An Architectural View of Distributed Objects and Components in CORBA, Java RMI, and COM/DCOM

František Plášil[1], Michael Stal[2]

[1] *Charles University*
Faculty of Mathematics and Physics,
Department of Software Engineering
Prague , Czech Republic
e-mail:plasil@nenya.ms.mff.cuni.cz
www: http://nenya.ms.mff.cuni.cz

[2] *Siemens AG Corporate Technology*
Dept. ZT SE 1
Munich, Germany
e-mail: Michael.Stal@mchp.siemens.de

Abstract

The goal of this paper is to provide an architectural analysis of the existing distributed object oriented platforms. Based on a relatively small number of design patterns, our analysis aims at a unified view of the platforms. We achieve this by articulating a series of key issues to be addressed in analyzing a particular platform. This approach is systematically applied to the CORBA, Java RMI, and COM/DCOM platforms.

Key words:

CORBA – Java RMI – COM/DCOM – Distributed Computing – Distributed object – Design patterns – Software architecture

1 Introduction

With the wide spread utilization of object technology, it has become more and more important to employ the object oriented paradigm in distributed environments as well. This raises several inherent issues, such as references spanning address spaces, the need to bridge heterogeneous architectures, etc. It is the main goal of this paper to provide an architectural analysis of current software platforms in this area. One of the obstacles to overcome in order to achieve this aim is the fact that the available descriptions of these platforms speak different languages. Thus to target the issue, we have decided to employ design patterns [3,19] as a common denominator which will help us provide an unified view on the platforms analyzed.

We focus on the following key distributed object platforms: CORBA, Java RMI, and COM/DCOM. The first one, CORBA, is specified by OMG [12], which is the largest consortium in the software industry. CORBA has undergone an evolution ranging from CORBA 1.0 (1991) and CORBA 2.0 (1995) to CORBA 3.0, which is soon to be released. The Java environment, designed by Sun Microsystems, has probably experienced the greatest evolution recently. From the broad spectrum of the Java platform segments, we will focus on Java RMI [22], which targets working with distributed objects. The last platform analyzed is the Microsoft Component Object Model (COM). This platform has also been evolving gradually along the milestones OLE, COM, DCOM, and COM + [18]. In this paper, we will focus on COM/DCOM [6], as this is where Microsoft targets distributed objects.

The paper is structured as follows: Section 2 articulates the general principles of working with distributed objects. The division of the section reflects our approach to the architectural analysis - basic principles, basic patterns, provision and employment of a service, and inherent issues. Using the same structuring as in Sect. 2, we offer analyses of CORBA (Sect. 3), Java RMI (Sect. 4), and COM/DCOM (Sect. 5).

Due to the limited size of the paper, we could not focus on security questions and benefits of code mobility over the Internet. Also, a thorough evaluation of each platform could not be provided. All of these areas have become very broad and each of them deserves at least a separate paper.

2 Distributed objects

2.1 Basic principles

2.1.1 Request & response

Under the term *distributed objects*, we usually understand objects which reside in separate address spaces and methods of which can be subject of remote method calls (a *remote call* is issued in an address space separate to the address space where the target object resides). By convention, the code issuing the call is referred to as the *client*; the target object is referred to as the *server object* (or simply *remote object*); the set of methods which implements one of the server object's interfaces is sometimes designated as a *service* that this object provides. Similarly, the process in which the server object is located is referred to as a *server*.

An important goal of the client and server abstractions is to make it transparent how "far" the client and server spaces actually are - whether they reside on the same machine, are on different nodes of a local network, or even reside on different Internet URLs (thus being "intergalactic"); as a special case, client and server may share the same address space.

Because it is inherently delivered over a network communication infrastructure, a remote method call is typically divided into the *request* (asking the service) and *response* (bringing results back to the client) parts. In principle, from the client's view, the request and the response corresponding to a remote method call can be done as one atomic action (*synchronous call*), or they can be separated, where the client issues the request and than, as a future action, issues a wait for the response (*deferred-synchronous call*). Sometimes the response part may be empty (no out parameters and no functional value). In this case, the corresponding method is usually termed a *one-way method*. A one-way method can be called asynchronously, where the client does not have to wait till the call is finished. In a distributed environment the exactly-once semantics of remote calls is practically impossible to achieve; real distributed platforms ensure the at-most-once semantics of a synchronous and deferred-synchronous call (exactly-once semantics in case of a successful call, at-most-once semantics otherwise); best-effort semantics is ensured for a one-way method.

2.1.2 Remote reference

One of the key issues of remote method calls is referencing of remote objects. Classically, in a "local" case, in a method call `rro.m(p1,p2, ..., pn)` , `rro` contains a reference to the target object, `m` identifies the method called, and some of the parameters can contain references to other objects as well; let us suppose only one of the parameters, say `pj`, contains a reference. However, in a distributed environment we face the following issue: `rro` should identify a remote object over the network, and so should `pj`. It is obvious that classical addresses will not do as the references, at least for the following reasons: in addition to the data record of the target object, a reference has also to identify the node and the server in which the target object resides. Moreover, the target object may implement more (non polymorphic) interfaces; thus, `rro` should also identify the particular interface which the target object implements, and to which `m` belongs. By convention, a reference that contains all this information is termed a *remote reference*. Hence, a remote reference identifies a service. In addition, representation of a remote reference has to span the differences in the hardware architectures of the nodes where the objects involved in a

particular remote method call reside.

2.1.3 IDL Interface

In principle, a client's code and the server object that is subject to a remote call from the client can be implemented in different languages and can run on heterogenous architectures. To span this kind of difference, the interfaces of a server object are specified in an architecture-neutral Interface Definition Language (IDL). Typically, IDL provides constructs for specification of types, interfaces, modules, and (in some cases) object state. However, there is no means for specifying code of methods. Usually a mapping from IDL to standard programming languages, such as C++ and Java, is a part of an IDL definition. CORBA IDL and Microsoft MIDL are examples of IDL languages.

2.1.4 Proxy: local representative

To bridge the conceptual gap between the remote and local style of references, both in the client and server code, the actual manipulation with remote references is typically encapsulated in wrapper-like objects known as client-side and server-side proxies. The *client-side proxy* and the corresponding *server-side proxy* communicate with each other to transmit requests and responses. Basically, the client-side proxy supports the same interface as the remote object does. The key idea behind proxies [17] is that the client calls a method `m` of the client-side proxy to achieve the effect of calling `m` of the remote object. Thus, the client-side proxy can be considered a local representative of the corresponding remote object. Similarly, the key task of a server-side proxy is to delegate and transform an incoming request into a local call form and to transform the result of the call to a form suitable for transmitting to the client-proxy. Thus, a server-side proxy can be considered as the representative of all potential clients of the remote object.

2.1.5 Marshalling: transmitting request & response

Both the request and response of a call are to be converted into a form suitable for transmitting over the network communication infrastructure (a message, or a TCP socket connection for transmitting streams might be an example of the infrastructure). Typically, serialization into a byte stream is the technical base of such conversion. By convention, this conversion is referred to as *marshalling* (the reverse process is *unmarshalling*). Some authors use these concepts in a narrower sense, where marshalling/unmarshalling refers only to conversions of the parameters of a remote call.

The key issue of marshalling and unmarshalling is dealing with objects as parameters. The following two approaches indicate the basic options: passing remote references and passing objects by value. Suppose `rro.m(rp,lp)` is a remote call issued by a client C. The `rro` reference identifies a proxy which encapsulates a remote reference to a remote object RO in a server S; the `rp` represents a remote reference, `lp` is a reference to a local object which is to be passed by value. Typically, when the request corresponding to the call is delivered to S, a client-proxy encapsulating `rp` is automatically created in S. On the contrary, a copy of the `lp` object is created in S.

