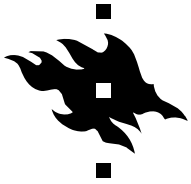
Department of Computer Science
Series of Publications C
Report C-2001-nn

Evaluation and exploitation of OMG CORBA and CORBA Component Model

Lea Kutvonen, Juha Haataja, Egil Silfver, Markku Vähäaho



University of Helsinki
Finland

Department of Computer Science
Series of Publications C
Report C-2001-nn

Evaluation and exploitation of OMG CORBA and CORBA Component Model

Lea Kutvonen, Juha Haataja, Egil Silfver, Markku Vähäaho

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: Lea.Kutvonen@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 708 51

Telefax: +358 9 708 44441

Computing Reviews (1998) Classification: C.2.4., D.2.0
Helsinki 2001

Evaluation and exploitation of OMG CORBA technologies

Lea Kutvonen, Juha Haataja, Egil Silfver, Markku Vähäaho

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
Lea.Kutvonen@cs.Helsinki.FI

Technical report, Series of Publications C, Report C-2001-nn
Helsinki, March 6th 2001, vi + 42 pages

Abstract

The PILARCOS project developes middleware solutions for automatic management of interorganisational applications. The project works at three fronts

- architecture development for a federated environment where applications are constructed based on community descriptions populated by traded components;
- prototyping in CORBA some essential middleware services required for global negotiation of federations and instantiating communities; and
- designing an ODP viewpoint based software engineering process and tool that interfaces with the middleware repositories of the runtime environment.

This document introduces OMG CORBA and CCM technologies selected to support Pilarcos architecture, describe how those technologies will be used, and shows the needs for extensions on these technologies.

Essential functionalities of Pilarcos architecture to be supported are those of denoting and storing architecture designs for runtime population, matching components against the architecture designs, storing and instantiating component assemblies, providing support for complex bindings, and promoting the usage of organisation-wide policies.

Computing Reviews (1998) Categories and Subject Descriptors:
C.2.4 Computer-communication networks: Distributed Systems
D.2.0 Software engineering: General

General Terms:
Design, Experimentation, Standardisation

Additional Key Words and Phrases:
Software system architectures, federated systems, software engineering, component models, CORBA

Acknowledgements

This report is based on work performed in Pilarcos project at Department of Computer Science at University of Helsinki. The Pilarcos project is funded by the National Technology Agency TEKES in Finland, together with Nokia, SysOpen and Tellabs. Plenty of the ideas behind this research are encouraged by discussions within and around the ODP standardisation work.

Helsinki, March 2001
Lea Kutvonen

Contents

Chapter 1

Introduction

The Pilarcos infrastructure aims for an advanced middleware that is able to support interorganisational applications [2]. Interorganisational cooperation raises problems on managing interworking between services that are supported in heterogeneous environments and evolved and managed independently from each other. As the systems involved are autonomous, the ontologies for management need to be abstract enough and be mappable to locally selected technologies. The management ontologies need to sustain changes in cooperation schemata and in the members taking part in cooperation.

The Pilarcos solution for these problems is based on splitting a traditional application architecture into two: an architecture description and service providing components that can populate the architecture. The architecture description is a contract for providing services and describes how component services need to interact in order to provide the supported service. Meta-information services like traders are needed to carry offers for component services for populating the architectures. Negotiations on cooperation and assignment of performers to tasks is based on predicates over the components involved, in ODP terms, on types.

The Pilarcos project promotes technology and platform independent modelling and programming of applications. Automatic, transparent, federation establishment is one of the goals in Pilarcos architecture [2]. The automated federation management requires conceptual advancement in most platforms, i.e., introduction of generic negotiation protocols, heavy usage of late binding mechanisms with conformance test between communication partner interfaces, and automatic interceptor configurations.

From organisational IT systems and their management needs stems a requirement for governing the services provided by the organisation's IT system with enterprise wide policies. This need can be accommodated by the Pilarcos architecture by the use of explicit contracts in all communication between autonomous assemblies and by the use of policy aware component model.

In order to achieve the above goals, the Pilarcos middleware supports functionalities for denoting and storing architecture designs for runtime population, matching components against the architecture designs, storing and instantiating component assemblies, providing support for complex bindings and promoting the usage of organisation-wide policies.

The OMG CORBA architecture with extensions is considered to be one of the

most promising infrastructures to carry out large system computing and interoperability between organisations. Thus the Pilarcos project uses the OMG Technologies as an exemplary case to show the feasibility of the new architecture. At the same time, contributions are fed for the OMG technologies themselves.

The purpose of this report is to evaluate the suitability and adequacy of the OMG technologies for usage in Pilarcos architecture, concentrating on the runtime environment and support found in CORBA [8] and the new CCM (CORBA Component Model) architecture [6]. An accompanying document concentrates on software design aspects around UML and MOF (meta-object facility) [9].

The Pilarcos architecture places several direct requirements on the supporting platform. Firstly the underlying platform must provide support for the usage of abstract architectural descriptions that can be mapped to technology dependent services. In addition the concepts of type and template must be separated in a way conformant to the ODP reference model by introducing additional services (including a type repository). The client and server views of a shared interface must also be separated. This raises a need for symmetrical support facilities on the object bus for both on client and server sides, thus modifying the interaction model to a true peer-to-peer model. In order to extend modelling capabilities and to support dynamic system configurations software components need to be able to support several independent interfaces. The components should also be able to have alternating epochs with differing numbers of interfaces offered. The components need to expose the requirements placed for their environment and to other components in addition to declaring what services they can provide themselves. Standard life cycle service support for managing the creation and destruction of application components is also needed.

In order to promote software reusability via supporting repositories, there must exist standard ways to compose the existing components into new configurations. To support dynamic federations and interorganisational communication the infrastructure services should make sure that the applications obey their organisational policies automatically. The communication channels between the components need to be manageable by QoS negotiations and business policies and adaptable to changes in resource availability in the network.

This report evaluates the basic CORBA architecture, and the component model extensions (CCM) against these requirements. Chapter 2 first introduces the facilities available, Chapter 3 then discusses how the above mentioned services can be brought to the CORBA CCM platform. Chapter 4 provides conclusions.

In addition, interesting parallelism of needs and intended usage can be found on two areas; modelling concepts and management interfaces. The goals of Pilarcos architecture driven software is conformant with the current MDA (Model Driven Architecture) theme arising in OMG [12]. The technical requirements for middleware services and the programming environment includes the rise of abstraction level in the programmer interface; explicit communication establishment should be replaced by implicit services, and personalisation by profile information taken into account. A draft on CORBA management services request for proposals [7] exists, but the work has never been started properly. Therefore, there are no recommendations for the CORBA management interfaces. A definite need for these interfaces exists for a number of reasons. First, telecommunication applications heavily trust on domain management services, but can-

not move to CORBA platforms as far no standard interfaces are guaranteed. Second, the platform management functions form a necessary basis for application management services and management of interoperability and federations both across platform technology domains and organisational domains. Third, the application structures are affected by the management facilities available.

Chapter 2

Overview of CORBA and CORBA Component Model (CCM)

The Pilarcos architecture [2] itself is technology independent, but the concrete applications must be implemented and run in some technology dependent environment. The OMG has introduced an architectural framework, Object Management Architecture (OMA) [11] for building distributed object-oriented systems. The CORBA architecture [8] specifies the object request broker (ORB) forming the basis of OMA. The CORBA ORB is supported by accompanying CORBA object services [4] and CORBA object facilities [3]. The new CORBA Component Model (CCM) specification [6] extends the CORBA architecture with a standard component model. The role of CORBA in the Pilarcos project is to be the exemplary platform which demonstrates how the different technologies can be exploited by Pilarcos runtime architecture and what requirements are be placed for them. This chapter provides an overview of OMA, CORBA and CCM.

2.1 Overview of CORBA

OMA defines a reference model that divides the problem space of distributed systems into four distinct sub-spaces: object request broker (ORB), object services, object facilities, and application objects. The ORB is the heart of the OMA and is responsible for the actual communication between objects. The Common Object Request Broker Architecture (CORBA) [8] is a specification of the ORB. CORBA object services [4] consist of common infrastructure services (e.g. naming, trading, persistence, transactions, events, life cycle, security, ...). Their goal is to provide support for programmers in the form of ready-made components and to improve interworking by standardisation of the operating environment. The object services are external to the ORB and provide a public interface unlike the ORB internal services which are ORB implementation specific. The CORBA facilities [3] are build on top of CORBA services and divided into horizontal and vertical facilities. Horizontal facilities consist of user interface facilities, information management facilities (e.g. Meta Object Facility [5]), systems management facilities, and task management facilities and are common to all application domains. Vertical facilities are application domain specific and they include for example medical,

financial, e-commerce, and telecom facilities.

2.1.1 The concept of object

The OMG object model defines a core object model and a set of rules which allow the core to be extended. The basic concepts of the core object model are summarised as follows [1, 8]: an object has a distinct identity, an internal state, and an observable behaviour. Object's identity cannot be changed and it is independent of object's state and behaviour.

An operation is an action that can be requested from an object. Each operation has a signature that specifies its name, parameters, and results. The operations are the only way to affect the object's internal state. An interface is the collection of the operations supported by the object.

