Basis Modeling

Technical Documentation

OMG Concepts - OMA and CORBA 2.0

By the Department
Basis Modeling
For internal use only

SAP AG
Neurottstraße 16
D-69190 Walldorf
Germany

50017941

February 1996

BME022.02

Table of Contents

Table of Figures

The following figures can be found at the end of this report:

Table of Tables

Preface

Audience

This report is intended for those who are interested in the concepts of the *Object Management Architecture* (OMA) and the *Common Object Request Broker Architecture* (CORBA), which have been defined by the Object Managment Group (OMG). Both, OMA and CORBA are not products but specifications. Therefore, this report does neither describe implementation details nor CORBA compliant products currently available on the market.

List of Reports

The following reports have been published by the R/3 Basis Modeling Department:

- Introduction to Concepts of the R/3 Basis System
- Description Method – Modeling Principles and Structure Plans
- Dispatcher and Taskhandler
- ABAP Runtime
- ABAP Generation
- Memory Management
- SAP Data Dictionary
- R/3 Database Interface
- DYNP
- Enqueue Server and Update Task
- R/3 Background Processing
- Change Management and Transport System
- Desktop Integration with RFC API
- Remote Function Call
- R/3 Frontend on Windows NT and Windows 3.1
- R/3 Communication
- Protocol of the Messages between Frontend and Application Server

Overview

The first chapter describes the objectives of the OMG when introducing CORBA and OMA. Chapter 2 deals with a reference model for the Object Management Architecture (OMA). In this reference model, the main components of OMA are discussed. The way of how we have to think in OMG objects is covered in Chapter 3. An object model that defines important terms and concepts is shown. Following this, Chapter 4 provides details of the Common Object Request Broker Architecture and describes the components comprising it. This architecture is part of OMA and basically deals with communication between objects and their implementation. Chapter 5 describes the most important *Object Services.* These services are defined in the *Common Object Services Specification* and add the capability to create, name, store objects etc. to the Common Object Request Broker Architecture. A short introduction to the *interface definition language* is given in Chapter 6. This language is used to define object types independently of their implementation. Finally, Chapter 7 suggests ways for the coupling of R/3 to the OMA object world, more precisely to the CORBA component.

1. Introduction

Client Server Computing

Since the mid 80's, a decreasing use of monolithic systems has been observed. Often, these systems have been replaced by smaller ones that are cooperatively coupled via computer networks. In this network scenario, single machines usually serve specific purposes and provide services for other machines. Therefore, the resource for application development is not only one single computer but a whole set of connected machines. This leads to a different development paradigm – the *client server computing*. **Figure C1** illustrates this application style. The connections symbolized as *req/res* characterize a request/response relationship between client and server.

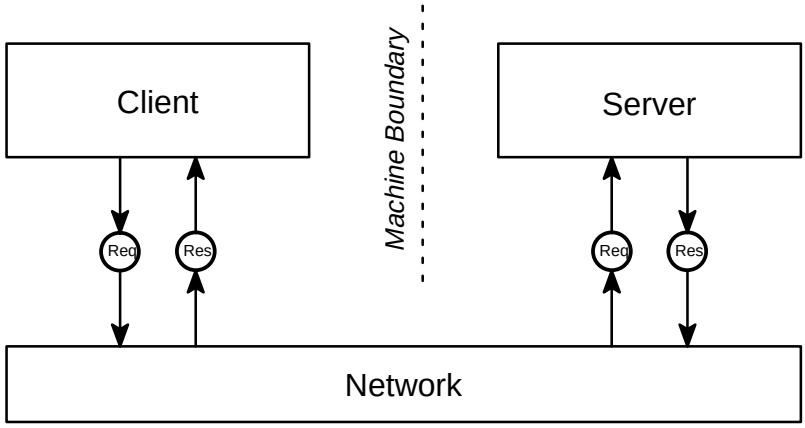


Figure C1 Model for Client Server Applications

Object Orientation

Another technology of increasing interest in the last years is *object orientation*. Traditional analysis, design, and development style separate data from functions. Today object–oriented techniques encapsulate data and the respective accessing functions (often called *methods*) into *objects* which are designated to represent entities from the real world. This approach allows the data of an object to be manipulated from outside only via these accessing methods. The set of methods accessible from outside is sometimes called the *interface* of the object – see **Figure C2**. Because an object can also send requests to other objects, there is a second request/response interface in the block diagram. The *communication agent* is responsible for transporting requests between objects.

Other concepts from the object–oriented world serve the better organization and reuse of code. Objects of the same kind are usually grouped into *classes*. It is possible to establish an *inheritance* relationship between classes. This implies that an object of the inheritee class (also called subclass) can be used in any context that demands an object of the inheritor class (also called superclass). Furthermore, the objects of the inheritee class get all the methods from the inheritor class. This means a very elegant way of avoiding to write common code several times. It is usually also possible to enhance the inherited code.

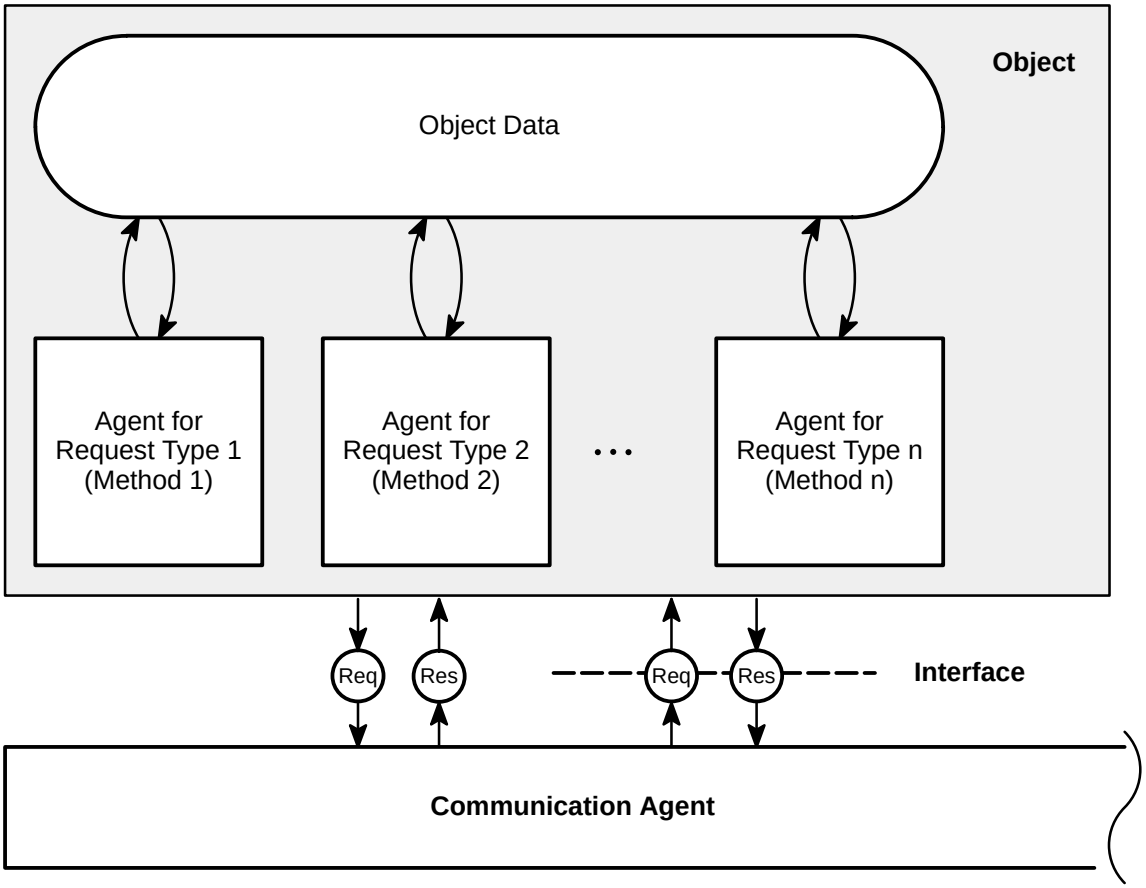


Figure C2 Basic Model for an Object

Two objects may belong to different subclasses of a common superclass. In this case, both objects may have an identical interface, i.e. they can receive the same method calls. Nevertheless, their methods could be implemented with different program code. Therefore, the reaction of an object to a method call may depend on the object's type. This is called *polymorphism.*

Distributed Object Systems

Recently, systems trying to combine the two paradigms have entered the market. One example is the NeXT environment and its *Portable Distributed Objects* layer that hides network distribution aspects and complexity from the application developer. A second example for the mixing of object orientation and distributed computing is Forté which provides a development environment that supports the programming of this kind of applications. Of course, the object communication mechanisms of these systems are proprietary. Furthermore, all objects have to be developed within these systems, urging the use of a certain kind of programming environment and language (Objective–C, C++). Also, the integration of legacy software is hardly supported.

