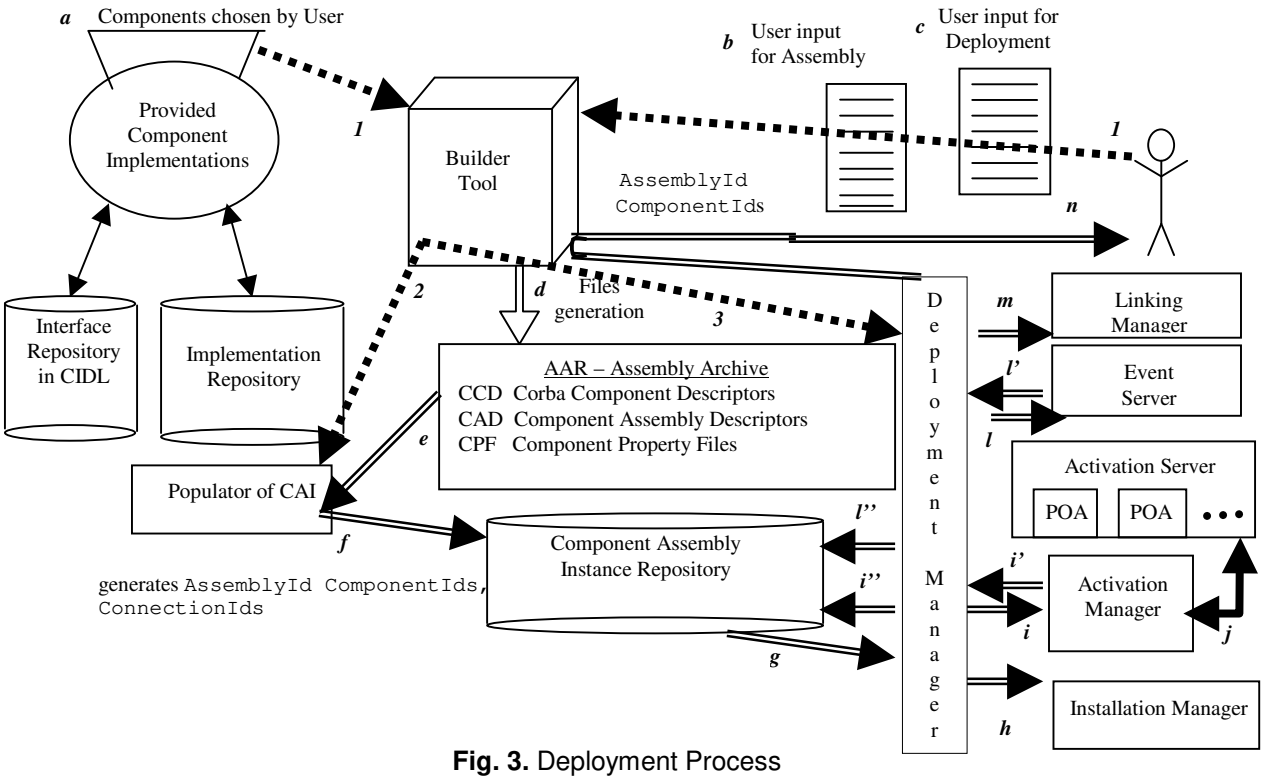

Fig. 3. Deployment Process

4.1. Development Environment

Our development environment was an Ethernet network composed of SUN workstations running Solaris 2.7 in the Institute of Computing at UNICAMP. The implementation is based on TAO [9] version 1.1, which is ORB compatible with CORBA 2.X specification, uses the IIOP1.1 protocol, provides a POA implementation, and contains some extensions to support real time.

4.2. Issues of the Implementation

All the model processes are CORBA objects, and except the Builder Tool process, the others are daemon processes, which must be started before the use of the Builder Tool. In order to guarantee uniqueness, each event channel name is the concatenation of the `AssemblyId` and the `ConnectionId`. These *ids* are unique and are generated by the Populator. It is important to note that the Deployment Manager has the role of propagating the generated component references to all the fields of the CAI repository entries.

Component Reference encoding. Our proposal is that every reference for a component contains into the field object identification, a string with the name of the DLL that implements the component and the name of the function that must be called to create a component instance, for example: `MyServant:create_MyFoo`. The POA can implement generic methods to read this coded information, load the DLL into memory and execute the function to create the component. Another approach would be storing the information about the DLL and the creation function into the CAI Repository and coding into the reference the corresponding CAI Repository entry key. Anyway, the minimum set of POA instances is created, because the same POA can be used to manage/create/destroy servants of different components. Therefore, there is one POA instance for each pair component category X activation model, because of the different policies required for each pair as illustrated in Tables 2 and 3.