An object type is a predicate defined over the set of all objects. A subtype is a refinement of an object type containing all features of the base type in addition with some extending ones. Subtype can always be used in an environment expecting the base type. Subtype relationship is achieved by inheriting another type which then becomes a base type of the subtype.

The CORBA object model defines the concrete object model that underlies the CORBA architecture. The model is derived from the core object model and it makes it more concrete. Features added by the CORBA object model include [1, 8]: input, output, input/output parameters, exceptions, and basic data types in addition with the possibility to combine them into more complex structures.

2.1.2 The concept of object reference

A CORBA object is represented to a client with an object reference. The object reference identifies a specific CORBA object and encapsulates binding related information between the client and the target object. Because ORBs have their own internal representation of object references, the ORB interoperability architecture defines the so called interoperable object reference (IOR) to enable different CORBA ORBs to interpret object references in a standardised way. An IOR contains at least an interface type identifier, type of the ORB, notion of the security mechanisms used at that interface, and a sequence of object-specific protocol profiles. There are two basic types of object references: transient and persistent. A transient object reference contains the address of the CORBA server at the time of object reference creation plus a unique server identifier and a unique object identifier to identify the referenced object. The transient nature of an object reference means that its contained information is valid for just as long as the server process is alive. When the server crashes and restarts it creates a new unique server identifier for itself and no longer accepts requests for the objects in the old server. Persistent object references are valid even after the server crashes or is shut down. They are valid until the corresponding conceptual CORBA object is intentionally deleted. Persistent object references do not contain the address of the server hosting the referenced object, but they contain the address of the implementation repository instead. When a persistent object reference is used to make a request, the request is first delegated to the address of the implementation repository

which sends back a so called 'location forward' message (in case of GIOP messaging) containing the actual address of the server hosting the target object. Client ORB then automatically rebinds to the server address. CORBA messaging specification [8] also adds functionality allowing clients to control whether to rebind on 'location forward' or not. To support for example failure tolerance the object reference may contain multiple addresses referencing to multiple replicas of the server object.

2.1.3 CORBA binding mechanisms

In CORBA, the binding actions are implicit and performed by ORB services. The essential components of the ORB related to binding are interface repository, implementation repository, and inter-ORB communication facilities. To make a request, the client can use the dynamic invocation interface (DII) or an interface specific stub generated from an IDL description. The dynamic invocation interface is independent of the target object's interface. The requests arriving to the object implementation are received as an up-call either through the IDL skeleton or through a dynamic skeleton (DSI).

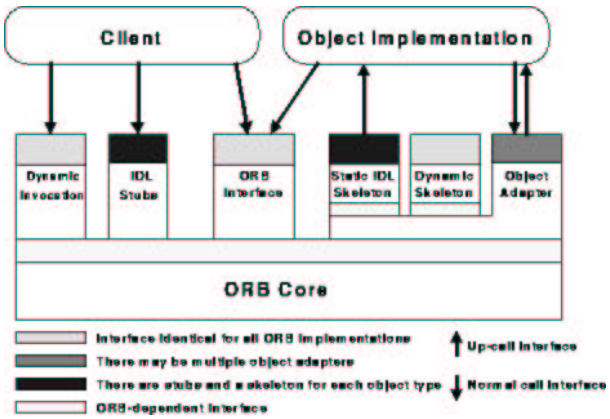


Figure 2.1: Overview of the basic ORB components [8].

An object adapter enables the communication between the object implementation and the ORB. Services provided by the object adapter include generation and interpretation of object references, method invocation, security of interactions, object activation and deactivation, mapping object references to implementations, and registration of implementations. The default object adapter is the Portable Object Adapter (POA) [8]. Figure 2.1 shows the interfaces used by the application developers when performing requests to CORBA objects. Visualised are the standard dynamic invocation and skeleton interfaces, object interface specific stubs and skeletons, object adapter interface, and the standard ORB interface.

The repositories used during binding are the interface repository and the implementation repository. The interface repository stores interfaces as objects available for

queries at runtime. The ORB may use the interface repository information, for example, to perform runtime type checking of interfaces. The implementation repository contains information that allows the ORB to locate and activate implementation of objects. Information like implementation locations and control policies related to the activation and execution of object implementations can be stored in the implementation repository.

The binding can also be affected through several lately developed meta-programming mechanisms which include interceptors, smart proxies, and pluggable protocols [15]. Interceptors are objects invoked by the ORB in the operation invocation path. With interceptors it is possible to change the behaviour of the invocation without changing client or server application software. Smart proxies are custom stub implementations that override the default stubs created by an IDL compiler. They allow the customisation of the client behaviour without changing the application. Pluggable protocols are objects which implement the transportation mechanisms used in the ORB. A pluggable protocols framework allows the application developer to plug-in custom communication mechanisms without changing the ORB internal implementation. The portable interceptors specification makes the interceptors a standard part of the CORBA specification but the smart proxies and pluggable protocols are still proprietary solutions offered by some ORB implementations but not by others.

2.1.4 ORB interoperability

The ORB interoperability architecture [8] provides a conceptual framework and a set of techniques for implementing interoperability between independently developed ORBs and other object-oriented platforms. The interoperability architecture divides distributed systems into domains where objects share common characteristics or follow a set of common rules. Some typical domains include referencing domain, representation domain, security domain, and type domain. The referencing domain is the scope of an object reference. The representation domain is the scope of a specific representation format. The security domain is the scope of specific security technologies, security policies, and rules. Type domain is the scope of a specific type system. A domain may extend over multiple ORBs or in some cases an ORB may be split into multiple domains.

In order for a request to cross a domain boundary it must traverse a bridge which ensures that the original contents and semantics of the request are mapped correctly to the those of the target domain. The bridging approaches are divided into immediate and mediated bridging. In case of immediate bridging the needed conversions are done at the domain borders directly between the domain internal formats. In case of mediated bridging the conversions are done via a common format. The immediate bridging provides optimal performance but mediated bridging is often more practical because there is no need to support mappings between each internal format. This reduces the amount of mappings from exponential to linear scale.

Two alternative techniques exists for implementing bridges between ORBs. The first approach is for the ORB to use directly the internal API of another ORB. This approach is called in-line bridging. The approach is efficient but it tightly couples the ORBs together. The other implementation approach is called request-level bridging. In

request-level bridging the ORB in domain A uses the dynamic skeleton interface (DSI) to pass the client's call to the local half-bridge. The local half-bridge in domain A then communicates the request to the remote half-bridge in domain B. The half-bridge in domain B constructs a corresponding operation call to the server using dynamic invocation interface (DII). The half-bridges may use e.g. proxy objects to keep up the binding relationships between the interacting objects. The request-level bridging approach is visualised in Figure 2.2

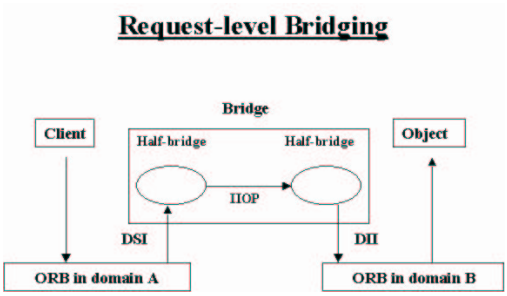


Figure 2.2: Request-level bridging with IIOP.

The most commonly used ORB messaging protocol is the general inter-ORB protocol (GIOP) which defines the transfer syntax and a set of message formats for sending operation requests and replies. GIOP can be implemented on top of most transport protocols. The obligatory, and most widely used GIOP-implementation is Internet inter-ORB Protocol (IIOP). IIOP defines how GIOP is implemented using TCP as the underlying transport mechanism.

2.1.5 Bridging to non-CORBA systems

Bridging to non-CORBA systems means relaxing a few requirements from the basic inter-ORB bridging. These requirements include: (1) No strict compliance to CORBA object model, and (2) no strict compliance to IDL interface definitions and IDL concepts. Also the support for communication protocols and strategies used may variate considerably. A consequence of this relaxation is a more difficult environment to achieve complete interoperability. The non-CORBA bridging relationship is often referred to as an interworking relationship.

Basic concepts of the interworking architecture are the interworking object model and the view object model. The idea behind the interworking model is that all interworking objects should have a well defined interface which can be accessed via a well defined object reference. The conversions between differing object models are done with so called view objects which provide the client a view of the target object in a proper referencing format using the type system understandable by the client. The view objects are conceptually components of the interworking bridges and are implemented as real objects in their own technology domain. The physical location of a view object is however situation specific. The interworking object model and the view object model

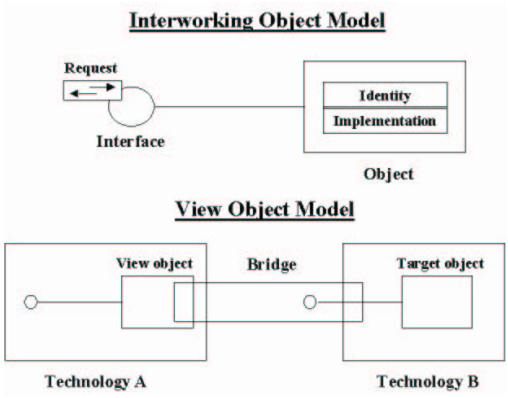


Figure 2.3: Interworking object model and view object model.

are visualised in Figure 2.3. A view object can be an 'adapter' for some specific interface or a group of interfaces. On the other hand a view could be implemented as a general adapter which can be used to adapt a large set of interfaces using some dynamic mapping mechanism.