OMG

In the late 80's, several IT enterprises formed an industry consortium called the *Object Management Group*. The objective of this organization is to establish standard interfaces for cooperative software components (i.e. objects). In their *Object Management Architecture*, the OMG describes a framework for the

development, evaluation, and presentation of object technology. The consortium does not develop a detailed specification of a system that implements the concepts of this architecture itself. Instead, it merely describes its ideas (in a rather unprecise way) and lets interested companies work this out by issuing a *Request for Information* or *Request for Proposal*. Then the answers are discussed and the most promising solution is adopted. As a consequence of this time consuming approach, there are some specifications not completely adopted, yet. The whole process is still fluent and improvements in some areas make it necessary to rework other (maybe older) specifications.

OMG Objectives

Among other goals, the OMG especially wants to enable easy development of distributed, object–oriented applications that work in heterogeneous networks. There should be enhanced portability, reusability, and interoperability. The developer is not bound to a certain programming environment, but can use the most appropriate one. Also she or he should be able to work with the object system of any vendor, always being able to change this decision.

The software that the OMG expects to be built for these distributed environments is intended to follow the idea of *componentware*. Small packages of functionality will be grouped together to applications according to the users desire. Of course, this would lead to a substitution of the current monolithic applications and give users more flexibility in choosing their favourite software configuration.

2. Reference Model for OMA

The *Reference Model for the OMG's Object Management Architecture* (OMA) basically describes interacting objects with certain capabilities and a mechanism these objects can use to cooperate. It identifies interfaces and protocols to standardize object access. Only basic statements, not detailed specifications are provided at this point. Details are provided in the descriptions of the components of OMA.

OMA Components

Figure 1 shows the OMA components. The *Object Request Broker* (ORB) provides the mechanism for the transparent issuing/receiving of object requests. The objects that use this service can be divided into three groups: *Object Services, Common Facilities, and Application Objects.*

Object Services

The Object Services provide basic services that are necessary for developing useful distributed applications. These services are specified in detail in the *Common Object Services Specification* (COSS). The specified object services provide a very basic functionality and OMG expects them to be available on every OMA–compliant system.

COSS 1

The COSS 1 contains four services. The most important ones might be the *naming* service that is responsible for the administration of names in an environment and mapping names to objects and the *lifecycle* service that describes how objects can be created, deleted, copied, or moved in a distributed environment. Another service handling the persistent storing of objects is called *persistence* service. The last service in COSS 1 is the *event notification* service that allows objects to register any kind of event or react on its occurrence. These services are presented in some more detail in Chapter 5.

COSS 2

The recently adopted COSS 2 adds some more useful utilities to this set. Concurrent access to objects is important in a scenario of multiple distributed objects. Support for this functionality is provided by the *concurrency control* service. A mechanism to write objects to a stream (file, memory, network) is defined by the *externalization* service. It also covers the opposite direction, the internalization and restoration of externalized objects. The *relationships* service deals with creation, deletion, navigation, and management of relationships between objects. The last COSS 2 service is the *object transaction* service which supports atomic execution of one or more operations on one or more target objects. The ACID (Atomicity, Consistency, Isolation, Durability) principle is guaranteed. This is achieved via a 2–phase–commit mechanism for all involved objects.

For some object services, a client can simply issue a request to get the work done. Most services, however, make it necessary that certain (predefined) interfaces must be implemented in the clients that want to use them. For example, an object that is intended to be lifecycle–capable (i.e. it can be moved,

copied, removed) must carry the interface *LifeCycleObject*. Therefore, its interface will be derived from that interface and its coding has to contain the appropriate functionality.

Common Facilities

While the object services provide very basic, low–level functionality that is expected to be used by almost every application, the semantics of the *Common Facilities* are on a higher level and they address a smaller range of clients. Services that are useful for many applications will be established here so that the same functionality does not have to be implemented by each application but is developed once and can be called everywhere. Like the Object Services, the interfaces to the Common Facilities are specified by the OMG. But in contrast to Object Services, these facilities need not be available on every platform. Examples for this kind of utilities are a mail service or a hardware–independent print service.

Application Objects

The last category of objects shown in Figure 1 are the *Application Objects*. An application that is OMA–compliant comprises a set of cooperating classes and objects communication via ORB. They provide yet a higher level of objects than the Common Facilities and are dedicated to certain application domains, e.g. CAD, business applications, network management tools, office software, etc. Originally, the OMG did not plan to standardize and adopt these object types.

Hierarchy

The three discussed categories of objects are on different layers of complexity, as illustrated in **Figure C3**. The objects of a more complex layer use functionality from those of a lower layer. Additionally, objects of one layer might also use functionality of objects belonging to their own layer. This is especially the case for the object services category. For higher–level objects, it will rarely be neccessary.

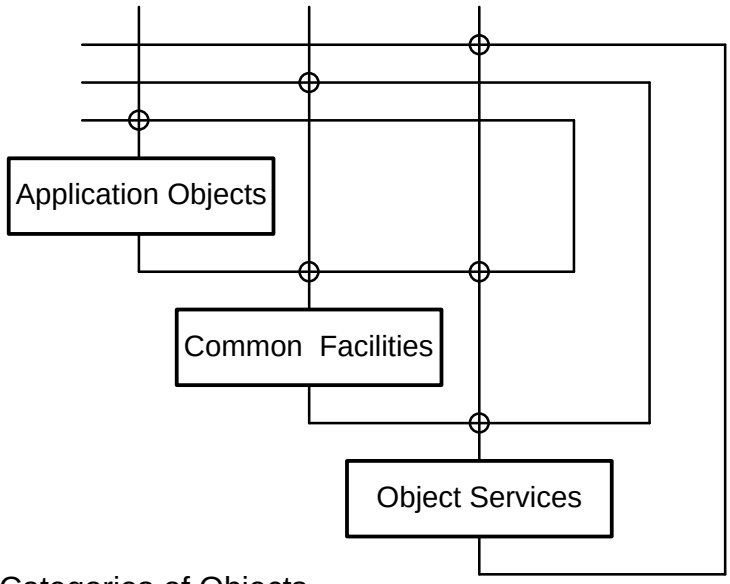


Figure C3 Categories of Objects

3. Object Model

Core Object Model

In addition to the reference model shown in Chapter 2, the Object Management Architecture defines an object model, called *Core Object Model*. It defines a set of terms and concepts describing what an object is, its structure, and properties, and how objects can be organized in an efficient manner. **Figure 2** illustrates some of the terms in an E/R diagram.

Object Reference

In the lower left, we find that *values* are devided into values which identify objects (i.e. *Object References*) and other values (*Non–objects*). Because the OMG object world is distributed and a transport of objects for every request issued would be very expensive in terms of performance, the model deals with references instead of the objects themselves. Every reference uniquely identifies an object. And of course, non–object values can be handled directly without the necessity of references.

Object and Non–object Types

The description of these values is given in the middle layer where all types can be partitioned (as with the objects) into *Object Types* and *Non–object Types*. The latter are not defined in the Core Object Model, but are intended to be specified in the components that detail OMA. In this context, ”component” means a Core Object Model–compliant enhancement to the OMA. In CORBA, for example, which will be discussed in Chapter 4, non–object types comprise basic data types (like integer, float, char), and constructed types (e.g. array, sequence, struct).

Sub/Super–typing

Figure 2 also shows that there is a relation between object types, but not between non–object types. This reflects the fact that the latter ones cannot be constructed by the application developer. They are predefined in the environment she or he is working with. Of course, object types can be defined as needed and new ones can be set into a relationship with existing ones. This relationship is called *Sub/Supertyping* and the semantics is as follows: a ”type S is a subtype of a type T” means that an object reference for an object of type S can be used in any context where a reference to an object of type T is appropriate. T is then called a supertype of S. In this model several supertypes are allowed for a single object type and there is one special object type that is not subtype of any other type but has all other types as descendants in the sub/supertype hierarchy, i.e. it forms the root of the type hierarchy.