Policies / POA	Request Processing	Servant Retention
ServiceExp	`USE_ACTIVE_OBJECT_MAP_ONLY`	`RETAIN`
SessionExp	`USE_ACTIVE_OBJECT_MAP_ONLY`	`RETAIN`
ProcessExp	`USE_ACTIVE_OBJECT_MAP_ONLY`	`RETAIN`
EntityExp	`USE_ACTIVE_OBJECT_MAP_ONLY`	`RETAIN`

Table 2. POA policies for the Explicit activation model

POAs for Components. The Activation Server is responsible for creating the POAs when it starts. Table 2 indicates some policies used to create the POAs with the activation model Explicit. Table 3 indicates some policies used to create the POAs with the activation model OnDemand.

Policies / POA	Request Processing	Servant Retention
ServiceNR	USE_SERVANT_MANAGER	NON_RETAIN
ServiceR	USE_SERVANT_MANAGER	RETAIN
SessionNR	USE_SERVANT_MANAGER	NON_RETAIN
SessionR	USE_SERVANT_MANAGER	RETAIN
ProcessNR	USE_SERVANT_MANAGER	NON_RETAIN
ProcessR	USE_SERVANT_MANAGER	RETAIN
EntityNR	USE_SERVANT_MANAGER	NON_RETAIN
EntityR	USE_SERVANT_MANAGER	RETAIN

Table 3. POA policies for the OnDemand activation model

4.3. Use Case

In this section we present a very simple use case on component assembly for creating a graphical user interface, in order to exercise the Deployment Process. Let's consider the following component declarations in CIDL:

```
module MyBeans {
interface Smile {
 void happy();
 void sad()
}
component SmileyBean supports Smile {
 void draw();
 provides Smile  actionToBePerformed;
 uses   Smile   actionToBePerformedAt-
Neighbour;
 attribute boolean isSmile;
 consumes  ButtonPressedEvent  action-
Event;
}
component ButtonBean {
 void draw();
 attribute string label;
 emits ButtonPressedEvent pressEvent;
}
};
```

We can build a GUI application by assembling together four component instances:

- LeftSmile: the SmileyBean that appears at the left
- RightSmile: the SmileyBean that appears at the right
- HappyButton: the ButtonBean that has the label "Happy"
- SadButton: the ButtonBean that has the label "Sad",

as illustrated in Fig.4.

Graphically our GUI application looks like the Fig. 5. Our application has the following behavior. The property files are used to set up the initial status of LeftSmile and RightSmile. Therefore, the buttons HappyButton and SadButton change the status of LeftSmile. Then, when the LeftSmile status is changed, it forwards its previous status to RightSmile. For instance, if the LeftSmile is "Sad", and the RightSmile is "Happy" (Fig. 5 a) and the HappyButton is pressed, a ButtonPressedEvent is sent to the LeftSmile. Then the LeftSmile toggles its status to "Happy" and sends its previous status ("Sad") to the RightSmile (Fig. 5 b).

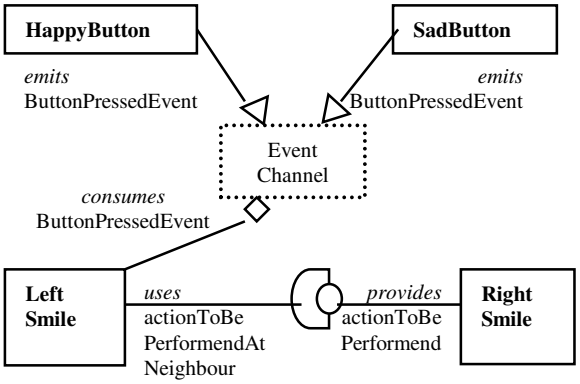

Fig. 4. Component Assembly to form the application

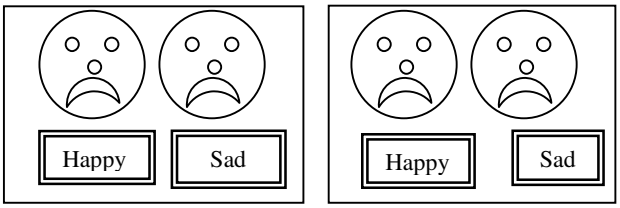

Fig. 5. GUI Behavior: a) Initial Status, and b) Status after Pressing Happy Button

Let's follow all the steps showed in the Fig. 3 for this use case:

(a) *Components chosen by the user*: The user chooses the component implementations to be used: `SmileyBean and ButtonBean` , informs which instances should be created: two instances of `SmileyBean` (LeftSmile and RightSmile) and two instances of `ButtonBean` (HappyButton and SadButton), and what is the initial status of each instance (property files). Each component implementation stores the information of the DLL location and the create function used to create a component instance.