2.2 The goals and structure of the CORBA Component Model

The original CORBA object model was lacking several features considered important in modern software development. These features include [14]:

- support for standard programming model for CORBA servers,
- standard life cycle management of CORBA objects,
- support for the provisioning of multiple unrelated interfaces from a single access point,
- means to describe the dependencies between an object and its environment as well as dependencies between distinct objects (i.e. object connections), and
- lack of standardised deployment model for the objects.

OMG tries to provide answers to these kinds of problems with the CORBA Component Model (CCM) standardisation effort. CCM specification introduces new concepts including CORBA component, component home, and component container. The CCM specification also introduces a framework for component implementation and a component deployment architecture. The CORBA component is an extension to the notion of CORBA object and the component home is a component specific factory enabling remote creation and life cycle management. The component container acts as the components runtime environment providing a standard deployment place.

2.2.1 The concepts of component and component home

The concept of home

The component home is a specialised component factory and provides management operations, like 'create', 'find', and 'destroy' which standardise the component life cycle handling. Each CCM-component has an associated home which is specific to the component type. To be exact, one home type is dedicated to one component type, but a component type may be supported by several different home types. At the instance level a component instance is always managed by one home instance.

Homes are divided into two main categories, keyless and keyful homes. Keyless homes provide only the so called factory operations for creation and destruction of components. Keyful homes also support finder operations to find a persistent component matching a previously specified unique key. The keys used are considered unique within one homes name space. Component homes can also be provided with so called 'Configurator' objects which encapsulate a specific component attribute configuration that can be used to initialise the component's internal state. The configuration mechanism makes it possible to divide the component life cycle into configurational and operational phases and provides a tool for creating generic components which can be specialised at instantiation time.

The concept and structure of component

Components extend the CORBA object model by defining a new set of interfaces called ports. Ports can be divided into four sub-categories called: facets, receptacles, event sources and sinks, and attributes. Facets are mutually unrelated interfaces provided by the component and their implementation is hidden from the client. Receptacles are hooks which can be connected to previously defined interfaces and provide a generic way for the component to describe its dependencies from other components. Event sources and sinks are specialised interfaces which provide a simplified access to the CORBA notification service and can be used to connect components together by a loosely-coupled mechanism. Event sources can publish or emit events to other components or to event channels. Event sinks can be used to specify the components interest to events produced by some other entity. Attributes are typically used to configure the component's internal state. Attribute values are viewed and changed via specific 'get' and 'set' operations which can be defined to throw exceptions if used after the configuration phase of the component. Configuration phase is finished by calling 'configuration complete' operation on the component after it is instantiated and configured. Components are represented to clients as component references.

IDL extension aspects

In order to provide support for using the new features, the component specification extends the OMG Interface Definition Language (IDL) with standard notations to describe component and home features.

These extensions include:

- 'component' keyword for describing components,

- 'home' keyword for describing homes,
- 'supports' keyword for describing component's supported interfaces,
- 'provides' keyword for describing component's provided interfaces,
- 'uses' keyword for describing the interfaces component needs from other components,
- 'emits' keyword for describing component's emitted events,
- 'publishes' keyword for describing component's published events,
- 'consumes' keyword for describing component's consumed events.

Components and homes can inherit interfaces from other components and homes, but only single inheritance can be used. The component and home IDL definitions can be mapped to conventional IDL definitions which makes it possible to use an existing IDL compiler by implementing a pre-compiler above it.

2.2.2 The container abstraction

A container is an abstract function which provides the necessary runtime environment to deploy and execute the components. A container encapsulates one or several components and their related homes, provides an API to use some CORBA services (security, transactions, notification, persistence), and serves as a standard deployment place for components. Handling of some of the services (namely transactionality and persistence) can be totally given to container responsibility. For each container a specialised object adapter is created which manages the activation of servants and the creation of necessary object references for the components and homes deployed in it.

Container APIs

A CORBA container is essentially a server side concept and it is characterised by a related container API type and a CORBA usage model. The container API type defines interactions between the component and the container and is divided into two main categories, a session container API for components using transient object references and an entity container API for components using persistent object references. Entity API is designed to be used with components that have a persistent state which can be stored and loaded when needed. Session API is designed to be used with components that have no state at all or have a state which exists only for the lifetime of the process which created it.

The container API is separated to container internal API and the component callback API. The container internal API defines the operations which can be used by the component. The container internal API includes a 'HomeRegistration'-interface to provide a standard way to make component homes available to clients. If homes are registered this way, the client can find the component home with the corresponding 'HomeFinder'-interface. The container callback API defines the operations which the component must implement in order to be successfully deployed into a container. For

example the entity API requires that operations like 'store' and 'load' for persistent state must be supported by the component. Their implementation can be either implemented manually or generated into the component executor. The CCM container architecture and its key concepts are visualised in Figure 2.4.

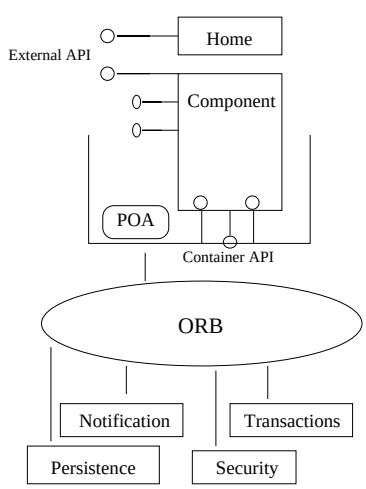


Figure 2.4: The CCM container architecture.

CORBA Usage Model

The CORBA usage model describes the container's interaction model (1) with the POA, (2) with the ORB, and (3) with the supported CORBA services. The CORBA usage model has three main categories which are stateless usage model, conversational usage model, and durable usage model. Stateless usage model is characterised with the usage of transient object references and servants which can support any object identifier and can only use a method lifetime policy. Conversational usage model uses transient object references but expects each POA servant to be dedicated to a specific object identifier. This leads to a possibility of using all possible servant lifetime policies called method, transaction, component, and container. Durable usage model uses persistent object references and a POA servant dedicated to a specific object identifier. All four servant lifetime policies are possible in the durable model.

Component categories

There exists four combinations of the container API type, the CORBA usage model, and the home type which define four component categories (service component, session component, process component, and entity component) suitable for deployment in containers supporting them. Service components use a combination of the session API type and a stateless usage model. They have no state, only behaviour and can

be used, for example, as wrappers for legacy procedural applications. Because service components use transient object references their home must be keyless. Session components use a combination of the session API type and a conversational usage model. They maintain a transient state which exists for the lifetime of one component instance, can be used for example as iterator objects, and are often associated with transactions. These components again use transient object references so their home must be keyless. Process components use a combination of the entity API type and a durable usage model. They have a persistent state, which however is not directly visible to the client. Process components can be used to model business processes and are often associated with transactions. These components use persistent object references, but their home is keyless because the persistence is not externally visible. Like process components the entity components also use a combination of the entity API type and a durable usage model but they expose their persistence to the client by supporting indexing with a primary key. Entity component homes are keyful meaning that they associate a primary key with the component and enable finding the component later. Entity components can be used for example to encapsulate a row in a database.

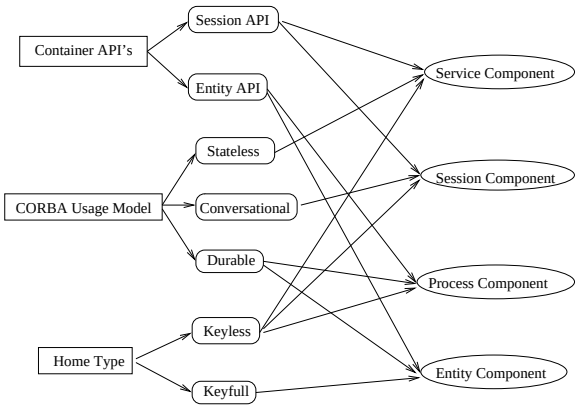


Figure 2.5: The defined component categories and their relationships to container API types, CORBA usage models, and home types.

The component categories and their relationships to container API types, CORBA usage models and home types are visualised in Figure 2.5. In addition to the described categories others can be provided by supporting additional APIs. A container can, for example, support an Enterprise Java Beans [13] API, like EJB session API or EJB entity API. In this case, an enterprise bean can be deployed into a CCM Container and communicate with the CORBA environment using an EJB view.

Implementing the container

The container abstraction has many possible implementations which are all applicable if their behaviour is consistent with the container specification. The standard proposes to use the capabilities of the Portable Object Adapter, and especially the POA servant

managers. A more advanced approach is to combine the servant manager usage with the portable interceptors and other meta-programming mechanisms [15, 16].

The specification [6] provides an example architecture which uses a specific POA servant manager design called 'ServantLocator'-architecture. Servant locators enable specialised servant manager interfaces to implement the container functionality by being on the invocation path for all requests directed to the component. In the basic approach a child POA is created to support the container, its policies are configured according to container category, and a proper servant locator is registered. A simplified run-time view of the 'ServantLocator'-based container implementation is visualised in Figure 2.6. The interactions reviewed shortly are (1) client makes an operation invocation, (2) the object adapter performs a pre-invocation on the servant locator, (2.1) the servant locator instantiates or locates a proper executable servant, interacts with necessary object services, interacts with the component executor using the container callback API, and returns a reference of the executable servant to the object adapter, (3) the object adapter performs the actual operation call. After this the object adapter performs a post-invocation which has similar functionality than the pre-invocation.