Interface Definition Language

While the left side of Figure 2 shows the core concepts, the right side depicts the descriptional side of the object model, i.e. it deals with how to describe the object types. An object type is directly related to an *interface*. The definition of such an interface is given in textual form in a language called the *Interface Definition Language* (IDL). Again, this is not part of the Core Object Model, but specified in additional OMA components which describe what has to be expressible in that language. The most important items are *Attributes* and *Operations*. The former act as data storage within objects, while the latter

describe an object's functionality. In the signature of an operation, it must be possible to specify (input and output) parameters and a result type. Further specifications can be demanded in refining components like CORBA. There, for example, also a call context and means for exception (error) handling are defined. (A call context can be seen as a special request parameter sent to the server object.)

Sub/Supertype vs. Inheritance

The relationship between the sub/supertyping and inheritance relationships has the following meaning: Whenever a type S is a subtype of a type T, then the interface of S inherits all operations from the interface of T. In this way, the somehow artificial separation of both concepts is tied together again in the end. Therefore, the fact that a type can have several supertypes can also be described as: there is *multiple inheritance* between the interfaces.

Dynamics

The diagram does not show the dynamic aspects of the object model. Dynamic interaction occurs if a client issues a request. Such a request must identify a server object that has an appropriate operation and can act as a server for the request. These aspects will be examined in more detail in Chapter 4.

CORBA Object Model

In addition to the model described above, CORBA defines its own object model which is based on the OMA object model with several compatible enhancements. Among these are a detailed specification of the non–object data types, exceptions, request contexts, and a formal specification of an interface definition language. Additionally, the separation between object type and interface is dropped, i.e. the sub/supertype relationship and the inheritance relationship are unified. More details can be found in the OMG specifications.

Figure 3 illustrates the term "CORBA Object" by means of a block diagram. The figure shows, that there are the Constants as the part of the object's data which can only be *read* by others or the object itself. The rest of the data consists of Attributes and the part of object data that depends on the object's implementation. The Attributes can be accessed via implicitly defined "get"– and "set"–operations. ("set"–operations are not defined for read–only–Attributes.)

4. Common Object Request Broker Architecture

The *Common Object Request Broker Architecture* (CORBA) describes a framework for the implementation of an Object Request Broker and its components that provide the service of transparent requests between distributed objects. As we have mentioned in the introduction of this report, distributed object systems can be seen as a ”fusion” of client/server computing and object orientation. CORBA can be seen as such a fusion, too.

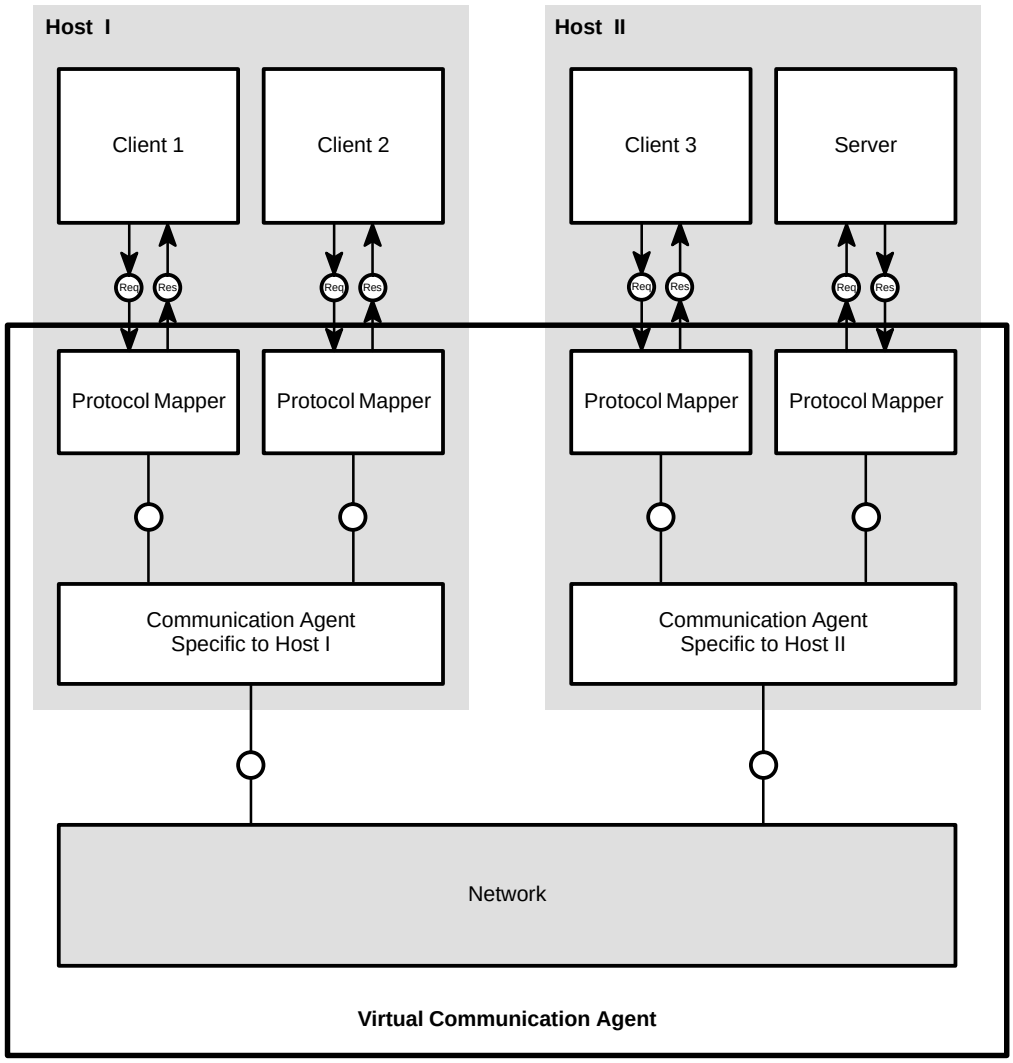


Figure C4 Structure of a Distributed Application

Figure C4 shows the inner structure of a distributed client/server application in a heterogeneous environment. The application consists of three clients and a server which communicate via the *virtual communication agent. Client 1* and *client 2* are running on one machine *(host I). Client 3* and the *server* are running

on another machine *(host II)*. On each host, there is a *communication agent* which uses the communication services of the local operating system. The *network* allows communication across machine boundaries. On the assumption that clients and servers are built using different programming languages there is the need for *protocol mappers*. These offer easy access to the local communication agent via interfaces that are specific to the developer's programming language and hide details of the local operating system. In practice, a protocol mapper is generated by compiling a description of the interface the server provides. (In the OSF's *Distributed Computing Environment* for example, the protocol mappers are called "stubs".)

Figure C5 shows the structure of an application developed using an object oriented programming language. Aside from the *objects* there is an *agent for communication and creation/destruction of objects.* Objects can use this component to send requests (method calls) to other objects or to initiate the creation and destruction of objects. This agent is *specific to the programming language.* It is often useful to show this agent in a model even if it doesn't appear as an object in terms of the programming language.

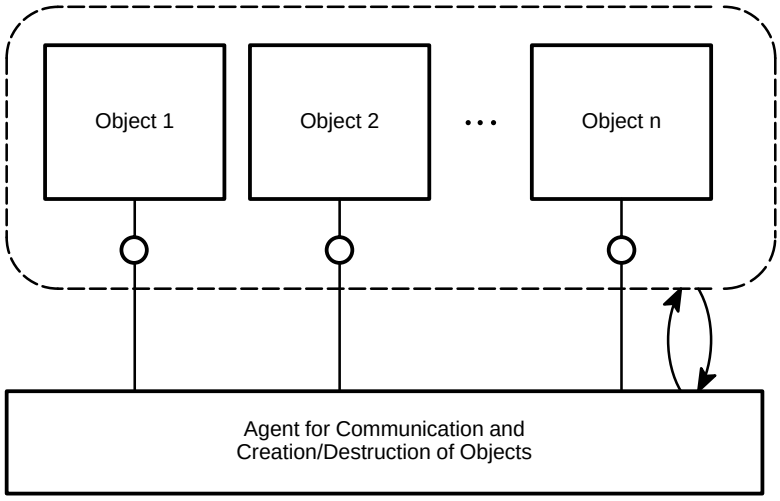


Figure C5 Structure of an Object Oriented Application

4.1 Object Request Broker

The fusion of both structures described above leads to a first, simplified model of a CORBA application as depicted in **Figure 4**. The structure type of Figure C5 can be found in the areas called *application subsystem A, B and C*. These subsystems represent parts of a CORBA application written in *different* (usually object oriented) *programming languages.* The coupling of the application subsystems is provided by the *Object Request Broker.* Therefore, the inner

structure of the Object Request Broker is very similar to the structure of the virtual communication agent in Figure C4. The Object Request Broker and its parts will be described in detail in the following subsections.