(b) *User input for assembly*: The user hooks up the component instances: the events emitted by the HappyButton and the SadButton are sent to the LeftSmile, there is a

receptacleXfacet connection between LeftSmile and RightSmile.

(c) *User input for deployment*: The user informs: in which host each component should be deployed, what is the category and activation model of each component. For this example all component instances will be service, no_retain, and be deployed at the same host.

(d) *AAR File Generation*: The AAR file is generated with the format defined in the specification, putting together all the information previously gathered.

(e,f) *The Populator*: reads the AAR file and creates an entry describing the assembly in the CAI Repository. It is responsible for generating unique identifiers for the fields `AssemblyId`, `ComponentsIds` and `ConnectionIds`.

(g,h) *The Deployment Manager*: reads the CAI Repository entry content and, for each component instance, calls the *Instalation Manager.*

(i) For each component instance, the Deployment Manager calls the *Activation Manager* using the information contained in the fields: *Hostname*, *CompCategory*, *Activation Model/ ServantRetentionPolicy*, and *DLLName/CreateFunction*. With the returned reference **(i'),** the Deployment Manager searches in the CAI entry for all fields that should contain the generated reference and updates them **(i'')**.

(l) For the event connections, the Deployment Manager calls the *Event Server* to create the event channels and stores the returned event channel references **(l')** in the CAI entry **(l'')**.

(m) The Deployment Manager calls the *Linking Manager* to link the components which are connected together via facet X receptacle relationships.

(n) Finally, the `AssemblyId` and `ComponentIds` are returned to the user, that can use this information to access/localize component instances that belong to the deployed assembly.

4.4. Test Scenario

In order to compare the different POAs for components, we have defined a simple component with a two-way operation, called `Long doit()`, which only returns a `Long` value. The component implementation was encapsulated in a DLL, and the request processing by the POA and the creation/binding/deletion of servants for the component were measured.

A SUN workstation was allocated exclusively for this work, in order to guarantee the measure accuracy. We have developed experiments using three runs, with 10, 100 and 1000 operation invocations, respectively, on each POA type for components. In each run, we have measured the first invocation time and the average between all remaining invocations. To get more accurate measures, we have used the average between three executions of each run. Additionally, at each execution, a new client was created at the beginning of the execution and destroyed at the end of the execution, and a new server with one specific POA type (Tables 1, 2, and 3) was created before the first client invocation and destroyed after the last client invocation. Fig. 6 shows the results from executions with 100 operation invocations. The results from executions with 10 and 1000 operation invocations, and experiments made between two workstations are not shown because they are similar to those in Fig. 6.

Fig. 6 shows clearly the component POA types with `RETAIN` servant retention policy (R and Exp suffixes) have lower operation invocation time (by 6.40 ms), except the first, than those with `NON_RETAIN` (by 36.15 ms) servant retention policy. This confirms the intuition because with the `RETAIN` servant retention policy the servant is created only in the first invocation and it is used in subsequent invocations. It is also clear that in the first invocation of a run, the POAs with explicit activation model (by 53 ms) are better than with on demand activation model with `RETAIN` servant retention policy (by 79.80 ms) because in the explicit activation model, the request processing is `USE_ACTIVE_OBJECT_MAP_ONLY` and in the on demand activation model is `USE_SERVANT_MANAGER`. In other words, in the explicit activation model only the Active Object Map is consulted, while in the on demand activation model the Active Object Map is always consulted and then a servant manager is called. Other feature is that the first invocation time is very close to POA types for component with on demand activation model and `RETAIN` (by 79.80 ms) and `NON_RETAIN` (by 81.50 ms) servant retention policies. This occurs because both policies have to set up the POA in the first invocation in order to prepare to subsequent invocations. The `NON_RETAIN` policy has a bit higher overhead than the `RETAIN` policy, in the first invocation, due to the object destruction after each invocation.

Particularly, optimization techniques [6], which have been applied to the implementation of the POA in the TAO, improve, for example, the memory management (use of thread-specific storage design pattern) and the POA demultiplexing (perfect hashing and active demultiplexing). These optimization techniques improved the difference between the first and remaining invocations in POAs with `RETAIN` and `NON_RETAIN` servant retention policies. Moreover, the high difference between the first and the remaining invocations with `RETAIN` policy has occurred not only due to the inherent feature of the `RETAIN` policy, but also because in TAO, the `RETAIN` policy also benefits from applying an additional reverse-

lookup map which associates each servant with its object ID in *O*(1) average-case time [6]. In general, the first invocation is about ((79.80+81.50)/2)/6.40 = 12.6 times slower than remaining invocations in POAs with `RETAIN` servant retention policy, and it is about 81.50/36.15 = 2.25 times slower in POAs with `NON_RETAIN` servant retention policy.