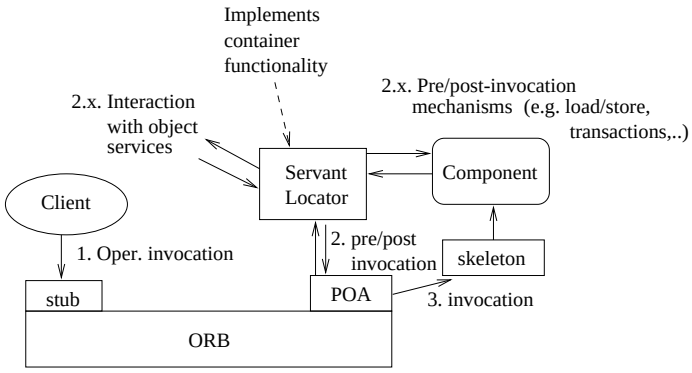


Figure 2.6: Simplified runtime view of the 'ServantLocator'-based container implementation.

2.2.3 Component development support

The traditional CORBA development support has relied mostly on IDL interface definitions. IDL definitions describe the external (client) view of a CORBA-object. The implementor of the server side has had no standardised implementation support above the programming language specific features. The component model extends the IDL with component features but it also provides new means to automate the server development with a Component Implementation Framework (CIF). The CIF is designed to be compatible with the existing POA framework so it can rather straightforwardly be implemented using it [6].

The CIF is a programming model for constructing component implementations. CIF

enables the generation of component executors which provide the implementation skeletons for the component developers. The executors embed component and component home related support functionality (navigation, identity, activation, state, life cycle management) and provide the framework for plugging in the actual business logic. The component executors work in cooperation with the component runtime environment (i.e. use the container's internal and callback API) to automate the management of the component state. The focal point of the Component Implementation Framework is the declarative Component Implementation Definition Language (CIDL). CIDL can be used to describe the structure and state of component implementations. CIDL can import constructs from IDL-descriptions and is a superset of the Persistent State Definition Language (PSDL). PSDL itself was designed for defining the persistent states for CORBA objects.

CIDL uses the concept of composition to describe the component implementations and their properties. Composition is used to declaratively bind together the component, its home, its life cycle category, and to associate between abstract storage types and internal storage formats. A composition definition may have several forms. The structure of a simple composition looks like the following

```
composition <category><composition_name>{
   home executor <home_executor_name>{
     implements <home_type>;
     manages <executor_name>;
   };
};
```

A more complicated composition with managed storage declarations for persistence handling has the the following kind of structure:

```
composition <category><composition_name>{
   uses catalog{
      <catalog_type><catalog_label>;
   };
   home executor <home_executor_name>{
     implements <home_type>;
     bindsTo <catalog_label.abstract_storage_home>;
     manages <executor_name>;
   };
};
```

The CIDL specifications can be used to generate implementation skeletons for components and their homes. The code generated by the CIDL compiler are the component executors which contain the auto-generated implementations and hook methods that allow component developers to add the custom business logic [14]. In addition the CIDL definitions can be used as a basis for the generation of the component descriptors. Component descriptors are embedded in the software packages used in packaging and deploying component applications.

A possible CIF based development scenario is presented in Figure 2.7. It represents a development scenario utilising the IDL, the CIDL, and the interface repository to

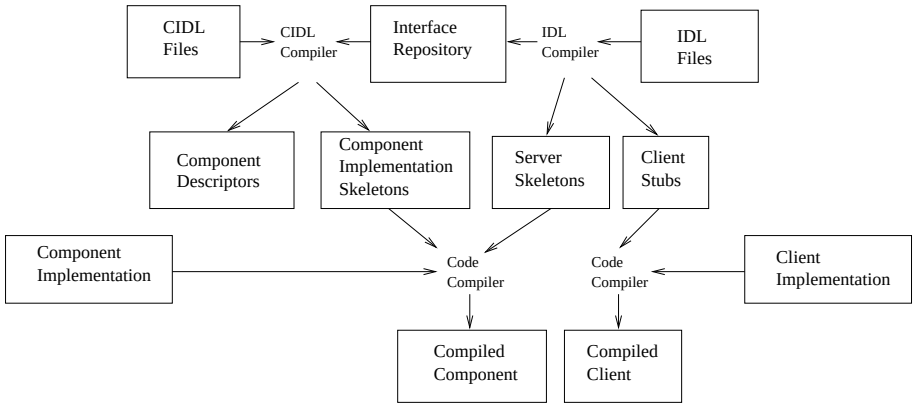


Figure 2.7: CIF based development.

generate the needed skeletons and executors for a component and a stand-alone client. In case the client would also be a CCM component the component descriptors and component implementation skeletons would be generated for the client too.

2.2.4 Packaging and assembling the components

Component implementations can be packaged in component packages which are typically zipped archive files containing a component package descriptor and a set of files. The files contain the actual implementations of component and its home.

Component package descriptor

Component package descriptor is an XML document containing information of the package in general and its internal structure. Component package descriptor contains elements like title and type of the package, author of the package, licensing contract information or a link to it, dependencies from other software packages or the environment, and links to the actual component and home implementations (e.g. '.dll' or '.class' files). The component implementations can be located in the same package with the component package descriptor or they can be located in separate repositories. In the latter case only links to them are embedded in the descriptor. Component package descriptors can be generated with help from a proper tool.

Component descriptor

The component implementation details are described with a separate XML descriptor called a component descriptor. A component package can contain one component descriptor applicable to all implementations or a separate component descriptor for each implementation. Component descriptor is generated mostly by a CIDL-compiler

and it can include specific information of the implementation. This information includes version of the CORBA expected, repository identifiers for component and home interfaces, transaction policies, security policies, threading policies, links to property files describing the initial component and home configurations, interfaces related to the component, all external features of the component and home (i.e. ports), persistent state descriptions, and components life cycle category.

Component assembly packages and assembly descriptors

Component packages are the simplest deployment units and are deployed to component servers. Components can be assembled together to form component assemblies which makes it possible to construct numerous different applications from existing software. Component assembly packages are analogous to the component packages in that they also contain a set of files and a descriptor which describes the contents and general information of the assembly. Example of the files included are the component packages. Assembly package can be distributed as a physical zipped archive file containing all the related component packages or it may contain only the assembly descriptor holding links to its components located in some other location. The assembly descriptor is the most important concept considering the advanced deployment scenarios in that it holds the necessary information needed when deploying, instantiating, and connecting sophisticated component compositions. The assembly descriptor XML DTD is quite massive but its main elements are links to the component packages forming the assembly, logical partitioning information (host and process collocation) for related components and their homes, and descriptions of the connections to be established between the components at instantiation time. Example hierarchy of assembly descriptors, component package descriptors, and component descriptors can be seen in Figure 2.8

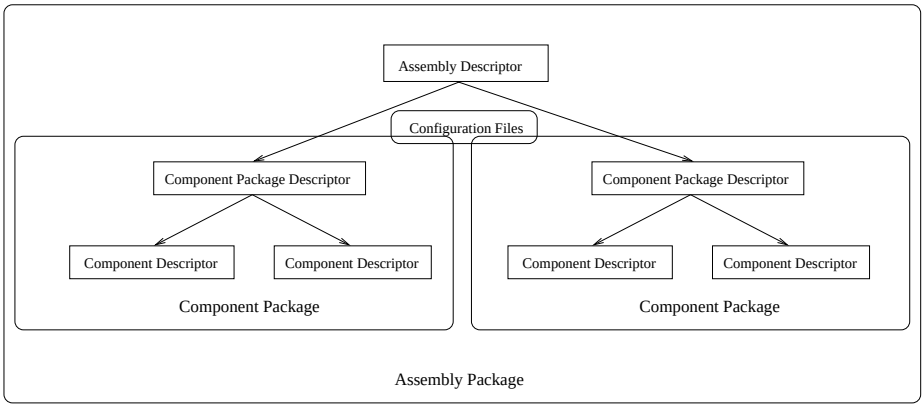


Figure 2.8: Example hierarchy of the packaging descriptors defined in CCM model.

2.2.5 Deploying the component applications

Deployment process maps the logical software configuration to a physical environment. Component assembly life cycle includes four distinct phases (1) installing the software into correct hosts, (2) activating and connecting the components in each host, (3) deactivating the components and tearing down the connections in each host, and (4) uninstalling the software from each host.

The CCM deployment architecture defines three mandatory interfaces to support the deployment process. The interfaces are 'ComponentInstallation', 'AssemblyFactory', and 'Assembly'. The 'ComponentInstallation' interface is used to install and uninstall a single component. The 'AssemblyFactory' is used to create, find and destroy assembly objects. An 'Assembly' object represents one assembly and provides operations to build (instantiate and connect) and tear-down the physical configuration. The 'ComponentInstallation' objects are always active in the hosts where the components are to be installed. The 'AssemblyFactories' are always active in the hosts where assembly objects are to be created. The deployment tool interacts with 'ComponentInstallation' objects and 'AssemblyFactories' in order to install the components embedded in the assembly descriptor to each specified host and to create an 'Assembly'-object to control the instantiation and configuration of individual components and to establish the connections between them. The 'Assembly' object can also be used to tear the instantiated assemblies down later. The concrete installation, instantiation, and connection process is vendor specific although the component model specification provides an example architecture as a guideline for achieving a working deployment system.