4.1.1 Object Request Broker Core

Request Transmission

The central component of the Object Request Broker shown in Figure 4 is the *Object Request Broker Core* (ORB Core), which transports requests and their results among application subsystems. Operating system services and the underlying network are used for this purpose.

Synchronous Request

The definition of an operation via CORBA IDL may use one of two execution semantics. The distinction is made with the optional keyword *oneway.* If it is missing, the operation will be handled in a *synchronous* way, i.e. the client object waits for the request result being returned. In this case, the requested operation will have been executed exactly once if it returns successfully and at most once if an error occurred during request delivery or the operation itself.

Asynchronous Request

If the keyword *oneway* is specified, the operation will be performed *asynchronously,* i.e. no result will be returned and the client never synchronizes with the completion, of the request. The two cases are illustrated in **Figure C6.**

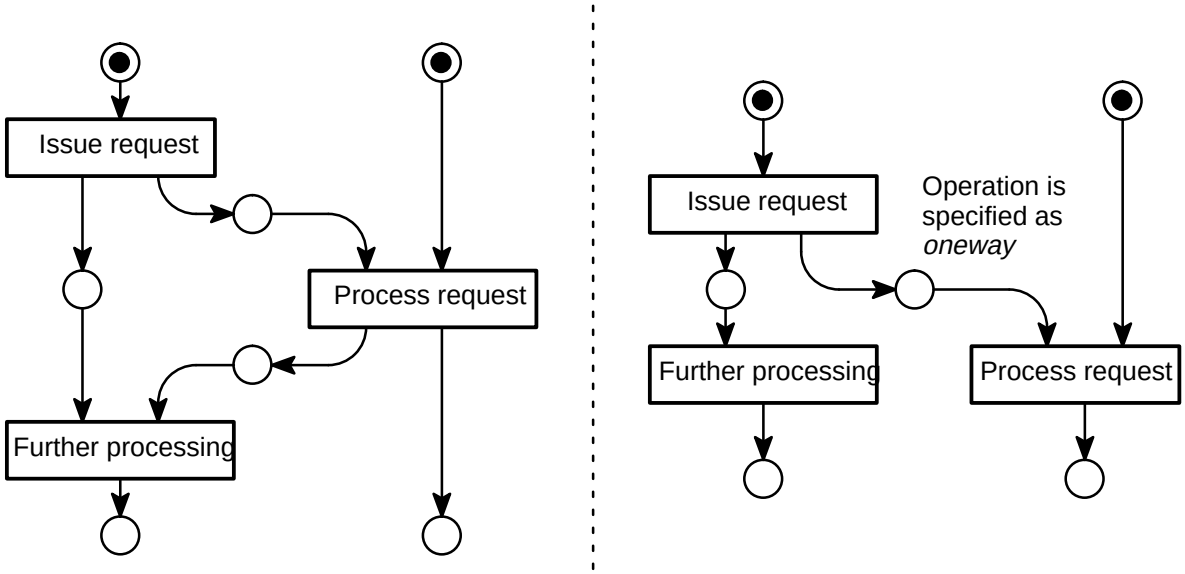


Figure C6 Synchronous vs. Asynchronous Request

Request Information

Each request sent to the ORB Core must contain the following information:

- *Object Reference* of the server object:
  For each Object that can be accessed via the ORB, there is a unique object reference known to the ORB Core. Because the ORB Core has additional information about the physical location of the server object, the request can be delivered correctly.
- *Request Type:*
  The request type defines the type of operation the server object will perform.

- *Request Parameters or Arguments:*
  The number of request parameters depends on the request type. There are three kinds of request parameters:

  – *in parameters* are values transported from the client object to the server object. An in parameter can also be a reference (pointer) to a storage place that contains the value to be added to the request information. This storage place is accessible for the client object.

  – *out parameters* are values transported from the server object to the client object as an outcome of a requested operation. More precisely, an out parameter is a reference to a storage place the returned value is written to. Again, this storage place is accessible for the client object.

  – *inout parameters* denote both, a value to be sent to the server and a value to be returned to the client.

- *Context Information:*
  The so–called context information can be seen as an additional request parameter – more details can be found in the CORBA specification.

Response Normally, the result of a request is returned by the ORB Core to the client object. The result consists of a return value and the out parameter values (if defined). If the requested operation could not be performed or the request could not be delivered, a so–called *exception* is returned to the client object. An exception can be seen as a special kind of result which contains information about the failure type.

4.1.2 Stubs and Skeletons

The connection of application parts to the ORB Core is given by the *stubs* and *skeletons*. These are generated by compiling an IDL interface description of the interface a server object provides. IDL is part of the CORBA specification and precisely defined there. This language and the process of creating stubs and skeletons are examined in Chapter 6 in some more detail.

This section will explain the functionality of stubs and skeletons by analyzing the possible communication processes between the *application objects A1, A2, A3, B, and C* depicted in Figure 4.

Client Object Object A1 is a client object because it only *sends* requests to other objects but does not provide services to other objects.

Server Object Objects which only *receive* requests from other objects and provide services are server objects. In the block diagram there are two server objects, A3 and C.

Of course, an object can act as both, a client and a server. Objects B and A2 are examples for this type of object. The layered structure diagram in **Figure C7** illustrates the client/server relationship of the objects.

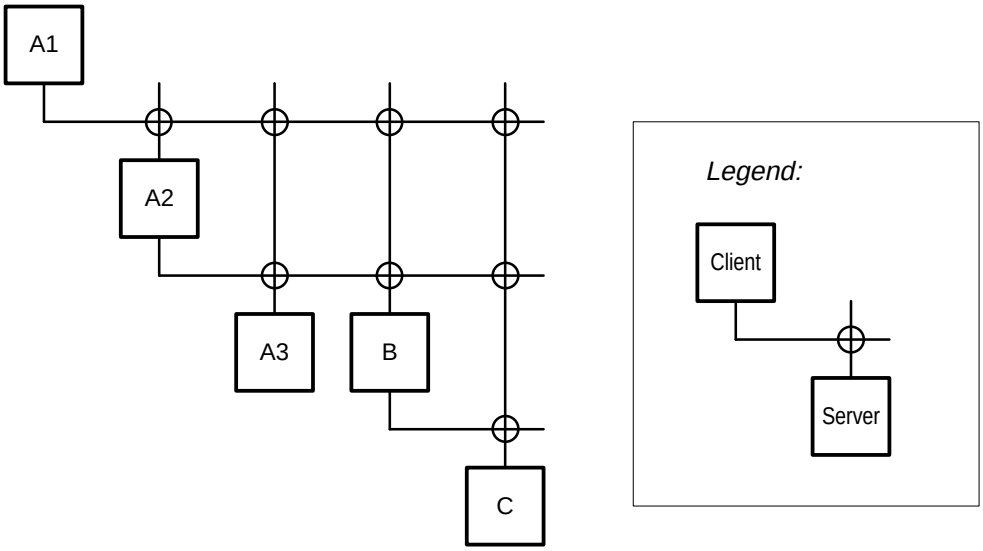


Figure C7 Client/Server Relationship of Objects

CORBA Objects vs. non–CORBA Objects

Objects which belong to the same application subsystem can communicate without the ORB being involved, i.e. they use their local communication agent. Only *server* objects which are addressed *via the ORB* are represented by an object reference inside the ORB Core – these are the CORBA Objects. Others – non–CORBA Objects – are unknown to the ORB Core, see also **Table 1**.