2.2 Combining basic principles: basic patterns

Design pattern are the "smallest recurring architecture in object oriented systems" [24]. For more details on design patterns (patterns for short) refer, e.g., to Buschmann et al. [1] or Pree [14], and particularly to the classical catalog of design patterns by Gamma et al. [3]. We will use informally just a few of them in the following sections to explain the basic ideas behind the architectural concept of the distributed object platforms.

2.2.1 Broker pattern

The Broker pattern in Fig.1 reflects the classical client/server relationship implementation in distributed environment. The role of the Client, Client-side Proxy, Server-side Proxy and Server was explained in the previous sections, so we can focus on the Broker object, which is in this pattern to capture the communication between Client-side Proxy and corresponding Server-side Proxy. In a loop, the Broker forwards requests and responses to the corresponding proxies. These requests and responses are encoded in messages by which the Broker communicates with the proxies. Figure 2 demonstrates a typical scenario of object interaction for delivering a request and receiving a response. Another scenario (Fig.3), illustrates how a server registers the service it provides with the Broker object. The Broker pattern, as described so far, is a *Broker with indirect communication* between proxies. In the typical modification of the Broker pattern referred to as Broker with *direct communication*, the Client-side Proxy and the corresponding Server-side Proxy communicate directly after a communication channel was initially established. A Bridge object is used to communicate with other Brokers.

2.2.2 Proxy pattern

The essence of the proxy idea (Sect. 2.1.4, 2.2.1) is captured in Fig.4. Both the Proxy and Original classes support the same interface defined by AbstractOriginal. A typical scenario of interaction among instances of these objects is depicted in Fig. 5.

2.2.3 Abstract Factory pattern

In principle, a factory is an object which creates other object; usually a factory is specialized for creating objects of a particular type. In distributed environments, factories are typically used on the server side (after a server has been initialized) to create new objects on a client's demand. The main reason for using factories by clients is that the usual language tools like "new" cannot be used in distributed environments. In a routine scenario the client issues a request to an object factory to create a new object; the object factory returns a remote reference of the newly created object. Typically, the created object resides in the same server as does the Object Factory involved in the request. The essence of the factory idea is captured by the Abstract Factory pattern (Fig.6). In particular, it illustrates that a Concrete Factory is devoted to creation of products of specific types.

2.3 Providing and employing a service

2.3.1 Registering a service with broker

In order to be remotely accessible, any service provided by a server has to be registered with a broker. As a result of the registration operation, the broker a creates a remote reference to the service (i.e. to a particular interface of the corresponding remote object). The remote reference is returned to the server and, e.g., can be registered with a naming and/or trading service (Sections 2.3.2, 2.3.3). Typically, following the Abstract Factory pattern, an object factory residing in the server would be used to create the new remote object implementing the service being registered.

2.3.2 Naming

Because the remote references are supplied by brokers in scarcely legible form, and so to enable the use of ordinary names, a distributed object platform typically provides a naming utility (naming for short). A *naming* defines a name space and tools for associating a name with a remote reference. Classical operations for resolving a name into a remote reference and associating a name with a remote reference are provided. Typical scenarios for a naming implementation include (a) providing a naming service accessible via a well-known remote reference, (b) embodying naming in the Broker's functionality.

2.3.3 Finding a service, trading

For a client, there are two ways to obtain a remote reference which identifies the requested service. First, it can ask naming to resolve the name the client knows, or, alternatively, the client can ask a *trading* utility (an analogy of yellow pages) to provide a list of remote references to the remote services possessing the properties which the client indicated as the search key. The typical scenario for trading implementation includes provision of a trading service accessible via a well-known remote reference.

2.3.4 Binding

As mentioned in Sections 2.1.5, 2.3.2, and 2.3.3., the client can receive a remote reference via naming or trading, or as a result of another remote method call (in fact if naming and trading are implemented as services, the client always receives remote references as a result of other remote methods calls - except for well-known remote references of, e.g., naming and trading services). It is a general rule of all the distributed object platforms that when a client receives a remote reference to a service, a proxy is created (if it does not already exist) in the client's address space, and the client is provided with reference to the proxy. At the latest with the first remote call that the client issues via the proxy, the connection between the client and the server is established (some authors also say that the client is *bound* to the server).

2.3.5 Static invocation and delivery

When the client was compiled with the knowledge of the requested service interface (e.g., in the form of an IDL specification), the remote methods' calls can be encoded statically in the clients code as calls of the proxy methods (*static invocation*). This is done fully in compliance with the proxy pattern (Section 2.2.2). Similarly, if the server code was compiled with full knowledge of a particular service's interface, the corresponding server-side proxy can contain statically encoded local calls to service's methods (*static delivery*).

2.3.6 Dynamic invocation and delivery

In principle, the client and the server can be compiled separately; thus they might not always be current with respect to the static knowledge of available interfaces. Moreover, there are specific applications in which the exact knowledge of the service interfaces is inherently not available at compile time (e.g., debugger, viewer, bridge). To overcome this obstacle at the client side, a proxy is usually provided with an interface (say `IRemote`) which allows issuing dynamically set-up calls (*dynamic invocation*). The name of the corresponding method is typically `invoke()`. As an example, suppose the `rop.m(p1, ..., pn)` call is to be issued dynamically as the signature of `m` was not known at compile time (`rop` is a reference to the proxy of the remote object). So at compile time the client code can contain the following call: `rop.invoke (method_name, paramlist_array)`. Thus before executing the invoke method we have to assign `m` to `method_name`, and fill `paramlist_array` with the parameters `p1, ..., pn`. Usually some kind of introspection is necessary to obtain/check details on the target interface at run time (types and numbers of method's parameters, etc.).

Similarly, new objects can be created in a server with interfaces not known at compile time (e.g., in a viewer or a bridge). Calls to these objects can be *delivered dynamically* if these objects are equipped accordingly. The usual technique employed is the following: Suppose the `rop.m(p1,...,pn)` call is to be delivered to the object RO identified by the `rop` remote reference; however, the object is located in a server which was compiled without any knowledge of RO's interfaces - except for the `IRemote` interface supported by all remote objects. `IRemote` contains the `invoke()` method which accepts dynamically created calls and forwards them to the actual methods of RO. Thus to be able to accept dynamic calls, RO has to implement the `Invoke` method. The server-side proxy associated with RO is very simple - it supports just the `IRemote` interface; therefore the code of the server-side-proxy can be very generic and

can be used for any server object supporting the `IRemote` interface.

Some Broker implementations, e.g. CORBA, even allow for combining static invocation at the client side with dynamic delivery at the server side and vice versa (combining dynamic invocation with static delivery).

2.4 Inherent issues

2.4.1 Server objects garbage collection problem

The server objects not targeted by any remote reference can be disposed and should be handled by a garbage collector. As opposed to the classical techniques used in single address space, garbage collection in distributed environment is a more complex issue that has been the subject of much research (e.g., Ferreira et al.[2]). We identify the following three basic approaches to garbage collection of distributed objects: (1) Garbage collection is done in the underlaying platform, e.g. at a distributed operating system level; this approach remains in an experimental stage [23]). (2) The broker evaluates the number of live references (e.g., derived from the number of live connections). (3) The server keeps track of the number of references provided to the outside; each of its clients has to cooperate in the sense that if it does not need a remote reference any more, it should report this to the server. This technique is called *cooperative garbage collection*.