Chapter 3

CORBA and CCM exploitation schemes

In order to support the requirements placed by the Pilarcos architecture [2] for the future middleware, the capabilities of the CORBA must be extended in different ways. From the runtime point-of-view the most important extension is the addition of Pilarcos binding services. In addition the exploitation possibilities of the CCM deployment architecture need a thorough evaluation. This chapter lists the requirements placed for the underlying middleware, evaluates how the Pilarcos extensions could be fitted into the CORBA platform, discusses the exploitation of the CCM packaging and deployment architecture, and provides a complete scenario for accessing federated services which shows how the different extensions relate to each other and are used in practice.

3.1 The requirements placed by the Pilarcos architecture

In Pilarcos the applications are composed of 'computational' components which are mapped to technology dependent concepts. In the case of CCM this concept is selected to be the component assembly. Of course a single CORBA component can also represent a computational component but this can be seen as a special case of a component assembly - an assembly consisting of only a single component. In Pilarcos architectural designs a computational component is represented by its external interfaces, and so from the Pilarcos point of view only the interfaces that are used to bind the assembly to the computational architecture are relevant. This implies that the internal bindings of the assemblies are considered to be hidden from other Pilarcos components. Basic requirement for the technologies to be integrated into the architectural designs, is a clear separation between the interface and implementation of a software component. Also a formal method for describing the interfaces should be available. CCM and CORBA in general perform well in this category with the IDL interface definition language.

There should also be support for (semi)automatic deployment of the components. This requirement is not as critical as the previous one, because the installation of the

components could be done manually or with some proprietary mechanism in many cases. Although it is not a mandatory requirement, the CCM standard deployment architecture clearly provides additional value for the developers and is used by the Pilarcos SE-tools where appropriate. Things required from the the deployment architecture is a container model for the components to be deployed into, and standard descriptors for describing the deployment configurations and the important non-functional aspects of the applications (security, transactionality, ...).

In addition the Pilarcos architecture aims at supporting complex bindings in a federated, inter-organisational, environment. This places additional requirements for the underlying communication mechanisms. The complex bindings taking place in these environments differ from simple bindings in several key factors. In order for a system to support complex bindings at least the following key concepts must be supported (1) support for group communication, (2) support for operation and stream based communication, (3) support for dynamical QoS features (dynamic = binding may be reconfigured during its lifetime), (4) hiding from the application programmer the programmatic problems related to constructing the complex bindings.

A suitable ORB implementation should implement a small core responsible for the most critical functionality, and standard ways to plug-in custom services, bridges, and communication protocols. This enables customising the ORB to match specific needs, including those of Pilarcos.

In the next sections we show how the CCM provided functionality is exploited in the Pilarcos project, present guidelines for integration of the Pilarcos and CORBA environments, and evaluate what are the possible shortcomings and problems of CORBA and CCM.

3.2 Integrating the Pilarcos binding services to CCM

3.2.1 Integrating the bridging mechanisms

Pilarcos promotes an architecture capable of instantiating proper bridging components on-demand. These bridging components are selected according to the negotiated binding contract as explained in an accompanying document [2]. The responsibilities of the bridging infrastructure includes adaptation to differences in heterogeneous object references, adaptation to differences in interface definition formalisms and interface semantics, and providing support for establishment of heterogeneous communication channels.

Adaptation to heterogeneous object references

When an object reference is brought into a system it must be converted to a format understandable within the system. When the object reference comes from a foreign system it always traverses through a bridge which performs the needed conversions and stores the mapping information needed when the object reference is used. Depending on the bridging strategy also an object reference leaving the system may have to be changed to a globally understandable standard format. Example of this kind of format is the Interoperable Object Reference, IOR. The object reference mapping can be realised, for

example, with proxy objects residing in the bridge, performing conversions according to standard mappings.

Adaptation to differences in interfaces

The adaptation to differences in interfaces can be divided into two main categories (1) adaptation to differences in interface semantics, and (2) adaptation to differing interface definition formalisms. The adaptation to differences in interface semantics is achieved by a conceptual separation of the client and server (=target) interface. CORBA itself does not separate these two concepts. In CORBA an interface is described with IDL and both the client and the server must obey the same description. The idea of a separate client and server interface stems from the fact that in a system spanning over multiple independent organisations the client and server components may have a life cycle independent of each other. Especially when the components are bought from independent component markets the client components cannot be assumed to strictly obey the server component interfaces. Instead the client and server interfaces should be dynamically bound by supporting infrastructure services using pre-made adapters. Whereas the separation of an interface and its implementation make it possible to change the implementation without affecting the interface, the separation of the client view of an interface and the server view of an interface makes it possible to change either one of them without affecting the other (interface substitutability) and enables interoperability between otherwise incompatible interfaces. The adaptation between client and server interface behaviour is in most cases interface specific and must be done separately for each pair of interfaces. This raises a need to provide development support for the construction of the needed adapters.

The adaptation to interface definition formalisms is needed because of the fact that there exists several differing interface definition languages with different concepts and syntax. In case the client and server interface do not differ only in their structure, but also in the formal concepts and types used to describe their structure there is a need to perform conversion between these concepts. All the concepts may not be mappable to each other which makes things complicated and the result may be non-equivalent conversions. For example if interface formalism A supports the definition of objects passable by value but interface formalism B does not, the bridge must convert the value objects to normal objects supported by formalism B. This kind of conversions may cause extra resource usage.

The basic strategy to build the bridging adapters in each case is the same (1) the adapting object must have access to the interfaces to be adapted (e.g. by inheriting their definitions or accessing them via some repository), and (2) the adapting object acts as a server to the client side interface, and as a client to the server side interface. Following IDL-description describes an exemplary adapter component mapping OMG IDL based client and server interfaces.

```
Interface ClientInterface{
  void service();
};

Interface ServerInterface{
```

```
  void business_service();
};

component Server{
  provides ServerInterface s;
};

component Client{
  uses ClientInterface c;
};

component Adapter{
  uses ServerInterface s;
  provides ClientInterface c;
};
```

Exploiting the new CCM features a simple adapter implementation might look something like the following:

```
// 'AdapterExecutor' is the generated part of the component.
// Adapter class represents the parts in the developer responsibility.
public class Adapter extends 'AdapterExecutor', ...{
   // The service operation in the client interface
   public void service(){
       // Bootstrapping the interface reference from the receptacle.
       ServerInterface s=get_connection_s();
       // Calling the business service on the server interface
       s.business_service();
   };
};
```

In the above example the interface definition languages used by the client and the server do not differ but if they did the adaptation could be done by exploiting the language mappings the adapter could inherit the interface definitions with programming language specific mechanisms. Figure 3.1 visualises a view-object adapting differences between client and server interfaces.

There is also a case where only the interface definition formalisms differ. In this case it would be possible to use a generic adapting mechanism in which a generic adapter object makes conversions between arbitrary interfaces. An example is when a MIDL definition is generated from an existing IDL definition according to a standard mapping. The generic adapter could do the needed conversions on the fly for an arbitrary number of interfaces. Some conventional CORBA/COM bridges use this kind of mechanism. Generic adaptation however would be extremely hard in cases where the structure of the interface signature and semantics of the client and server interfaces differ. Interpreting the semantic differences in a generic way would require excessive amounts of descriptional metadata.

The recommended approach is that the adapters should be pre-generated/implemented and stored into organisational software repositories (e.g.

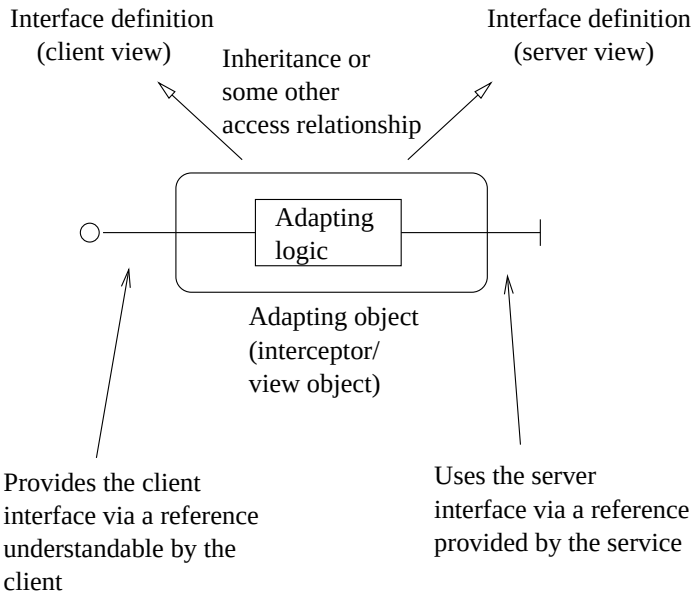


Figure 3.1: A view-object adapting differences between client and server interfaces.

into an assembly repository). The Pilarcos type repositories are then entered with the information needed to choose the adapters to be used at run-time.

3.2.2 Integrating the channel templates

In order to support custom communication protocols it is necessary for the middleware platform to provide some kind of pluggable protocols framework. The pluggable protocols framework should be standardised in order for the channel building blocks (e.g. the protocol objects) to be portable. As it is now, CORBA does not define a standard framework for plugging in additional protocols but many ORB implementations provide their own proprietary frameworks providing partial solutions. The exact requirements for the pluggable protocols framework should be evaluated further in order to provide a concrete recommended design for one that satisfies the Pilarcos needs. For example the TAO [10] pluggable protocols framework provides a good starting point for defining such.