Object is	CORBA Object	non–CORBA Object
Client	–	A1
Server	C	A3
Both	B	A2

Table 1 CORBA Objects vs. non–CORBA–Objects

4.1.2.1 Stubs

If an object (A1, for example) wants to send a request via the ORB Core, it can't address the server object (C, for example) directly, because it is unknown to the local communication agent. On the other hand, sending requests to the ORB

Proxy Objects

Marshalling

Core would force an application developer to deal with the details of the ORB implementation. The solution for the problem is a *proxy object.* This object receives the request instead of the server object and delivers it to the ORB Core – hiding ORB implementation details. A proxy object is responsible for marshalling the request and its parameters (i.e. putting them in an ORB–compliant format and handing this together with the appropriate object reference to the ORB Core.) It is also responsible for receiving the return value and output parameters of the request and passing them back to the caller. In the CORBA terminology, the program code of the proxy objects is also called stub.

4.1.2.2 Skeletons

In the server object's application subsystem, the skeleton is the corresponding part to the stub. A request sent to a skeleton by the ORB Core is delivered to the right server object. When the requested operation is finished, the skeleton marshals the result and output parameters back to the ORB Core. Similar to the stubs, a skeleton hides the details of the ORB implementation.

Sharing of Server Objects

If a client object wants to communicate with a server object via the ORB Core, there must be at least one proxy object within the application subsystem. Objects of a single subsystem can share a proxy object to request services from the same server object. (An example for this is given by the objects A1 and A2 which are both clients of object C)

Multiple Proxy Objects

Because client objects of the same server object can be part of different subsystems, there has to be a proxy object in each subsystem. (In the example scenario, there is a C proxy object in subsystems A and B.)

Language Mapping

Both, stubs and skeletons depend on the language both client and server objects are implemented in. The CORBA 1.2 specification defines a mapping of the CORBA IDL to the C programming language. This *Language Mapping* gives an application developer the means to access CORBA objects, issue requests, get information from the runtime environment, etc. Some more remarks about this can be found in Chapter 6. Other language mappings are adopted by the OMG, e.g. a SMALLTALK and a C++ mapping.

A mapping to a language that is *not object–oriented* (for example C) seems to be contradictory to the model presented above. Nevertheless, the model has to be modified only in some details:

If subsystem A would be written in the programming language C, the stubs corresponding to objects C and B do not represent proxy objects but are simply a set of procedures for each (server) object type. An operation that can be requested from an object is mapped to a procedure that can be called. By the execution of the stub code *this procedure call is converted into a request to a CORBA object* that is delivered via the ORB Core. For the ORB Core, there is

no difference between the requests received from a proxy object and those received via a stub procedure.

If subsystem C would be written in the programming language C, the skeleton would convert incoming requests to procedure calls – therefore, the implementation of ”object C” has to be done by writing a procedure for each request type that can be received by the skeleton. For the ORB Core, there still appears to be a server object.

Dynamic Skeleton

As an addition to CORBA 1.2, the new CORBA 2.0 specification defines another skeleton type called ”Dynamic Skeleton”. It will be discussed in context with interoperability issues, see Chapter 4.5.

4.2 ORB Interface

According to the block diagram in Figure 4, the ORB Core’s functionality is to transmit requests. In fact, there are a few additional services the ORB Core provides to objects of all types. An important one is the mapping of object references to strings and vice versa. Because an object reference is some kind of opaque handle and every CORBA implementor may choose a representation that suits best for her or his purposes, such a reference cannot be exchanged to another application or be written on a file for later reuse. Thus, creating a string from it gives the possibility to exactly do that. On the other hand, having a reference string allows a client object to create a proxy object. By doing this, it becomes connected to a server object.

Object Reference Conversion

ORB Proxy Object

One means for accessing the ORB Core is an *ORB proxy object* as shown in **Figure 5**. Both, object A and B can send requests to their local ORB proxy object. These requests are handled by the ORB Core itself and are *not* transmitted to some server object. Besides the conversion of object references there are some other services provided via ORB proxy objects. More details can be read in the CORBA specification.

Some of the requests which can be sent to *any proxy object* are *not* transmitted to a server object – they are handled by the ORB and/or the proxy object. The operations belonging to these requests are defined for the supertype of all proxy objects:

- Duplicating and releasing of (proxy) object references.
  A client object can duplicate and release references of proxy objects by sending special requests to the proxy object. This gives the opportunity for counting the client objects which need a certain proxy object. When all client objects have released their references, i.e. ”no one needs the proxy object any longer”, it will be deleted.
- Checking for the existence of a server object.
  A client object can ask if there is a server object belonging to a proxy object.

- Creation of a (local) ”request object”.
  (This will be discussed in Chapter 4.4.)
- Determining the type of a server object. (see below.)
- Determining details of a server object’s implementation. (see below.)

Interface Repository

There are two important parts of a CORBA implementation shown in the lower part of Figure 5. One is called the *Interface Repository.* It is the central place for storing information about the types of CORBA objects. A client object can ask a proxy object to return a description of the server object’s type. This means that the type information about any CORBA object is *available to a client object at runtime.* More details about this topic can be found in Chapter 4.7.

Implementation Repository

Analogous to a repository containing information about types of objects, there is also one that contains information about the *implementation details* of server objects, called the *Implementation Repository.* A client object can ask a proxy object to return a description of the server object’s implementation. This means that the implementation information about any CORBA object is *available to a client object at runtime* (in addition to type information). The Implementation Repository contains information that allows the ORB to create objects. Among the provided information is the location of the server object’s program code, for example.
In addition, debugging information, information about resource allocation, security, etc. could be placed in the Implementation Repository. The whole content of the Implementation Repository is outside the scope of the CORBA specification because it depends on the ORB implementation, the underlying operating system etc. .

ORB Interface

The functionality of proxy objects (as described above) and of ORB proxy objects is also called the *ORB Interface.*

4.3 Interface Repository

Interface Repository Information Structure

As already mentioned, one function of the Interface Repopsitory is to provide object type information at runtime. This could be useful if the type information of a server object was not avilable at the time the client object’s program has been written. In this case, it must be determinable which operations can be performed by a certain server object, which exceptions can be raised by an object of that type, and so on. To provide this information online in a CORBA environment, the Interface Repository has been introduced by the OMG. It maintains information like object type (”interface”) definitions, operation definitions, exception definitions, etc. *according to the scope of the Interface Definition Language IDL*. The hierarchical structure of those definitions is illustrated in **Figure C8**. The root of the tree is a so–called *Repository* which serves as a container for all available definitions. The various layers have to be

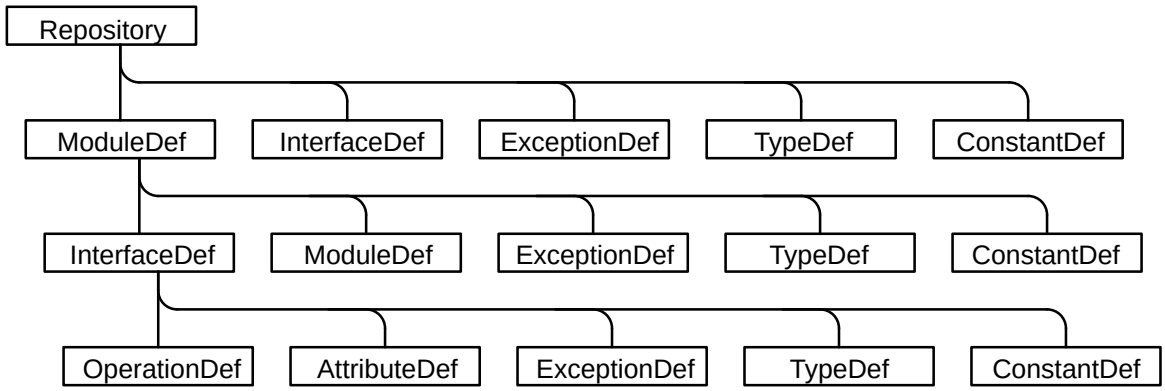


Figure C8 Hierarchy of Definitions in the Interface Repository

read as follows: Every definition in a layer beneath another layer can be contained in the definition of the upper layer to which it is connected. This enables navigation through the whole information tree. The interested reader may find more about this topic in the CORBA specification.

The Interface Repository is also useful to the ORB during request delivery because it can be checked if a requested operation is defined for an addressed server object.

Like the Implementation Repository, the Interface Repository may contain additional information that could be useful in a certain CORBA system, such as information for CASE tools like class browsers and so on.