The approaches (2) and (3) are based on reference counting: once the reference count drops to zero, the target object can be disposed of. Despite of its simplicity, reference counting suffers from a serious problem - cyclic referencing among otherwise unreachable object prevents zeroing of reference counts.

2.4.2 Persistent remote references

Lifecycles of clients and servers are not synchronized. For example, a server can time-out or crash, and its client can still hold a remote reference to an object located in the server. Thus, remote references can persist with respect to the server they target. In most of the distributed object platforms, using a remote reference to an inactive server is solved by an automatic reactivation of the server. Typically, before the server times-out, it is given a chance to externalize the state of its objects; after the server is reactivated, its objects can be recreated and their state renewed (some kind of lazy loading can be employed in the reactivation process).

2.4.3 Transactions

Transaction are an important tool in making distributed object applications robust. Three consequences of working with distributed objects with respect to transactions should be emphasized: (1) Because of transitive nature of the client-server relation (Section 2.4.5), nested transactions are almost inevitable. (2) Employing multiple databases (one type of resources) is inherent to the distributed environment. This implies that two-phase commit has to be applied (prepare and commit phases). (3) As objects possess state, they can also be considered as resources taking part in transactions.

It is a well-known problem that if a resource needs another resource in order to finish its prepare phase, the requests to prepare have to be issued in a specific order (respecting the implied partial ordering of resources) to avoid deadlock [11,16]. As an example, consider an object A, which when asked to prepare, is to be stored into a database B. It is obvious that A cannot prepare after B is requested to prepare. Thus, a transactional system to be used in distributed object applications should provide some support for dealing with this problem.

2.4.4 Concurrency in server objects

In the basic variant of the Broker pattern (Section 2.2.1), the Server enters a loop and waits for incoming requests. In this variant, the Server responds to the requests sequentially. However, many implementations employ multithreading; the following is an example of this approach: the Broker, sharing address space with the Server, forwards each of the incoming requests to a separate thread in the Server (there can be, alternatively, a pool of available threads, a thread can be created for each request, etc.). As the Server is now *multithreaded*, a server object may be subject to invocation of several of its methods simultaneously. Naturally, synchronization tools have to be applied in code of the Server's objects in order to avoid race conditions, etc. In addition, several threads running in the Server may call the Broker at the same, e.g., to register a newly created object. By convention, a Broker supporting this concurrency is called a *multithreaded* Broker.

2.4.5 Applying client-server relation transitively

It is very natural for a server S be become a client of another server S'. Simply an object O in S contains Object Reference to an object O" in S'. Thus any method of O can invoke a method of O''. This very likely happens when a remote reference is passed as a parameter in a request to S. In fact, we face a transitive application of the master/slave pattern [3]. Furthermore, this is also the underlying idea of n-tier architectures [13]. Similarly, we can also achieve callbacks from a server to its client by passing a remote reference of the interface, which is to be the subject of a callback from the client to the server (see also Section 2.5).

2.5 Reactive programming

The advantage of easy passing of remote references as parameters makes it relatively easy to employ even-based (reactive) programming style without introducing specific callbacks constructs. Most of the distributed object platforms define some abstractions to support this style of programming with significant comfort. In principle, all of them are based on the Observer (Publisher-Subscriber) pattern [3]. The basic idea here is that a *listener* object (Subscriber) can subscribe to an event source (Publisher) to be notified about and event (asynchronous signal) occurrence. The Publisher announces what interfaces its listeners have to honor and what methods of these interfaces will be called to report on an event. This reporting can be just a notification (one-way method will do), or a callback (some results can be delivered to the Publisher). One of the key benefits of this approach is that the Publisher does not have to know its listeners at compile time; listeners subscribe dynamically.

3 Distributed objects in CORBA

3.1 Basic principles

3.1.1 Request & response

In CORBA, a client issues a request to execute a method of an object implementation. There is no constraint imposed on the location of the client and the requested object implementation (remote object); they can share an address space, can be located in separate address spaces on the same node, or can be located on separate nodes. A server (process) can contain implementation of several objects or a single object, or even provide an implementation of one particular method only. However, typically, a multithreaded server encapsulates implementations of multiple objects.

3.1.2 Remote reference

Remote references are called *Object References* and the target of an Object Reference must be an object that supports the `CORBA::Object` IDL interface (such objects are called *CORBA objects*).

3.1.3 IDL Interface

CORBA specifies the CORBA IDL Language and its mapping to several programming languages (e.g., C++, Java, Smalltalk). The IDL language provides means for interface definitions; there are no constructs related to object implementation (object state definitions have been proposed recently by OMG).

3.1.4 Proxy: local representative

Using the Broker Pattern terminology, the client-side proxy code is called *IDL stub*; the server-side proxy code is referred to as *IDL skeleton*. However, in CORBA, the concept "proxy" is used to denote an object created on the client side which contains the IDL stub plus provides some other functionality, e.g. support for dynamic invocation.

3.1.5 Marshalling: transmitting request & response

Both request and response are delivered in the canonical format defined by the IIOP protocol which is the base for the CORBA interoperability over the Internet (other protocols are defined in the OMG standard [12] as well, but IIOP prevails). On the client side, marshalling is done in the IDL stub or via the functionality of the dynamic invocation interface (Section 3.2). On the server side, unmarshalling is done in the IDL Skeletons (or in the Dynamic Skeleton Interface, DSI) and partially in the Object Adapter. In a request (response), Object References can be provided as parameters; remote objects can be passes by reference. Passing objects by value has been the subject of a resent OMG proposal [7].

3.2 Combining basic principles: CORBA architecture as pattern combination.

The CORBA architecture (Fig.7) very much follows the Broker pattern. The Broker object in the Broker Pattern is mapped as the ORB Core; it is employed for establishing the connection between a client and a server. The code of the Client-side Proxy of the Broker pattern is reflected as an IDL stub (and DII stub, the code for dynamic invocations), and the code of the Server-side Proxy is reflected in POA (Sect. 3.3.1, IDL skeleton, and DSI skeleton serving for dynamic delivery. More specifically, the direct communication variant of the Broker pattern is employed in many of the CORBA implementation (e.g., in Orbix), where the client stubs directly communicate with a POA. The Interface Repository is used for introspection (particularly in case of a dynamic invocation and dynamic delivery). The Implementation Repository is used for reactivation of servers. The CORBA Lifecycle Service [9] employs the Abstract Factory pattern: In addition to a classical factory, the Lifecycle Service defines also *factory finder* (returns an object factory able to create object of a given type) and *generic factory* (which encapsulates the functionality of multiple factories).

3.3 Providing and employing a service

3.3.1 Registering a service with broker

Registering the object which represents the service is the task of the Object Adapter. Its currently mandatory version, POA, provides, e.g., the `activate_object()`, `activate()` methods to support registration. In principle, `activate_object()` registers the object with POA and assigns a new Object Reference, which will serve for referencing the newly created object; `activate()` signals that a particular server (or a particular POA, to be precise) is ready to accept requests. Finally, calling `ORB::run()` tells

the ORB to enter the main request processing loop.

3.3.2 Naming

Officially there is the CORBA Naming Service which should provide for name spaces and for mappings among Object References and names. However, in addition to the standard Naming Service, most of the CORBA implementations provide a vendor specific way to locate objects associated directly with registering of a service at the server side and a "binding" operation on the client side.