3.2.3 Integrating the application binder

In order to provide the applications with standard application programming support the Pilarcos architecture [2] introduces the concept of application binder. The responsibilities of the application binder include intercepting all requests to and from a component, acting as a policy guard for application and environment specific policies, and automating the binding between the client application and the desired service. When an

abstract concept, like the application binder, is brought to a technology specific world, certain problems occur. In case of the application binder, CCM and CORBA already provide some of the needed functionalities, and some functionalities may be hard to integrate smoothly. Keeping this in mind it is not clear whether the application binder functionality should be integrated into the CORBA ORBs, integrated into the object adapters and CCM containers, implemented as an external CORBA objects service, or integrated into the component executors generated by the component implementation framework

Comparison of the Pilarcos application binder and CCM container

The relation of the container concept defined in the CCM model and the application binder is complicated in many ways because (1) the container has a similar role than the application binder, but essentially only on the server side, and (2) the container has some differing responsibilities from the application binder. The similarities between the container and the application binder include

- both are conceptually between the ORB and the component,
- both intercept the requests to the components interfaces and perform various services during interception,
- both rise the abstraction level of the component programmer.

However, there exists various fundamental differences between the container and the application binder concepts. These differences include

- container acts as the component's and its home's run-time environment,
- containers intercept only the incoming requests and the interception is done for all component's external interfaces (depends however on the container implementation strategy),
- application binders are created for specific interfaces in already existing components. This means that instead of acting as a runtime environment the application binders themselves need to be executed in some existing environment (e.g. in a container),
- application binders must intercept both the incoming requests and the outgoing requests but only for a selected set of interfaces - those which take part in a complex communication or in a federated communication,

Review of the possible application binder integration strategies

This section provides a review four possible application binder integration strategies in the CCM environment. This includes (1) the container based integration, (2) the strategy where the application binder is implemented as a CORBA object/component (3) the component executor based integration, and (4) a combination of the four.

Container based integration strategy If the application binder functionality would be integrated directly into the container it would become, from component's point of view, an invisible part of its environment accessible only through the container interfaces. For some remote component, the application binder could look like a normal CORBA object The container should also be able to associate application binders with specific interfaces of the component. The container implementation could use the POA servant manager mechanisms and the CORBA portable interceptor [8] mechanisms to intercept requests to and from the components interfaces. This integration strategy would be a good solution in the sense that the application binder would be fully incorporated into the standard environment and the application binder interface would always be accessible to the component via the container. Some problems with this approach are that the standard container interfaces might have to be extended and the container would have to store and manage interface specific binding relations. Also customising the application binder functionality would become more difficult.

Component based integration strategy Next implementation strategy to be evaluated is to implement the application binder as a CCM-component. The application binder component would provide at least a management facet and a generic call facet. The application binder could then be created with its home and its generic call interface would be connected to a specific receptacle of an arbitrary component anytime after the component itself is instantiated. In this approach the component developer would get access to the application binder interface by declaring the application binder usage in the component interface definition with a 'uses' clause. This approach does not sound as good as the previous one and the problems included are how to provide support for automated bootstrapping of the application binder interface and how the application binder could intercept the requests going to the component. In case of loosely-coupled communication (using e.g. the event sources and sinks) the situation would become even more complicated. In addition to previously mentioned interfaces the application binder component should provide a registration mechanism which components could use to register specific facets and the event handling mechanisms for interception. The application binder would then masquerade these interfaces and register itself to the object adapter as the receiver of the requests to them. Implementing the application binder as a CORBA component could also pose performance problems and excessive resource consumption because the application binder components and their homes would need to be instantiated in addition to the components they support.

Executor based integration strategy In this strategy the application binder would be integrated into the component executor itself. As described before the component consists of an executor generated from the information contained in the IDL and CIDL definitions and the business logic added by the developer. It would be quite easy to extend the component executor by generating additional classes implementing the application binder functionality. The generated application binder could then be customised to match any specific needs.

The fact that the component receptacles, facets, event sources and sinks, and their management operations are also generated into the component executor justifies this approach even more. This way the application binder would become an integral part

of every component and the business logic would always have access to it. Because the application binder would be part of the component executor naturally wrapping the business logic the request interception would not be a problem but a natural consequence. The main problem with this approach is of course that the component executors grow larger. This could be avoided by extending the CIDL to support the definition of application binders so they could be excluded from components not needing them.

Another possible problem is a too tight integration of application binders and the component. This is a complex question because the application binders have functionality which should be tightly coupled with the component, but also functionality that might be reasonable to embed into the environment (e.g. handling of environment/domain specific policies).

Combining the integration strategies Of the evaluated group the executor based integration strategy looks most promising and is considered to be the recommended strategy. It is in many ways very natural and its biggest problems are related to the handling of environment specific tasks at the component level. The environment specific parts can, however, be integrated into the container, or implemented using portable interceptors [8] or some other meta-programming mechanism. When CCM-implementations become more mature and prototyping is possible the recommended strategy will be adjusted according to experience gained.

3.3 Exploiting the CCM packaging and deployment architecture

The CCM packaging mechanisms are quite flexible offering possibilities to describe almost anything related to packaged components and assemblies. As a drawback, the XML DTD's of the descriptors are quite massive and hard to read. The flexibility of the descriptors gives deployment tool implementors almost free hands but very complex descriptors decrease portability and usability. The Pilarcos project exploits the descriptors by defining the necessary features related to the components and component assemblies to be used in constructing the prototypes later in the project. The concrete structure of the descriptors will depend on the CASE-tools and the new middleware services to be introduced.

3.3.1 Assembly descriptors

Assembly descriptors may have three different representations in the system: (1) Master version of the assembly descriptor which describes the features of the assembly but does not include any information of the physical configuration, (2) installed assembly descriptor which is derived from a master version and contains additional information describing the installed components (location etc.), and (3) activated assembly descriptor which is derived from an installed assembly descriptor and contains information concerning the activated assembly instances. Example version tree is visualised in Figure 3.2.

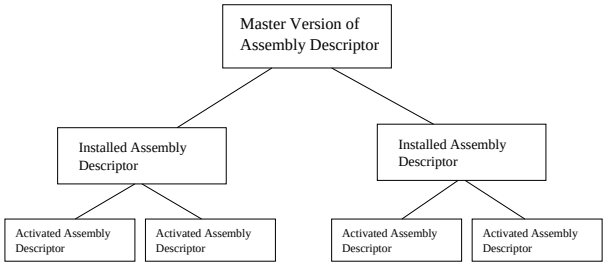


Figure 3.2: Example instance of an assembly descriptor version tree.

The main version of the assembly descriptor is located in the assembly repository. The other versions might be located in the same repository or stored locally by the entities using them (Application Component Deployers and Assembly Objects respectively). Whether a centralised or distributed design is used is an implementation oriented question but for explanatory simplicity a centralised version-tree is assumed to exist from now on.

3.3.2 Installation and un-installation of component assemblies

The installation procedure consists of installing component software in server hosts as specified in the assembly descriptor. The installation could be done either at development time or on-demand at runtime. In this project we are promoting development time installation because runtime installation, although extremely flexible, is generally too slow and error-prone. However some programming environments (e.g. Java-based platforms) may have special features which make the runtime installation and code distribution a realistic option.

An exemplary installation procedure of a CCM assembly is visualised in Figure 3.3 and described below.

1. (not shown in Figure 3.3) Development tool interacts with application component deployer and passes the id of the assembly descriptor to be installed,
2. application component deployer fetches a copy of the the related master assembly descriptor from the assembly repository,
3. application component deployer interacts with 'ComponentInstallation' objects in each server host and asks them to install specific components by giving a package and implementation id of each component,
4. 'ComponentInstallation' objects interact with the assembly repository and fetch relevant component files. If the needed files already exist in a specified host, then corresponding 'ComponentInstallation' object does nothing.

After the installation, the application component deployer updates its own copy of the assembly descriptor and stores it in a persistent form. The un-installation procedure

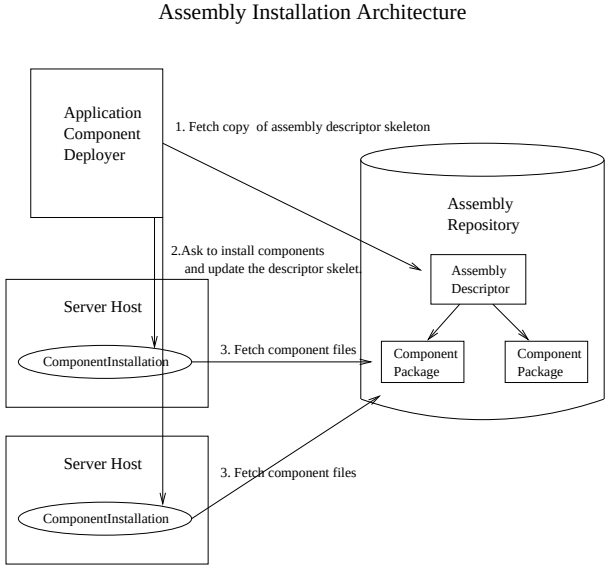


Figure 3.3: Overview of the assembly installation architecture.

is opposite to the installation procedure. Development tool interacts with application component deployer and asks it to un-install a specified assembly. Application component deployer then checks the component locations from the updated copy of the assembly descriptor and interacts with 'ComponentInstallation' objects in order to remove the software.