4.4 Dynamic Invocation

There are two situations in which sending a request *via a proxy object* is not possible (or very awkward):

- Situation 1: *The proxy object does not provide all operations the server object can perform.*

  This situation can occur, if the *type of the server object* (object B) is *not known exactly or totally unknown* at the moment the program code of the client object (object A) is written.
  For example, only an IDL description of a *super*type of the server object could be available to the implementor of the client object. In this case, some of the operation types are not included in the IDL description. The IDL compiler will only generate stub code for these operation types. Due to this, *the B proxy object will not be able to pass all types of requests.*

- Situation 2: *The client object must not be blocked when waiting for the result of a requested operation.*

  While sending requests via a proxy object, the client object is either blocked until the result is available or it does not receive a result at all (see also Chapter 4.1.1.).
  This means, an alternative way to issue a request is neccessary to *avoid the blocking of the client object during the server object's activity.* (A thread mechanism in the client object implementation as an alternative is not discussed here.)

Request Object

In these situations, sending requests is done via *Request Objects* – see **Figure 6**. A request object is an alternative means to send a request and return its result.

Dynamic Invocation

The process of preparing and issuing a request via a request object is called *Dynamic Invocation.* An overview of this process is depicted in **Figure 7**.

At the beginning, object A must know which type of requests the server object can perform. If this information is not available to the client object (first situation – see above), it must ask the Implementation Repository to return a type description of object B.

Creation of a Request Object

The first step of the dynamic invocation process is the *creation of a request object* (”create_request”). This is done by the proxy object and/or the ORB Core (see Chapter 4.2). There are two types of creation the client can request from the proxy object:

- *Completely defined request:*
  The client object tells the proxy object the intended *request type* and all *arguments* needed to specify the operation. The proxy object creates the request object.
- *Not completely defined request:*
  If the client object handles only the request type, but *no arguments* to the proxy object, the request object will also be generated.
  Unlike the process described first, arguments have to be passed *to the request object after its creation* (”add_arg”). This must be done until all arguments are stored in the local *request description* of the request object (see Figure 6).

Request Description

Because the reference of the server object has been stored in the request description during the request object's creation, the request description is complete. The request object is ready to send the request – now being defined by the request description – via the ORB Core. *The proxy object is not involved in delivering the request.*

”Synchronous” and ”Deferred Synchronous” Request

There are two different ways to deliver a request via a request object, called *Synchronous Request* and *Deferred Synchronous Request.* These alternatives

are indicated by the two grey shaded subnets in the lower part of Figure 7. The following two subsections will describe both alternatives in more detail.

4.4.1 Synchronous Requests

Blocked Client Object

The first way to issue a request via Dynamic Invocation is called *"Synchronous Request"*. **Figure 8** shows this process in detail. The principle is simple, because the request object passes the request and its result just like a proxy object ("invoke"). This means that the client object is *blocked for the time of the requested operation.*

This kind of request delivery is useful only in situation 1 (as described above) – the only advantage is, that the request type can be determined at runtime.

4.4.2 Deferred Synchronous Requests

Client Object not Blocked

The second kind of request delivery is more complicated but the client object is *not blocked for the time of the requested operation.*

Figure 9 shows the delivery process in detail. There are three types of requests the client object (object A) can send to the request object:

- At the beginning, object A tells the request object to deliver the request ("send"). This *starts* the requested operation (transition "method execution"), but does not block the client object.
  After starting the service operation, the client object can do some other actions (transition "other activity").
- If the client object has finished all other activities – that means, it may be blocked *now* – it tells the request object to return the result ("get_response (wait)" – see lower left part of the petri net.). If the result is not yet available, the client object is blocked.
- If the client object needs to know whether the service operation has finished and *blocking is still forbidden,* it tells the request object to return the result, *if available* ("get_response (no wait)"). If the result is available, it will be returned to the client object – otherwise the answer "request not finished" is returned. The client object remains active in both cases.
  The loop in the middle left of the petrinet shows that the client object can repeatedly ask for a result until it is available.
  On the other hand, the client object can decide to wait for a result at the end. (This means leaving the loop before the request is finished.)

Dynamic Invocation Interface

The functionality of proxy objects (as described above) and of request objects is also called the *Dynamic Invocation Interface*. Choosing Dynamic Invocation

instead of proxy objects makes a client more flexible, but causes some more effort during implementation.

4.5 Interoperability of CORBA Systems

Distributed ORB

Taking a look at the CORBA system shown in Figure 4, one can see a single ORB that uses a network to transmit object requests and results across machine boundaries. On the assumption that this ”distributed ORB” is built by a single vendor, there is no need for a standardized network protocol.

Cooperating ORBs

If there are two CORBA systems which have to be connected to a single CORBA system, it is obvious to connect the ORBs. Both ORBs could *cooperate* in transmitting requests and results between the two CORBA systems, i.e. they act as a single ORB. If the CORBA systems are implemented in *different* ways, there must be a *standardized protocol* for the inter–ORB communication. Within the CORBA 1.2 specification, there is no such protocol defined as being mandatory for a CORBA implementation. Therefore, the first implemented ORBs were often unable to cooperate.

CORBA 2.0

To solve this problem, the OMG has adopted interoperability standards within the CORBA 2.0 specification. This Chapter presents an overview of these additions to CORBA 1.2.

4.5.1 Domains

In terms of CORBA 2.0, interoperability means the possibility to transport requests and results across *domain* boundaries. Domains of a CORBA system are subsystems which have different technological or administrative characteristics – here are some examples:

Domain Types

- *referencing domain* – the scope of a CORBA object reference.
- *representation domain* – the scope of a message transfer syntax and protocol at ORB level.
- *security domain* – the extent of a particular security policy.

There can be many different domain types, depending on the needs of different users or system administrators. Due to this, bridging may be more than just passing requests and responses. For example, authentication has to be done at the moment a request is sent to a secure domain. When two referencing domains are connected, CORBA object references must be transformed into the right format etc. ...

Inter–ORB Protocol

Figure C9 shows an example CORBA system which consists of two domains – A and B – where each domain has its own implementation. Nevertheless, the

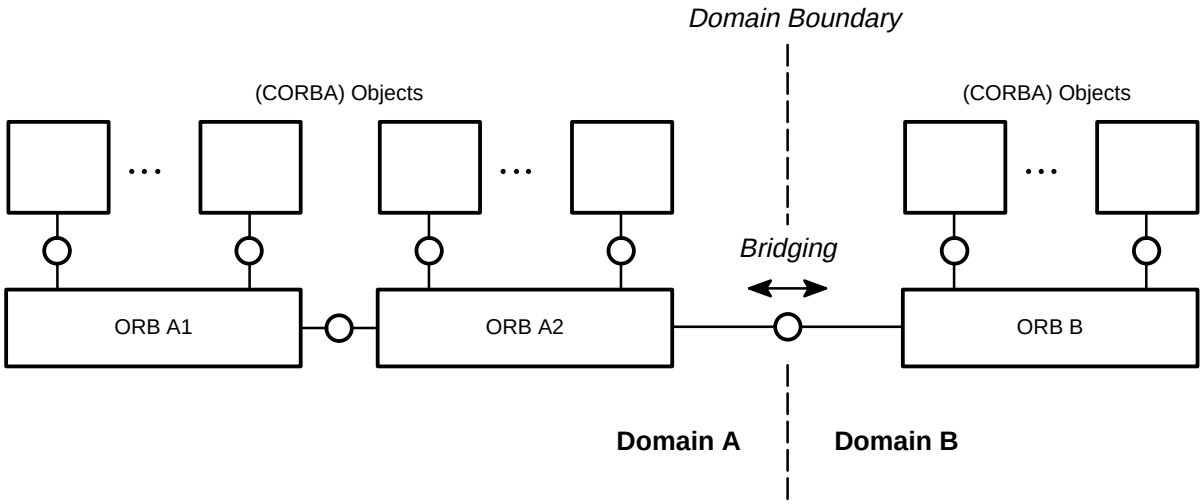


Figure C9 Example Domains

Bridging

CORBA subsystems cooperate by transmitting requests and responses accross the domain boundary. This communication via a common *inter–ORB protocol* is called *"Bridging"*. In contrast, the communication between ORB A1 and ORB A2 is *not* Bridging, but an internal implementation detail of domain A. The CORBA 2.0 specification only defines Bridging possibilities. Internal communication mechanisms can be designed freely.

In many cases – like the example above – domain boundaries and ORB implementation boundaries are identical. However, this may not be true for all domains. CORBA objects and ORBs can belong to more than one domain.