3.3.3 Finding a service, trading

For a client, there are two ways to obtain an Object Reference which identifies the requested service: first, asking a naming service to resolve a known name into the corresponding Object Reference; second, asking the CORBA trading service (in analogy to yellow pages) to provide a list of services (Object References) that possess the properties used as the search key. Some CORBA implementations (e.g. Orbix and VisiBroker), provide a proprietary *location service*. In a sense, this service is a hybrid between naming and trading. In searching for a service, the client provides a partial information on the service (e.g. using "incomplete names", providing the interface the target object should support, etc.); however, the location service does not cover the functionality of the CORBA Naming Service or CORBA Trading Service.

3.3.4 Binding

The client is bound to the requested server after it receives one of its Object References (and a corresponding proxy is created, Section 2.3.4).

3.3.5 Static invocation and delivery

When the client was compiled with knowledge of the requested service IDL specification, the proxy implements the mapped IDL interface methods. The corresponding code of the proxy is encapsulated in the IDL stub (Sect. 3.2). Similarly, the corresponding IDL skeleton is the code of the server-side proxy which implements static delivery.

3.3.6 Dynamic invocation and delivery

For dynamic invocations, the proxy contains the `create_request()` method which returns a `Request` object. There is also a way to submit arguments with the request. When creating a request, the Interface Repository can be consulted for details about the target interface. The actual dynamic invocation is initiated by calling `invoke()`, a method of the `Request` object.

As far as dynamic delivery is concerned, the server-side proxy contains the Dynamic Skeleton code which delivers the request via the `invoke()` method supported by the `PortableServer::DynamicImplementation` interface. CORBA allows for combining static invocation at the client side with dynamic delivery at the server side and vice versa. At the server side, the choice between static and dynamic delivery is made by the server object itself (it can be determined, e.g., as a part of the registration with POA), regardless of the way a request was created, i.e., via a dynamic or static invocation.

A half-way between static and dynamic invocation is dynamic downloading of stubs which can be applied in CORBA clients implemented in Java. In this case, the client code can be compiled without employing the IDL compiler (just the knowledge about the interface form will do).

3.4 Inherent issues

3.4.1 Server objects garbage collection problem

CORBA addresses the garbage collection problem via the third method (with assistance of the clients' code, the server is supposed to keep track of the number of references to a given object provided to the outside of the server). It is assumed that if the client to which the reference was provided does not need it any more, it lets the server know. For this purpose, every `CORBA::Object` (proxy and the implementation object are considered as separate entities - both are `CORBA::Objects`) possesses the `duplicate()` method to be called if new reference to this object is to be created. If the reference is not needed any more, the `release()` method of ORB is to be called. However, most of the CORBA implementations employ slightly modified semantics of these methods: every `CORBA::Object` is associated with its reference counter, `duplicate()` increases and `release()` decreased the counter. If the counter reaches zero the object (proxy or the implementation object) is disposed of.

3.4.2 Persistent remote references

Object references themselves should be persistent in principle. In its IIOP format, an Object Reference contains, in addition to the identification of the target object, the identification of the target machine (hostname:socket) together with the interface type id. An Object Reference R is valid until the server deletes the target object of R. However, especially for effectivity reasons, the run of a server S which has not been answering any request for a time-out period is usually terminated. More specifically, the server's loop inherently contained in this operation is terminated after the timeout period expires. Just before being actually terminated, S can store its state (this also includes some kind externalization of S's objects, e.g., [4]). Thus when a request with R as the target comes via ORB and S has been already terminated, S is typically reactivated, the target of R is recreated, and its state is internalized again. Currently, the reactivation of servers is done in a vendor-dependent way (e.g., Orbix uses *loaders* for this purpose). With introduction of POA, the reactivation of server has been standardized: a *servant manager* supplied with the server assumes the role of reactivating all necessary server objects.

3.4.3 Transactions

There is no implicit support for transactions upon remote objects. On the other hand, the CORBA Transaction Service (OTS), closely cooperating with the CORBA Concurrency Control Service, has been specified. The CORBA implementations supporting OTS include Orbix and VisiBroker. The OTS specification [11] addresses the problem of respecting the partial ordering of resources in the prepare phase by introducing *synchronization objects* which are given a chance to store themselves before the very first prepare action starts.

3.4.4 Concurrency in server objects

In multithreaded servers, standard implementation (language-dependent) synchronization tools are supposed to be employed in server object to avoid race conditions. Note that the CORBA Concurrency Control Service is used at different level of granularity - it provides locking only and was designed in particular to support locking in nested transaction. The other important point is that the implementation of ORB and POA has to reflect the multithreading in servers. As multithreading inherently complicates the implementation, some vendors offer their ORB alternatively in single and multithreaded versions (e.g., ORBIX).

3.4.5 Applying client-server relation transitively, callbacks

Applying client-server relation transitively is straightforward in CORBA. Implementing a callback from a server S to its client C (via the approach mentioned in Section 2.4.5) typically requires the code of C to include also the library code supporting servers. To avoid deadlock when a callback takes place while responding a request from C, the client C has to be of multithreaded implementation, or the callback is to be a one-way (asynchronous) call.

3.5 Reactive programming

Reactive programming in CORBA is supported by the CORBA Event Service [10]. This service provides the *event channel* abstraction. To an event channel, an object can subscribe as an event supplier or event listener. A supplier generates an event which can be associated with event data. Both push and pull models of delivering events to consumers are provided. Thus the CORBA Event Service fully implements the Observer Pattern. A more sophisticated model of reactive programming, the CORBA Component Model, (very similar to Java Beans, Section 4.5) is being prepared by OMG [8].

4 Distributed Objects in Java

4.1 Basic principles

Currently, there are two ways to handle distributed objects in the Java environment: (1) Employing the Java RMI (Remote Method Invocation) service together with the Java Object Serialization service, or (2) employing the CORBA IIOP protocol, alternatively: (a) in a Java-written client code via stubs which speak the CORBA IIOP protocol, or (b) directly in a CORBA ORB fully implemented in the Java environment (e.g., OrbixWeb, VisiBroker)). In this section we limit ourselves to RMI (JDK1.2, [22]).

4.1.1 Request & response

The Java RMI architecture is based on the Broker pattern (Section 2.2.1). In addition to the classical scenario of a client request targeting a particular interface of a single server object, the RMI architecture is open to support also, e.g., replication of server objects (transparent to the client request). At present, the Sun Java RMI implementation includes only simple point-to-point requests and responses based upon the underlying TCP-based streams; we will further limit ourselves to this variant of RMI. In principle, "server" means an object extending the `RemoteServer` class residing in a JVM (Java Virtual Machine) separate (in general) to the client's JVM.

4.1.2 Remote reference

Remote references are handles to *remote objects* (those implementing the `Remote` interface). Remote references are not directly accessible; they are encapsulated in proxies (stubs and skeletons, Sect. 4.2). A key RMI feature is that, from a client's point of view, a remote reference can always be employed only via an interface (derived from `Remote`). This conceptually separates the interfaces and implementations of objects.

4.1.3 IDL Interface

RMI uses the Java language constructs for interface specification (instead of employing a separate IDL language). The rmic compiler generates client stub and server skeleton classes to a particular remote object class from the bytecode.

4.1.4 Proxy: local representative

Anytime a remote reference is provided to a client, a proxy is automatically created (if it does not already exist). In compliance with the Broker pattern, the proxy supports the same remote interface as the target interface. The client always works with a local reference to the proxy. As for proxy creating, the basic strategy (at least in the current Sun Java RMI implementation) is that anytime a remote reference is brought into a Java address space, a new proxy is created; thus more proxies can coexist in one address space even though they embed the same remote reference.