3.3.3 Activation and de-activation of component assemblies

The activation procedure consists of instantiating and connecting the component servers, component homes, and components in each host. The activation procedure, like the installation procedure, could be done either at development time or at runtime. Runtime activation introduces some extra overhead and longer response-times but provides more dynamism and reduces the resource usage in very large systems where the number of installed assemblies may be huge and there may not be enough resources to keep them all active at the same time. The performance should not be a severe problem because the most popular services will probably stay active all the time and only the rarely used services will be activated and passivated on-demand. This justifies the approach of promoting a run-time activation procedure of the component assemblies.

An exemplary activation procedure is visualised in Figure 3.4. Among other things, the figure contains a 'ComponentActivator' object which is not part of the CCM standard deployment interfaces but is needed for a complete deployment and activation architecture. The activation procedure is further described below.

1. (not shown in Figure 3.4) Business service deployer asks application component

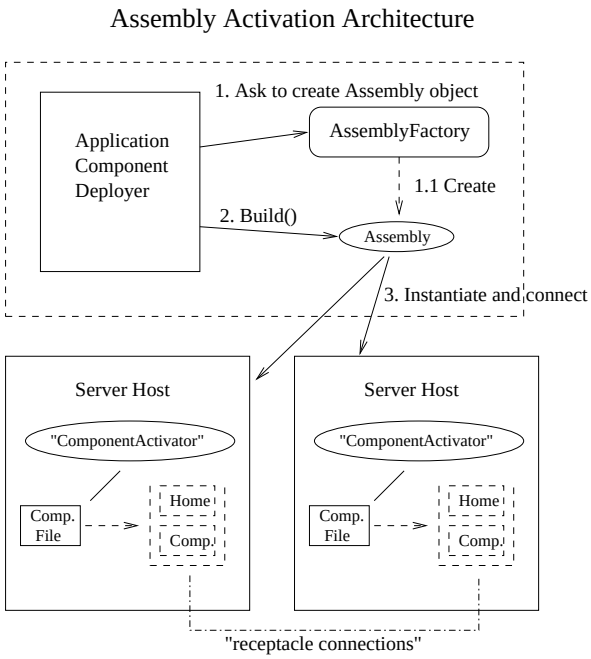


Figure 3.4: Overview of the assembly activation architecture.

deployer to activate a specific assembly,

2. application component deployer contacts an 'AssemblyFactory' and asks it to create an Assembly object. The reference to the assembly descriptor containing the installation information is passed to the 'AssemblyFactory',

3. 'AssemblyFactory' creates an 'Assembly' object to guide the activation process,

4. application component deployer asks the 'Assembly' object to build the assembly,

5. 'Assembly' object interacts with 'ComponentActivator' objects in each server host and asks them to create the necessary container environment plus home and component instances. After the instantiation phase, the Assembly object interacts with each component and creates the necessary connections by calling operations on the components receptacles and event sinks.

The deactivation procedure is quite simple in principle. Some entity contacts the 'AssemblyFactory' and asks it to tear-down the instantiated assembly and destroy the corresponding 'Assembly' object. 'Assembly' object can tear-down the assembly by interacting with the component, component homes, and the 'ComponentActivator' objects in each server host. The timing of the deactivation, instead, is not a trivial problem. The deactivation could be done for example by some entity which monitors the usage

of the services and tears down the rarely used assemblies. Some additional problems raise from the fact that an assembly may be part of several larger designs in which case tearing down only one assembly would not free all the resources but would handicap the service provided by all the assemblies together. These kind of problems have an implementation oriented flavour and are not solved in this document.

3.4 A complete scenario for accessing federated services

3.4.1 Discovering the business services

As described in the conceptual Pilarcos-architecture [2], the service requests and service offers are considered to be mediated by business service brokers. Service offers can be exported to and imported from them. The business service brokers use the services of the underlying type repositories when matching the requests. Business service brokers may exist in all organisational domains. On the other hand there might exist only globally used business service brokers (e.g. global service portals). The actual situation is probably going to be somewhere in between. The business service brokers may be (manually) federated, in which case the amount of service offers reachable via one broker is increased. The federated brokers should have the same understanding of the existing type relationships in order for the broker federation to have any added value. This means that the underlying type repositories must be federated also or at least the parts relevant in federated type matching.

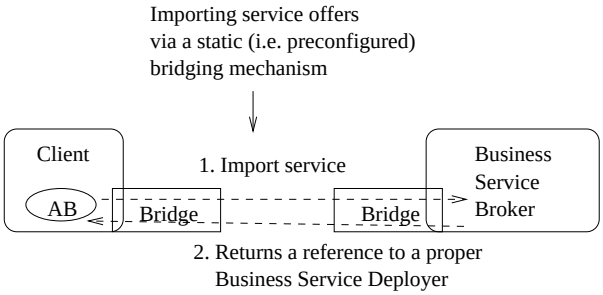


Figure 3.5: Importing the service offers via a static bridging mechanism.

The client application binders use the business service brokers during the federated binding process to get access to the client desired services. Both the client and the business service brokers are expected to use some middleware platform as a communication mechanism. If the client and the business service broker use different platforms the communication between them must be bridged. This bridging is static in nature in the sense that one or at most few pre-configured bridging protocols are probably supported. Figure 3.5 visualises the service offer import situation and related bindings.

3.4.2 Activating the services

After discovery of a suitable service the service preparation and activation phase takes place as explained in the Pilarcos architecture document [2].

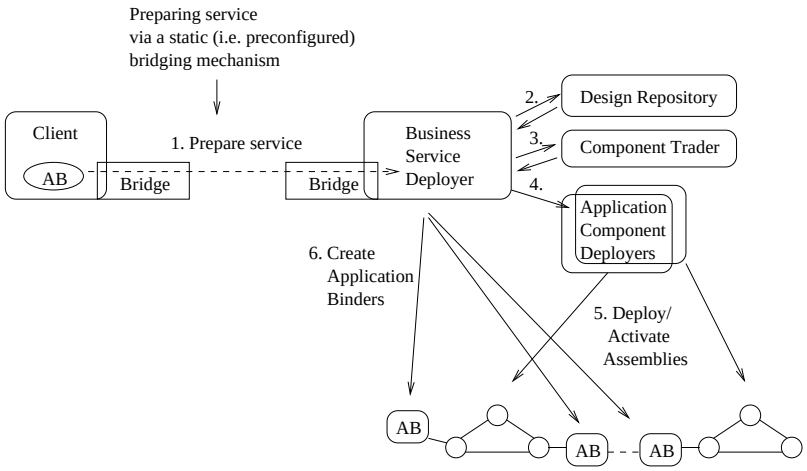


Figure 3.6: Client communicating with business service deployer via a static bridging mechanism.

The service preparation process is initiated by the client application binder which, after receiving the reference from the broker, contacts the corresponding business service deployer and asks it to prepare the wanted service. The communication is established via an existing bridging mechanism very much like in the service discovery phase. Actions taken during the service preparation are shortly reviewed in Figure 3.6. The server side application binders get created during the activation process at selected connection points. The process of connecting the application binders within the organisation and computational design is controlled by the business service deployer during the binding process. The actions taken when establishing the communication between them is a subset of the federated binding case and uses the same underlying binding services.

3.4.3 Negotiating and establishing a federated binding

Federated binding scenario differs from the basic intra-design binding in that it involves negotiations of the federated binding contract with the initiating application binder residing in another organisational domain.

Initial suggestion for the federated binding contract is included in the service preparation request but in case the service offer exported to the broker did not contain all specific details a new contract can be negotiated by the interacting application binders.

Each partner of the contract stores its own version of the final binding contract in a local repository.

The federation contract contains information describing the service including [2]:

- A service type identifier (for each type system involved),
- service type specific QoS agreements,
- identifier of the related business architecture and the role played in that architecture,
- the computational design used to implement the business role and the role of the service in that computational design.

In addition the contract contains information describing the communication details like:

- A binding type identifier (for each type system involved),
- failure detection and recovery protocols,
- remuneration protocol,
- the technical interface expected by the client,
- communication protocol,
- channel type identifier (for each type system involved),
- interface reference for the channel controller managing the created channel.

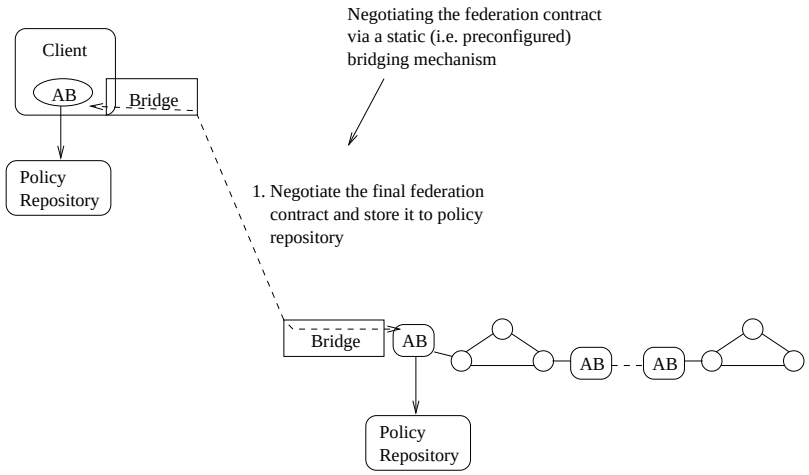


Figure 3.7: Application binders negotiating the final federated binding contract.