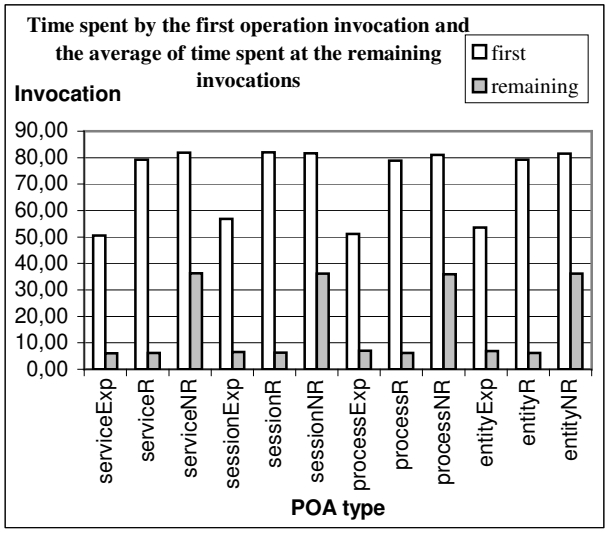

Fig. 6. Average of three executions, each one with 100 operation invocations

5. Conclusion

This paper discusses important issues for the implementation of component deployment models. In relation to the model presented in [10], this paper shows in details the POA policies for on demand and explicit activation models and all component categories. Besides, it presents a use case to illustrate better the interaction between the model participants (repositories, processes, and user information) and between the user and the model during the deployment process.

Finally, according to performed experiments, the best performance of POAs for components was obtained first by POAs with explicit activation model, which are 79.8/53 = 1.5 times faster in the first invocation and they are similar in the subsequent invocations, when compared to POAs with on demand activation model and `RETAIN` servant retention policy. However, depending on the application, it's not reasonable to create all the components in advance, mainly because of the resource consumption. So, a good approach would be using explicit activation for those components that are highly used and demand fast time response.

The worst performance was obtained with POAs with on demand activation model and `NON_RETAIN` servant retention policy, which are 79.8/53 = 1.5 times slower in the first invocation and 36.15/6.40 = 5.65 times slower in the subsequent invocations, when compared to POAs with on demand activation model and `RETAIN` servant retention policy. This activation model should be chosen for components that are used not very frequently.

As a future research work we propose to compare the "basic" CCM Deployment Model implementation with our Proposed Model implementation and evaluate performance improvements and penalties.

Acknowledgements: The authors thank FAPESP (Grant 00/05369-2) and CNPq by their financial support.

6. References

1. Hissam, S. A., Moreno, G. A., Stafford, J. A., and Wallnau, K. C. Packaging Predictable Assembly. *First International IFIP/ACM Working Conference on Component Deployment*, June, 2002, Berlin, Germany, as a Springer-Verlag LNCS publication.
2. Kebbal, D., and Bernard, G. Component Search Service and Deployment of Distributed Applications. *Proc. 3rd International Symposium on Distributed Objects and Applications (DOA'01)*, Rome, Italy, September, 2001.
3. Mikic–Rakic, M., and Medvidovic, N. Architecture–Level Support for Software Component Deployment in Resource Constrained Environments. *First International IFIP/ACM Working Conference on Component Deployment*, June, 2002, Berlin, Germany, as a Springer-Verlag LNCS publication.
4. Object Management Group. CORBA Components Volume I. Document orbos/99-07-01. 1999.
5. Object Management Group. CORBA 2.6 - chapter 11 - Portable Object Adapter. Document formal/01-12-49. 2001.
6. Pyarali, I., O'Ryan, C., Schmidt, D. C., Wang, N., Kachroo, V., and Gokhale, A. Applying Optimization Patterns to Design Real-time ORBs, Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems, May, 1999, San Diego, CA.
7. Rudkin, S., and Smith, A. A Scheme for Component Based Service Deployment. *Proceedings of Trends in Distributed Systems: Towards a Universal Service Market*, Sept. 2000, Springer-Verlag, pp 68-80.
8. Wang, N., Parameswaran, K., Kircher, M., and Schmidt, D. Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation. *Proceedings of the 24th Annual International Computer Software and Applications Conference* (COMPSAC 2000), Taipei, Taiwan, October, 2000.
9. TAO (The ACE ORB). TAO is a CORBA ORB available to download at the site: http://www.cs.wustl.edu/~schmidt/ACE_wrappers/TAO.tar.gz
10. Barros, M.C.B.; Madeira, E.R.M. and Sotoma, I. – "A Deployment Model for CORBA Components"- International Symposium on Distributed Objects & Applications – DOA '02, Irvine, USA, October 2002, Technical Report - University of California, Irvine.