4.1.5 Marshalling: transmitting request & response

RMI uses a proprietary internal format for request and response marshalling. As far as parameter passing in remote calls is concerned, local objects are passed by copy using the Java object serialization (the objects have to support the `Serializable` interface). When a remote object is passed as a parameter (a reference to a proxy is indicated as the parameter) the proxy itself is passed (serialized); this complies with the idea of multiple proxies in one address space (Section 4.1.4). As the proxy contains the remote reference, the RMI specification [22] also says "a remote object is passed by reference".

4.2 Combining basic principles: basic patterns

The RMI architecture is organized in three cooperating layers (bottom-up): transport layer, remote reference layer, and stub/skeleton layer. This very much follows the Broker pattern - in RMI, the Broker object is implemented by the transport and remote reference layers.

4.3 Providing and employing a service

4.3.1 Registering a service with broker

A server object can implement multiple `Remote` interfaces (those inheriting from `Remote`). Not very much reflecting the key RMI policy to strictly separate interfaces and their implementations (Section 4.1.2), a server object is registered together with all the `Remote` interfaces it implements. The actual registration with the RMI Broker (with the remote reference layer) is done via the `exportObject()` method of the `UnicastRemoteObject` class (e.g., it can be inherited into the `Remote` object implementation). The result of the registration is a proxy supporting all of the `Remote` interfaces the object implements.

4.3.2 Naming

At every node supporting RMI, there is the daemon process called RMI Registry. In principle, an RMI Registry is a name server which supports a flat name space and registers pairs <name, proxy> associated with the `Remote` objects existing in this node. A `Remote` object is registered under a chosen name by calling the operation `bind()` (or `rebind()`) of the RMI Registry at a particular node. The RMI Registry is accessible form the outside of the node via its Internet address.

4.3.3 Finding a service, trading

There is no trading or location service in RMI at present.

4.3.4 Binding

To establish a connection from a client C to a server object SO located on a node N and registered there under the name name_SO, C contacts the RMI Registry at N and via its operation `lookup()` resolves name_SO into the registered proxy of SO; a copy of this proxy is delivered to C as the result of the

`lookup()` operation. Moreover, via introspection C can get from the proxy the information on all of the `Remote` interfaces SO (and thus also the proxy) supports. The client can of course receive a proxy as a result of another remote call not targeting a RMI registry. The client is bound to a server object at the moment it receives the proxy.

4.3.5 Static invocation and delivery

If the client was compiled with knowledge of the requested service interface, it can use the methods of the corresponding proxy via statically encoded calls. Similarly, the corresponding skeleton at the server-side implements static delivery.

4.3.6 Dynamic invocation and delivery

There is no support for dynamic delivery on the server side. On the client side, however, dynamic invocation is always possible via the implicit opportunity to use the introspection features provided for any object to dynamically set up a call of a particular method in one of the proxy's interfaces. The key benefit of the Java code mobility is that a proxy code can be always dynamically downloaded into a running application (no recompilation is necessary).

4.4 Inherent issues

4.4.1 Server objects garbage collection problem

The RMI system uses the second approach mentioned in Section 2.4.1: it keeps track of the live TCP/IP connections. Basically, each registration of a remote reference in the RMI Registry implies also one live connection. If the number of live connections reaches zero, the server object is handled as if it was a local object in the server's JVM and thus being a subject of the standard local garbage collection process.

4.4.2 Persistent remote references

Remote references can be persistent in the RMI system in the following sense. If a client uses a remote reference and the connection from its proxy to the target skeleton does not exist any more, the corresponding server object can be reactivated supposing it is *activatable* (via the `exportObject()` method of the `Activatable` class). In the current Sun Java RMI implementation, an activatable object AO can register with the RMID (RMI Daemon) running at its node, and, consequently, a remote reference to AO will contain also an information on the RMID. In case the remote reference is used and the corresponding connection to the target skeleton does not exist any more, the RMID indicated in the reference is contacted instead; the RMID reactivates the server object and provides the client proxy with an updated remote reference.

4.4.3 Transactions

Recently, the Java Transaction Service (JTS) has been announced. Essentially, JTS is a mapping of the CORBA Transaction Service (Sect. 3.4.3) to the Java environment.

4.4.4 Concurrency in server objects

Following the Reactor pattern [15] in principle, the RMI system can deliver multiple calls at the same time to a particular server object and they will executed in separate threads. (Nothing can be assumed about the actual dispatching strategy.) The standard Java synchronization tools are to be employed to handle concurrent access/modification to the server object's state. For this purpose, Java provides, e.g. the `synchronized` clauses, and the `wait`, resp. `notify` methods.

4.4.5 Applying client-server relation transitively

Applying a client/server relation transitively is straightforward in Java RMI by providing the server with a remote reference to another server as described in Section 2.4.5. When a callback from a server S to its client C is implemented this way, deadlock should be avoided by creating a dedicated thread to accept the callback in the client (there is no guarantee as for the number of the threads available for delivering requests, Sect. 4.4.4). As an aside, the Java Beans event model (Sect. 4.5) can also be also employed for callbacks (with a significant comfort).

4.5 Reactive programming

To support the Observer (Publisher-Subscriber) pattern, Java Beans event model is defined in the Java environment. The basic idea behind these abstraction is that a Publisher (a Bean) announces a set of "outgoing" interfaces it will call to report on an event (there are event objects defined for this purpose). A subscriber, *event listener,* implements some of the "outgoing" interfaces and subscribes with the Bean [21] to be called on these interfaces. The Bean notifies all of its subscribers by systematically calling the method, which corresponds to the event being reported, of all these subscribers. The new version of Java Beans, *Glasgow*, supports *BeanContext* services. One the proposed services is the *Infobus* [20]. The basic idea of the Infobus is that, a subscriber does not have to know the particular remote reference of the Bean notifying of the event in which the subscriber is interested. The subscription targets just the common "event bus" and addresses the event directly (not a specific source of the event).

5 Distributed objects in COM/DCOM

5.1 Basic principles

Microsoft's distributed object infrastructure is based on its Distributed Component Object Model (DCOM) [6]. It is an enhancement of the Component Object Model (COM) which is a means for component-based development in the Windows environment. While COM supports component interaction on local machine, both in one address space (process) and across separate address spaces, DCOM provides similar functionality across node boundaries. We focus on COM interaction across separate address spaces and will emphasize some of the key DCOM's architectural enhancements to COM. An important part of Microsoft's object technology is OLE (Object Linking and Embedding); originally designed to support compound documents, OLE is mainly a library of predefined interfaces and default implementation of some of them.

In COM, the key concept is *COM object.* Here "object" is not a classical object, it is a component composed of (classical) objects and providing a set of interfaces to the outside. Using the terminology of Sect. 2, a client residing in one address space calls a remote method of an interface of a COM Object residing in a server (in this paper: *server COM object*, or *server object* for short). Each interface of the server COM object corresponds to one service provided by this object.

5.1.1 Request & response

The COM/DCOM architecture is based on the Broker pattern. In COM, a client issues a request targeting a particular interface of a server object; the server object can reside in the same process as the client does (*inprocess server*), in a separate process (*local server*) but on the same node as the client. DCOM supports clients and servers residing on separate nodes (*remote server*). In both COM and DCOM, remote method calls are synchronous. In COM, request and responses are delivered via the Lightweight Remote Procedure

Calls (LRPC); DCOM uses the Object-Oriented RPCs (ORPCs) developed upon the base of DCE remote procedure calls from OSF.