4.5.2 In–Line Bridging

The first bridging possibility defined by CORBA 2.0 – called "In–Line Bridging" – is depicted in **Figure 10** (upper part). In this case, Bridging is done *by the ORB itself,* i.e. it allows requests and responses to cross domain boundaries without any help of the application programmer.

4.5.3 Request–Level Bridging

If bridging is not a built–in feature of the ORB itself, there is a second possibility to achive interoperability, called *Request–Level Bridging.*

4.5.3.1 Interface–Specific Bridges

The principle of Request–Level Bridging is shown in the center of Figure 10:

Server Proxy Object

In the Client Object's domain, there is an additional object which receives requests instead of the Server Object. This object – called *Server Proxy Object* –

Client Proxy Object

passes the request to another additional object located in the Server Object's domain. The addressed object – the *Client Proxy Object* – plays the role of the original Client Object and sends the request to the Server Object. If requests have to be transmitted in *both* directions, a second pair of Proxy Objects (including Stubs and Skeletons) has to be installed in analogue form.

Different Meanings of "Proxy"

The reader should notice, that the term "Proxy Objects" – in the context of Bridging – does not correspond to the local Proxy Objects implemented as stub code.

As shown in the center of Figure 10, the coupling of Proxy Objects to their local ORBs is done by using Skeletons and Stubs as defined by CORBA 1.2 (see Chapter 4.1.2.). This solution works well, as long as the type of the Server Object remains unchanged. But each time, the type is changed by the application programmer, the type of the Server Proxy Object has to be changed, too. The application programmer has to rebuild both, the Server Proxy Object and the underlying Skeleton.

Due to this dependency on the Server Object's interface, this kind of Bridging is also called *Interface–Specific Bridging.*

4.5.3.2 Generic Bridges

Dynamic Skeleton (Interface)

As described above, the programmer of the Server Proxy Object needs to know the interface of the Server Object at compile time. The OMG has defined a new skeleton type – called *Dynamic Skeleton* – to get rid of this limitation. A Dynamic Skeleton (also called Dynamic Skeleton Interface or DSI) provides the same services as a "static" Skeleton: It passes requests to the Server Object and returns the results back to the ORB. In contrast to a static Skeleton, the Dynamic Skeleton does not map each type of requested operation to its own method. Instead, there is *one* unique "method" in the Server Object's Implementation, to which *all requests* are mapped. This special method is called *Dynamic Implementation Routine* (DIR). The Dynamic Implementation Routine receives a *Server Request* description as parameter, which provides the following information:

Dynamic Implementation Routine

- *the name of the requested operation's type.*
- *a reference to the operation definition.*
  This reference allows searching through the Interface Repository to acquire the type description of the requested operation (see also Chapter 4.3.). This can be useful for looking up the Server Object's interface, for example.
- *the context information.* (see also Chapter 4.1.1)
- *the operation parameters.*
- *the information needed to return the result correctly.*
  The determination of this information depends highly on the language mapping.

- *the information needed to identify the Server Object.*
  This also depends on the language mapping.

The lower part of Figure 10 shows the usage of the Dynamic Skeleton Interface when building a *"Generic Bridge"*. The Server Proxy Object receives the request description and simply passes it to the Client Proxy Object, using an inter–ORB protocol. The Client Proxy Object delivers the received request description to the ORB Core via the Dynamic Invocation Interface. (see also Chapter 4.4.) When using the Dynamic Skeleton and the Dynamic Invocation Interface, both Proxy Objects *can be implemented independently of the Server Object's Type.*

Half Bridge

Each Proxy Object – including it's Dynamic Skeleton or Dynamic Invocation Interface – is called a *Half Bridge.* The combination of two Half Bridges, as shown in Figure 10, is called a *Full Bridge.*

Full Bridge

Of course, the Dynamic Skeleton Interface has other applications beyond Bridging. An example is the coupling of the ORB Core to a Server Object Implementation which is written in a dynamically–typed language like LISP.

4.5.4 Inter–ORB Protocols

Inter–ORB Protocol Types

Of course, bridging between any two CORBA systems is only possible if there is a common Inter–ORB Protocol. The CORBA 2.0 specification defines several protocols, see **Figure C10.**

These Inter–ORB Protocol types are defined by the OMG:

- *General Inter–ORB Protocol (GIOP)*
  The GIOP is defined as an *connection–oriented* protocol. The specification defines the low–level data representation of IDL types like integer, character or floating point. IDL Interfaces are defined for handling connections and encoded data.
- *Internet Inter–ORB Protocol (IIOP)*
  The IIOP specifies a subtype of the GIOP based on the TCP/IP Protocol.
- *Environment Specific Inter–ORB Protocol (ESIOP)*
  The OMG allows other, non–GIOP protocols for Inter–ORB Communication. These protocols can be used to integrate other distributed computing systems which are already in general use.
- *DCE Environment Specific Inter–ORB Protocol (DCE ESIOP)*
  The Distributed Computing Environment (DCE) has been developed by the Open Software Foundation to simplify the implementation of distributed applications. The basic communication mechanism of DCE is a remote procedure call, DCE RPC.
  The DCE ESIOP is an Inter–ORB Protocol based on the DCE RPC protocol.

CORBA Interoperability

To achive interoperability, a CORBA 2.0 Implementation *must provide an implementation of the GIOP IDL Interfaces and an IIOP compliant mapping on*

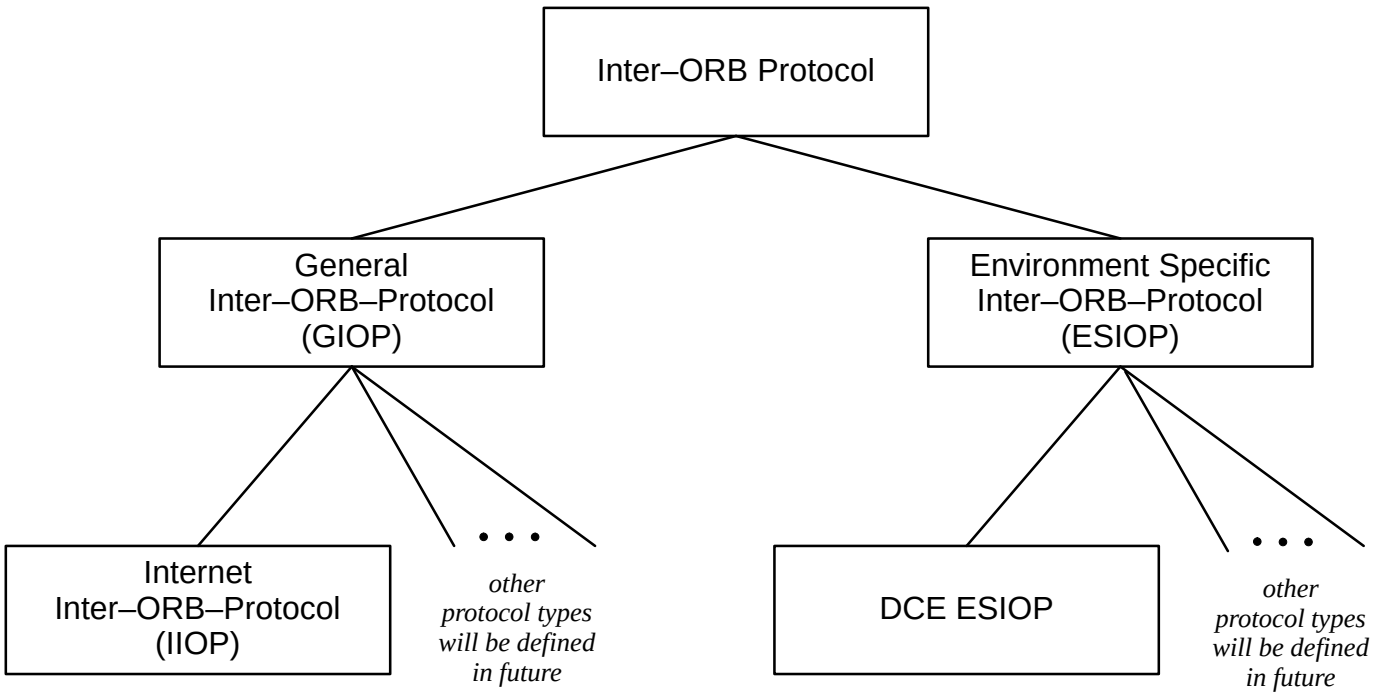


Figure C10 Inter–ORB Protocol Types

TCP/IP. IIOP support must either be provided by In–Line Bridging or by Request–Level Bridging via Half Bridges.
The DCE ESIOP is defined by the CORBA 2.0 specification, but is *not mandatory* for interoperability.