The contract information is stored into local policy repositories so that all components participating the service interactions have access to them. Figure 3.7 shows two application binders negotiating the federated binding contract.

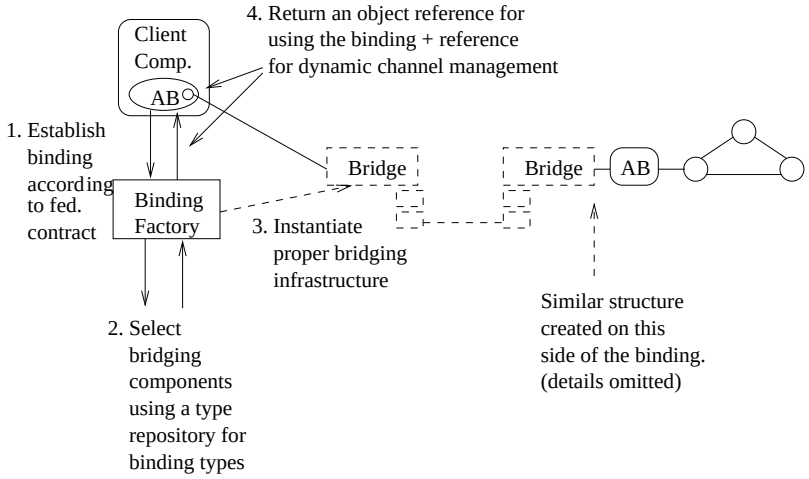


Figure 3.8: Application binders communicate with binding factories to select the bridging components.

3.4.4 Establishing the communication channel

In order to establish the actual communication channel identified in the negotiated contract, the application binders must communicate with a local binding factory. The binding factory is given the binding type information and it searches the underlying type repository for a channel type matching (1) the binding type information and (2) the requirements of the local platform.

The binding factory then creates a channel controller to manage the binding to be created and a channel section for each channel endpoint within its own domain. Finally it registers the channel controller with all interfaces it should manage. The application binder is returned an object reference through which the service can be invoked and a reference to the management interface of the newly created channel controller. The Figure 3.8 shows the interactions between the application binder and the binding factory.

The channel parts at each communicating side instantiate all the needed protocol objects which establish a concrete transport connection. Example of what this process might look like is visualised in Figure 3.9. The example uses similar strategy used by the TAO connection establishment framework [10, 15]. Following is a short review on the interactions performed (1) The connector at the client side contacts an acceptor at the server side, (2) both of them instantiate the needed protocol objects and register the management interfaces to the channel controller, (3) the bridging components use the instantiated channel to send and receive requests. In addition resource reservation protocols might be used.

The figure also shows the channel controller management interface needed to be able to dynamically re-configure the channel. This is the interface returned to the application binder by the binding factory. In addition to local channel management

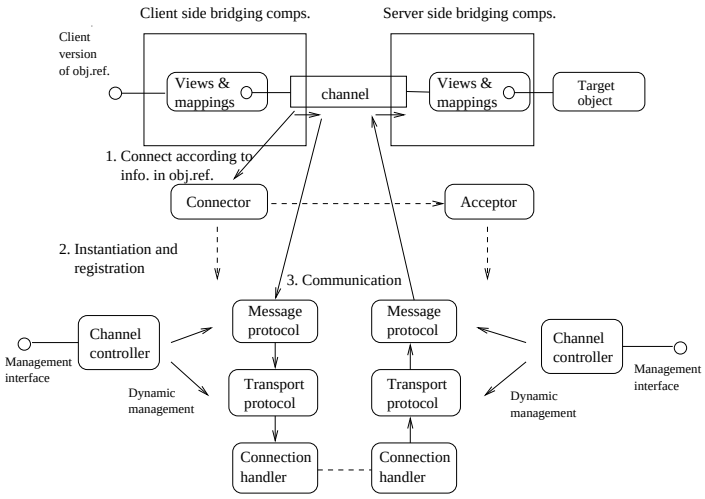


Figure 3.9: Example instantiation of a communication channel.

capabilities, communication between channel controllers in different domains might be needed to enable distributed resource reservation.

Chapter 4

Conclusion

In order to address the requirements placed by the Pilarcos architectural designs and their mapping to technology specific deployable software the CCM deployment architecture was reviewed. It was discovered that the capabilities of the deployment model are enough to support the Pilarcos model.

The separation of types and templates as well as the separation of client and server interfaces can be tackled with additional type handling services (e.g. a type repository) and by using configurable bridges dynamically exploiting proper adapters. A separate document concentrating on the software design aspects provides an evaluation on supporting the type handling in the OMG environment. The best candidate to be used as a platform for Pilarcos type repository is the OMG MOF.

Design strategies for introducing the symmetrical support facilities to both ends of the object bus were evaluated. These new support facilities include the deploying mechanisms, bridging components, and the binding facilities (like the application binders). The deploying mechanisms of the CCM architecture meet the Pilarcos requirements well enough. The possible application binder integration strategies were given a detailed evaluation. Possible integration strategies discovered were (1) the container based integration strategy, (2) the component based integration strategy, and (3) the executor based integration strategy. Of these four the design where the application binders would be integrated with the component executors in addition to using some supportive meta-programming mechanisms was considered most practical. The concrete design to be used in Pilarcos prototypes might, however, be affected by the future research and experimentation.

The requirements placed for the communication channels, like manageability by QoS negotiations and business policies and adaptability to changes in resource availability can be addressed with the introduction of the symmetrical support facilities. At the higher level the application binders supporting federated binding negotiations and enforcement of the organisational business policies and binding factories for selecting the proper channel components based on existing type information will provide the needed support. In lower levels the needed platform support for communication should include support for multiple user configurable protocols and channel controlling mechanisms for dynamic configuration. In order for the developers to be able to plug-in their own protocols and other channel constructs the underlying platform should support a pluggable

protocols framework flexible enough to support all requirements placed for the channel components. This framework should be a standard part of the platform in order to achieve portability of the channel templates and to simplify the development work. At the moment CORBA provides no standard pluggable protocols framework and further research and experimentation with the existing proprietary designs must be performed in order to come up with a concrete design for a Pilarcos compatible framework.

The requirements placed for the Pilarcos compatible components, such as life cycle service support, support for composability, and the need to support multiple interfaces, are answered by the new features introduced by the CCM. The relevant new features include the concept of component, the concept of component home, the concept of component container, and the concept of a component assembly. In order to define the requirements placed by the component for its environment the capabilities of the new IDL extensions, component containers, and packaging descriptors will be exploited.

Further recommendations also need to be created on management facilities for CORBA.

References

[1] Kahkipuro, P. *Overview of CORBA*, 1998. http://www.cs.Helsinki.FI/u/kahkipur/corkur/corbaintro.ps.

[2] Kutvonen, L., Haataja, J., Silfver, E., and Vähäaho, M. Pilarcos architecture. Tech. rep., Mar. 2001. C-2001-xxx.

[3] Object Management Group. *CORBAfacilities: Common Facilities Architecture and Specification*, 1997.

[4] Object Management Group. *CORBAservices: Common Object Services Specification*, 1997.

[5] Object Management Group. *Common Facilities RFP-5: Meta-Object Facility*, 1998. OMG TC Document cf/96-05-02.

[6] Object Management Group. *CORBA Component Model - Volume 1.* Framingham, MA, USA, 1999. OMG Document orbos/99-07-01.

[7] Object Management Group. *CORBA Management: ORB Instrumentation*, May 1999. RFP 99-05-07.

[8] Object Management Group. *Common Object Request Broker Architecture (CORBA) v2.4.2*, february 2001. OMG Document formal/01-02-01.

[9] OMG. *Meta Object Facility (MOF) Specification.* Object Management Group, April 2000.

[10] Schmidt, D. *The ACE ORB.* http://www.cs.wustl.edu/ schmidt/TAO.html.

[11] Soley, R., and Stone, C. M., Eds. *Object Management Architecture Guide.* John Wiley & Sons, Inc., June 1995. Revision 3.0.

[12] Soley, R., and the OMG Staff Strategy Group. *Model Driven Architecture.* Object Management Group, Nov. 2000. OMG white paper, draft 3.2. Document number omg/2000-11-05. Also http://cgi.omg.org/cgi-bin/doc?omg/00-11-05.ps.

[13] SUN Microsystems, Inc. *Enterprise JavaBeans Specification v1.1*, 1999.

[14] Wang, N., Schmidt, D., and O'Ryan, C. *Overview of the CORBA Component Model*, 2000.

[15] WANG, N., SCHMIDT, D., OTHMAN, O., AND PARAMESWARAN, K. *Evaluating Meta-Programming Mechanisms for ORB Middleware*, 2001. http://www.cs.wustl.edu/ schmidt/new.html.

[16] WANG, N., SCHMIDT, D., PARAMESWARAN, K., AND KIRCHER, M. *Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications*, 2001. http://www.cs.wustl.edu/ schmidt/new.html.

ISSN -
ISBN -
Helsinki 2001