5.1.2 Remote reference

Remote references are handles to server objects' interfaces. Each of the separate interfaces of a server object is associated with its own remote reference. Remote references are stored in proxy objects. When used for remote calls, remote references are referred to as *binding information*, which can contain, for example, information on the transport protocol, host address, and the port number. When transferred as a parameter of a remote call, a remote reference is referred to as an *Object Reference (OBJREF).* In DCOM. an Object Reference is a tuple <OXID, OID, IPID, binding_info_on_OXID_resolver>. Briefly, OXID (Object Exporter Identifier is identification of the server, OID identifies the target object, IPID identifies the target interface, binding_info_on_OXID_resolver identifies the resolver process which can resolve the rest of the tuple into the corresponding binding information.

5.1.3 IDL Interface

The IDL language used by COM/DCOM is called *MIDL.* Interface specifications are compiled by the standard Microsoft IDL Compiler (also denoted as *MIDL*), which creates the code of sever stubs and client proxies. However, the code of stubs and proxies is generated by other Microsoft compilers as well (e.g., by Visual C++). It should be emphasized that the binary form of generated interfaces (the physical layout) is predefined in COM. In addition, MIDL can generate type libraries (analogous to interface repository entries in CORBA).

5.1.4 Proxy: local representative

Anytime a remote reference is provided to a client, a proxy object is automatically created (if it does not already exist) in the client's process. Naturally, in compliance with the Broker pattern, the proxy object supports the same interface as the target of the remote reference. The client always works with a local reference to the proxy object. The basic philosophy is that anytime a new remote reference is brought into a process, a proxy object is created; in principle, there is a proxy object per each interface of a server object.

5.1.5 Marshalling: transmitting request & response

COM uses a proprietary internal format for request and response marshalling. As far as parameter passing of remote calls is concerned (with DCOM on mind), data are marshalled in a platform independent format called *network data representation.* When a reference to a server object's interface is passed as a parameter (a reference to a proxy object is indicated as the parameter), the corresponding tuple <OXID, OID, IPID, binding_info_on_OXID_resolver> is passed as explained in Section 5.1.2. During this process, the necessary stub and proxy objects (Section 5.2) are always created automatically.

5.2 Combining basic principles: basic patterns

COM/ DCOM closely follows the Broker pattern. The client-side proxy of the Broker pattern is reflected in a proxy object, and the server-side proxy in a stub object. More specifically, the direct communication variant of the Broker pattern is employed, as proxies directly communicate with corresponding stubs. The Abstract Factory pattern plays a very important role in COM/DCOM: In servers, new objects are instantiated with the help of object factories. In a server, each of the server COM object (component) classes is associated with an object factory instance residing in the server which creates new server objects based on this particular class. (To follow the COM convention, we refer to "object factory" as *class factory.*)

5.3 Providing and employing a service

5.3.1 Registering a service with broker

Compared to CORBA and JAVA RMI, the basic philosophy of remote references in COM/DCOM is different: they are not expected to persist; they are not intended to be directly used for reactivation of servers (the persistent information necessary do reactivate a server can be delegated to a moniker - Sects. 5.3.2, 5.4.2). The basic strategy of working with remote objects is that the client asks a remote factory residing also in the server for a remote object creation and at the same time indicates the interface used for accessing the object. During this process the corresponding stub and proxy objects are created. The proxy object contains the linking information to that particular interface of the server object. If the proxy object's reference were not passed as a parameter of another remote call, no registration with "Broker" would be necessary - the linking information in the proxy object would do. However, if such proxy object's reference passing takes place, the corresponding Object Reference has to be passed instead (i.e., <OXID, OID, IPID, binding_info_on_OXID_resolver>). In its target node, this tuple has to be resolved into the linking information. Therefore, a server (object exporter) has to be registered with the OXID Resolver Process running at its node.

5.3.2 Naming

There are two separate name spaces used in COM/DCOM, each of them dedicated to a particular purpose. The first name space is global and contains the GIUDs (globally unique identifiers) used (a) to identify classes of COM objects (these identifiers are called *CLSIDs*) and (b) to identify interfaces of COM objects (these identifiers are *IIDs*). The second, also global name space, *names of monikers*, contains names of monikers (persistent objects used to internalize and externalize the state of COM objects). The names of monikers are based on indication of the URL together with the logical path to data stored on a permanent storage (e.g., a file of or an object in a COM *structured storage*).

5.3.3 Finding a service, trading

There is no trading service available in COM/DCOM. However, there is an option one could call *emulation service*: a class can emulate the functionality of a set of classes being generic or parametrized via an "emulation configuration".

5.3.4 Binding

A client can bind to remote service (an interface of a server COM object) in two ways. First, it can ask a known class factory to create a server object and provide access to its particular interface, reflecting the basic strategy of work with remote objects. The creation is done by calling the `CreateInstance(...,iid,...)` method of the corresponding class factory; `iid` indicates the required interfaces of the created COM object. Second, it can use the method `BindToObject()` of a Moniker associated with externalized state of a remote object. This method either creates a new COM object and internalizes its state, or recognizes that such a COM object already exists. In both cases, the final effect is that the client is provided with a reference to a proxy object supporting the required interface of the server COM object.

5.3.5 Static invocation and delivery

When the client code was compiled with knowledge of the requested service interface, the remote calls can be encoded statically as calls of the proxy object's methods. Recall that in COM/DCOM the granularity of proxies is such that there is a proxy object per each interface of a server object (Section

5.1.4). Similarly, static delivery is done by statically encoded calls executed in the corresponding Stub object..

5.3.6 Dynamic invocation and delivery

As far as dynamic invocation is concerned, COM/DCOM follows very much the approach described in Sect. 2.3.6. A COM object can support an `IDispatch` interface (predefined in OLE), the implementation of which is associated with an "indirect interface" (our concept). As an aside, a description of this interface's methods is available through a type library. `IDispatch` contains the `Invoke()` method; via its parameter of type `DISPID`, a method of the indirect interface can be called (the call is delegated in the implementation of `invoke()`). The `DISPID` value can be obtained from the type library. Similarly, another parameter of `Invoke()`, a variant record, supplies the parameters of the delegated call. The supplied parameters are to be provided in an encoded form (there are rules for encoding both the simple and composite types).

As opposed to CORBA, in COM the client decides if the delivery of a call will be dynamical or statical. However, if the call is to be delivered statically, the server object has to support a "classical" interface; if the call is to be delivered dynamically, the server object has to support an `IDispatch` interface the implementation of which delegates calls to the "classical" interface. However, the server COM object does not have to support both interfaces at the same time. If it supports only the `IDispatch` interface, the client is forced to use only dynamic invocation, even though it knows the "classical" interface at compile time. To achieve flexibility (and to improve efficiency), COM defines *dual interfaces.* In principle, a dual interface combines both the "classical" interface and the corresponding `IDispatch` interface. The dynamic invocation is in OLE referred to as *automation*.

5.4 Inherent issues

5.4.1 Server objects garbage collection problem

The cooperative garbage collection technique is employed in COM/DCOM (Section 2.4.1). The reference counting may be, alternatively, applied on a COM object as one entity, or separately for each of its interfaces. The first variant is the usual approach. However, some of the COM object's interface implementation can be associated with large data structures - in a similar case separate reference counting may be applied in order to dispose the interface implementation as soon as possible.

5.4.2 Persistent remote references

In COM/DCOM remote references are not persistent. The way to approximate persistence of remote references and their target server objects is to associate a server COM object with a moniker object and renew the reference via the moniker's `BindToObject()` method as described in Sect. 5.3.4.

5.4.3 Transactions

Through the Microsoft Transaction Server (MTS), transparent addition of transactional capabilities to a COM object is possible; references to other COM objects are automatically detected and the target COM objects are made transactional components as well (transitive closures of related COM objects are evaluated). Nested transaction are supported. The hart of the MTS, the MTS Executive, maintains a pool of threads to handle requests from clients to (a potentially large) number of transactional COM objects. There is another pool of threads to handle database connections. As an aside, COM structural storage operations have been announced to be subject to simple transactions in Windows NT 5.0.

5.4.4 Concurrency in server objects

In COM, there are two basic models of threading related to access to the objects made externally available by servers (to server objects): the *apartment model* and the *free threading* model. For a server object, being in an apartment means that there is only one thread to call its methods in response to the requests incoming from clients; in other words, the methods of all its interfaces are implicitly mutually excluded and no synchronization is necessary. On the contrary, if the free threading model is applied to the address space a server object resides, the methods of all its interfaces can be called and executed simultaneously by any number of threads. Appropriate synchronization has to be applied in implementation of these methods to handle the concurrent access/modification to the server object's state.

5.4.5 Applying client-server relation transitively

Applying client-server relation transitively is straightforward in COM/DCOM. To avoid deadlock while emulating a callback from a server S to its client C (via the approach mentioned in Sect. 2.4.5), the client C has to be of a multithreaded implementation (the free threading model has to be applied at the client side).

5.5 Reactive programming

To support the Observer (Publisher-Subscriber) pattern, COM/DCOM provides *connection points* and *outgoing interfaces*. The basic idea behind these abstraction is that a Publisher announces a set of interfaces it will call to report on an event (although no event object is defined). Each of these "outgoing" interfaces has to implemented by a separate connection point object. A subscriber, *sink object*, implements some of the outgoing interfaces and subscribes with the particular connection point(s). When asked by Publisher, the connection point notifies all of its subscribers by systematically calling the method that corresponds to the event being reported in all of its subscribers.

To become a publisher, a COM object implements the `IConnectionPointContainer` interface and creates at least one connection point object (which implements the `IConnectionPoint` interface).

6 Conclusion

We have provided an architectural overview of current distributed technologies. In principle, our attempt for architectural analysis, based on a relatively small number of design patterns, has illustrated what is intuitively obvious but hard to grasp when looking on each of the distributed object platforms separately: These platforms are based on the same principles and have to solve similar issues; however their descriptions often speak different languages.

In this overview, we did not target all of the issues related to the distributed object technologies and platforms. Beyond the basic principles, basic patterns, provision and employment of a service, and inherent issues we focused on, attention should also be given to security and benefits of code mobility over the Internet. Both of these areas have become very broad and deserve at least a separate paper.

Acknowledgments

The authors of this paper would like to express their thanks to Dušan Bálek, Radek Janeček, Jaroslav Gergič, and Petr Tůma for valuable comments and for taking part in programming experiments performed in order to make sure on a some of the analyzed platforms' properties not described clearly enough in the available documentation.

References

[1] Buschmann,F., Meunier,R., Rohnert,H., Sommerland,P., Stal,M.: A System of Patterns.Wiley, 1996

[2] Ferreira,P., Shapiro,M.: Larchant - Persistence by Reachability in Distributed Shared Memory through Garbage Collection. In: Proceedings of the 16th International Conference on Distributed Computing systems, Hong Kong, IEEE CS, 1996

[3] Gamma,E., Helm,R., Johnson,R., Vlissides,J.: Design Patterns. Addison-Wesley, 1995

[4] Kleindienst,J., Plášil,F.,Tůma,P.: Lessons Learned from Implementing the CORBA Persistent Object Service. Proceedings of OOPSLA'96. San Jose, CA, ACM Press 1996 (also ACM SIGPLAN 31(10) 1996.

[5] Kleindienst,J., Plášil,F., Tůma,P.: Corba and Object Services. Invited paper, SOFSEM 96, Springer LNCS 1175, 1996.

[6] Microsoft White Paper: Windows DCOM Architecture, 1998

[7] OMG: Object-by-Value RFP, OMG document ORBOS/96-6-14, OMG 1996

[8] OMG: CORBA Components. Joint Initial Submission, OMG 97-11-24, OMG 1997

[9] OMG: CORBAservices Specification, Lifecycle Service, OMG document FORMAL/97-2-11, OMG 1997

[10] OMG: CORBAservices Specification, Event Management Service, OMG document FORMAL/97-2-9, OMG 1997

[11] OMG: CORBAservices Specification, Object Transaction Service, OMG 97-3-4, 1997

[12] OMG:CORBA 2.2 Specification, OMG 98-2-33, OMG, 1998

[13] Orfali,R., Harkey,D.: Client/Server Programming with JAVA and CORBA, John Wiley, 1997

[14] Pree,W.: Framework Patterns. SIGS Books & Multimedia, 1996

[15] Schmidt,D.C.: Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In: J.O.Coplien and D.C.Schmidt, (eds.) Pattern Languages of Program Design Addison-Wesley, 1995

[16] Sessions,R.: Object Persistence, Beyond Object-Oriented Databases, Prentice-Hall, 1996

[17] Shapiro,M.: Structure and Encapsulation in Distributed Systems: The Proxy Principle, Proceedings of the 6-th intl.conference on distributed computer systems, pp. 198-204, Boston, 1986.

[18] Stal,M.: COMunication Everywhere - An overview of Microsoft's DCOM, Object Magazine, 1998,

[19] Stal,M.: Worldwide CORBA: Distributed Objects and the Net, Object magazine, March 1998

[20] Sun Microsystems: InfoBus Specification, August 97, http://java.sun.com/beans/infobus

[21] Sun Microsystems: JavaBeans 1.0, July 97, http://java.sun.com/beans

[22] Sun Microsystems: Java Remote Method Invocation Specification. October 1997, http://java.sun.com/products/JDK/1.1/docs/guide/rmi

[23] Sun Microsystems: A Spring Collection. SunSoft, September 1994

[24] Szyperski,C.: Component Software, Beyond Object-Oriented Programming. Addison Wesley, 1997

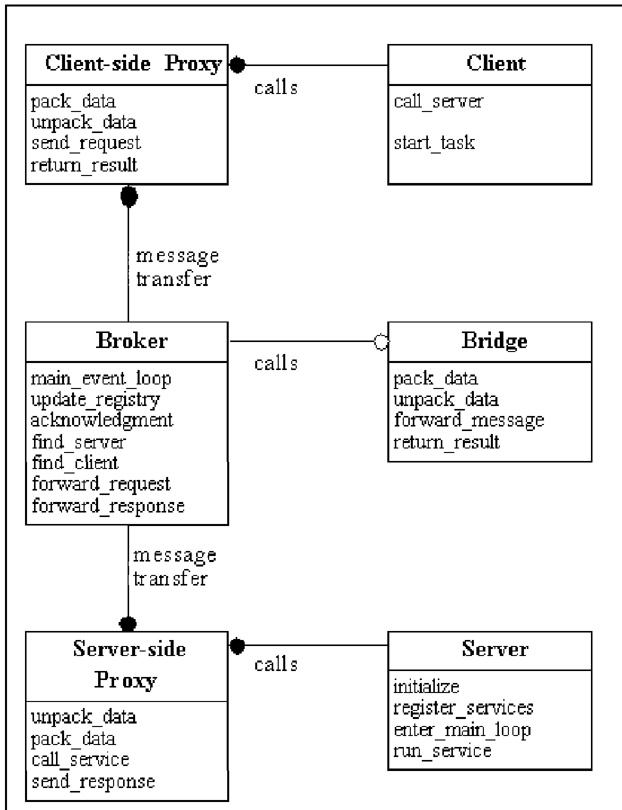

Fig 1 Broker pattern

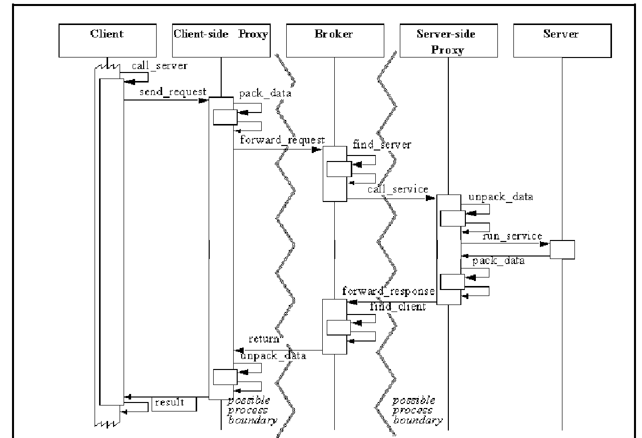

Fig 2 Request delivery scenario

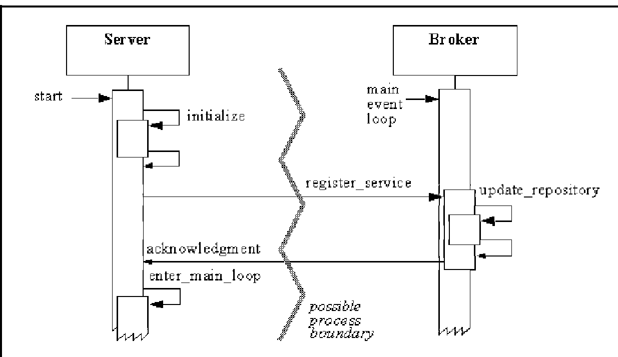

Fig 3 Service registration scenario

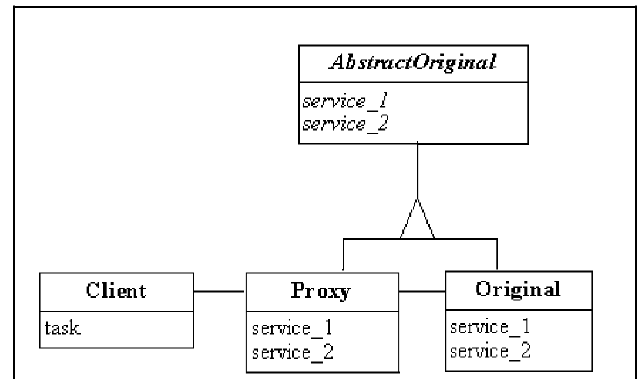

Fig 4 Proxy pattern

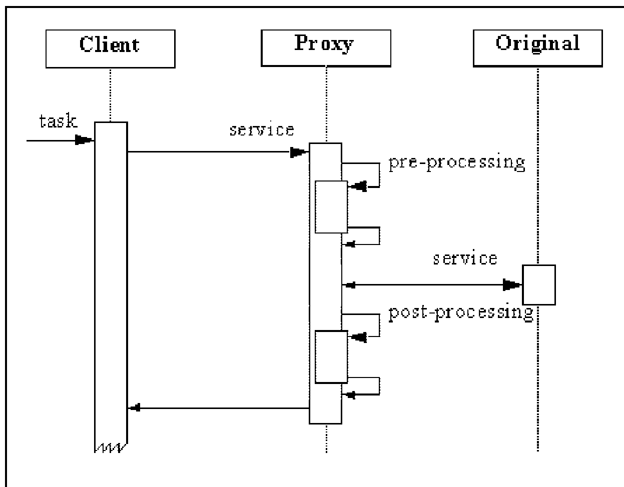

Fig 5 Interaction among objects in the Proxy pattern

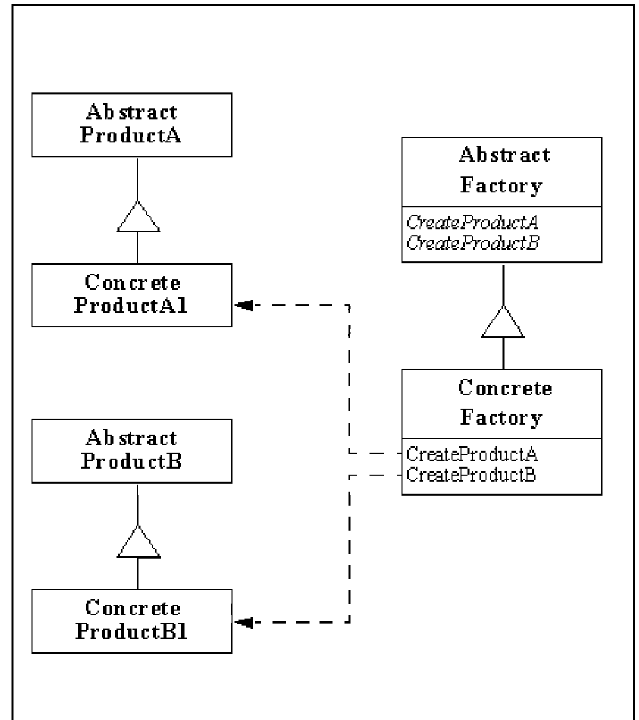

Fig 6 Abstract Factory pattern

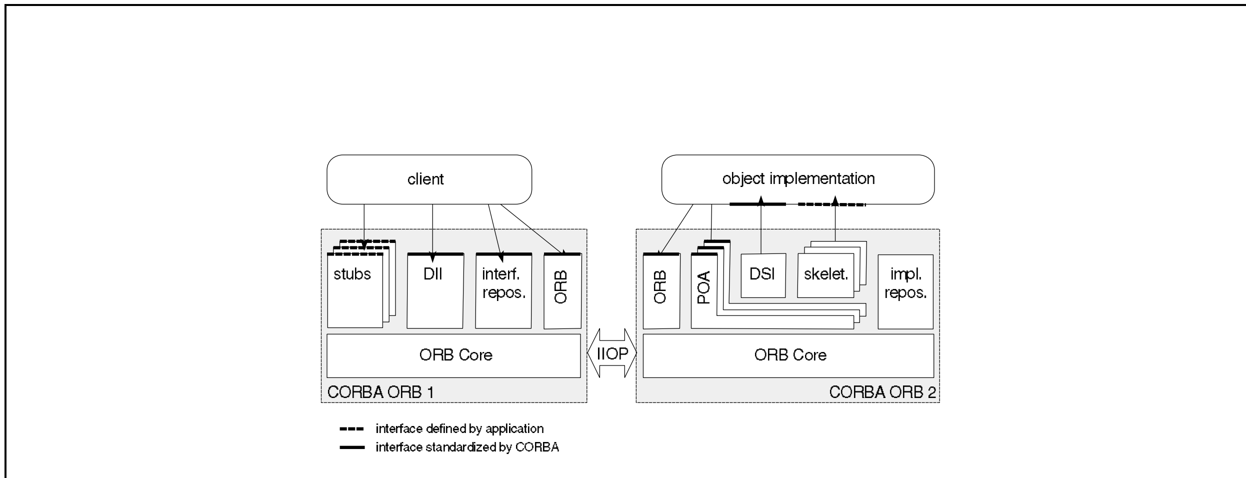

Fig 7 CORBA architecture (stubs are IDL stubs, DII denotes DII stub, skelet. are IDL skeletons, DSI denotes DSI skeleton)