4.6 Object Adapters

The model of a CORBA system as presented up to now could cause some severe problems when implementing server objects. An extension to the system has to be made to achieve the following features:

- *Interoperability with other object–oriented systems*
  Besides the CORBA specification there are other systems which are object–oriented and – probably – even distributed. For some reason, it could be useful to connect such a system to a CORBA–compliant system. In this case, objects of the different systems could cooperate. The question is *how to connect* the two systems.

The use of the Dynamic Skeleton Interface is a connection possibility, but may not be suitable for building an highly optimized interface.
Of course, there is also the possibility of using the Internet Inter–ORB Protocol to connect to the ORB. This solution could cause much effort, if it is neccessary to add an ORB's functionality to the non–CORBA system.
Due to this, there should be a way to *integrate objects of a non–CORBA system without the need for an additional ORB.*

- *Integration of existing software* ("legacies")
  It is desirable to offer the services of an existing application to objects of a CORBA system without rebuilding the application. A good solution would be reached if *the application was addressed as one or more CORBA server objects.*

- *Economize on hardware resources*
  There are likely to be server objects which are rarely used but require a lot of hardware resources when implemented. An example is a database server object which needs a lot of RAM but is only used a few times each day. Creating a server object each time one is needed (and removing it later) would force client objects to:
  - coordinate with other client objects,
  - check for a server object's existence,
  - start and remove a server object,
  - keep a server object's data persistent,
  - etc. ...

  These problems can be avoided if there is some mechanism to
  - *keep a server object addressable and persistent* on the one hand and
  - *free hardware resources if they are only needed during the server object's activity,* on the other hand.

  All this should be invisible for a client object, i.e. there seems to be a server object, even when there exists only a file containing the server object's data.

Object Adapter

A solution for the problems discussed above is given by the so–called *Object Adapters.* These are links between the implementation of server objects and the ORB Core and have been omitted in the diagrams up to now (see **Figure C11**).

Basic Object Adapter

There are two important types of Object Adapters. The first one is called *Basic Object Adapter* (BOA) and is intended to be useful for almost every implementation of server objects. This adapter is standardized within the CORBA specification and *must be available in every CORBA environment.*

The Basic Object Adapter provides all of the features to the CORBA system as described above. The Basic Object Adapter and the implementation of server objects using a BOA will be discussed in Chapter 4.7 in more detail.

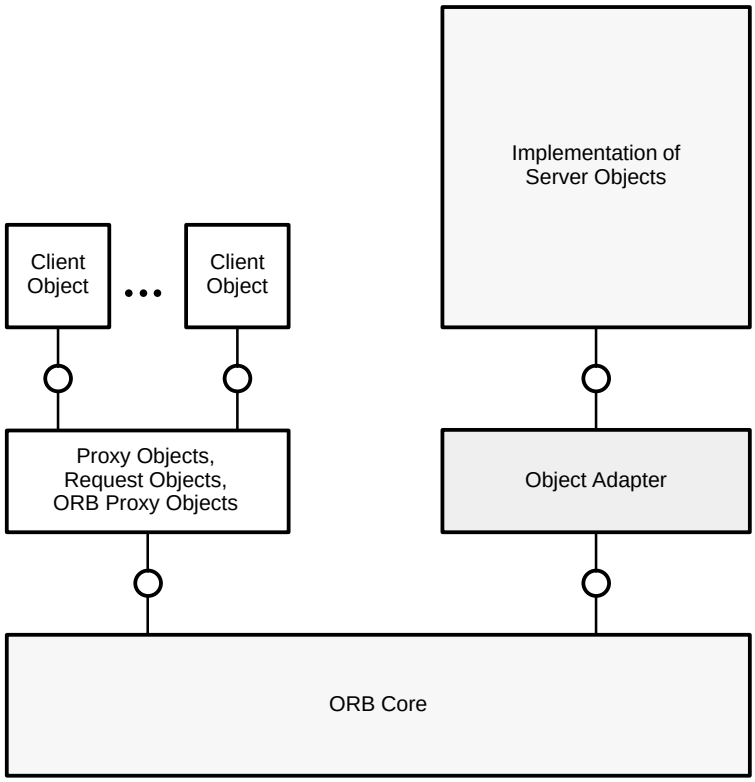


Figure C11 Object Adapter

Special Purpose Object Adapter

The second type of object adapter is not standardized by the CORBA specification. It is called *Special Purpose Object Adapter* and allows server object implementations for which the Basic Object Adapter is not efficient enough.
Some Special Purpose Object Adapters are discussed – but not defined – in the CORBA specification:
One of these is an adapter for mapping requests to objects of a non–CORBA object system.
Another one is called *Object–Oriented Database Adapter.* It is designed as a means to access object–oriented databases and simply serves as a request/response forwarder to an OODB system.

4.7 The Basic Object Adapter

Figure 11 presents an overview of a CORBA system, now showing the *Basic Object Adapte*r used to implement a group of server objects. The coupling of *client objects* to the *ORB Core* via *proxy objects*, *request objects or ORB proxy objects* can be seen in the upper left. In the lower part of the diagram, there are the *Implementation Repository* and the *Interface Repository.* These parts of the model are still valid.

A System Model Closer to Implementation

However, server objects – as shown in earlier diagrams – are no longer existing in the model presented here. Because the diagram is closer to the implementation, server objects and skeletons are replaced by the subsystem including the Basic Object Adapter. (*"System Part Implementing a Group of Server Objects"* – see Figure 11.)

Mapping of Object References

Now, any request sent by a client object will be delivered to the Basic Object Adapter by the ORB Core. The *ORB Core Communication Agent* maps Object References given with a request to the right object adapter. Due to this, there must be an *Object Adapter Location Database* that holds the neccessary information. (In a CORBA implementation this information is possibly stored in the Implementation Repository.)

Server

Service operations, as requested by client objects, are now performed by the so–called *Servers.*

Server vs. Server Object

The term "Server" has been chosen by the OMG as a name for an agent that performs requested operations instead of a server object. Because a Server does not stand for a server object and the two terms belong to different degrees of abstraction, they should not be mixed up.

In connection with Figure 11, the "existence of a server object" means the availability of an object reference that can be used by client objects when sending requests.

In an UNIX–based CORBA implementation, a server would be a "process" (in terms of UNIX).

Dispatching of Servers

As we have mentioned, service operations are performed by Servers, i.e. requests are delivered to them by the Basic Object Adapter. If no Server is available to perform an operation, the *BOA Server Dispatcher* will *create one.* This is done using the underlying *operating system services.* The Implementation Repository contains the information needed to create a Server for a certain request (for example, the name of a file containing the right program code). This information is retrieved by the BOA and passed to the operating system.

On the other hand, a Server can be *removed from the system* if it is no longer needed. The decision to do this is made *by the Server itself,* i.e. the Server can ask the operating system to remove itself. Before doing this, the Server will *notify the BOA that it is no longer available.*

Four Types of Servers

There are four possible types of Servers:

- *Server Per Method*
  A Server Per Method is *created each time a request is sent* to a server object. The Server only runs for the duration of the requested operation, i.e. it will be *removed after returning the operation result.* Several Servers for the same server object or even the same method of that object may be active simultanously. As a concequence, Servers for one server object must cooperate when performing operations which affect the same data.
  Such a kind of Server is useful in batch processing, for example.

- *Unshared Server*
  An Unshared Server stands for a single server object. (There may be more than one Server during the server object's lifetime, but the Servers do not exist simultanously.)
  A typical case of such a Server is a single application which is integrated as one CORBA object.
- *Shared Server*
  A shared Server stands for *multiple* server objects, usually being of *different object types.* Server objects being implemented by a Shared Server have the *same Implementation Type*. (One program code will be executed, for example.)
  This is likely to be the most common kind of Server. For example, a Shared Server could be a UNIX "process" programmed using an object–oriented programming language like C++.
- *Persistent Server*
  Like a Shared Server, a Persistent Server implements multiple server objects.
  Unlike all other Server types, a Persistent Server is build up once, *independently from the Basic Object Adapter* (it is created from "outside" the CORBA system). A Persistent Server is *not removed* when no longer needed.

Request Delivery

The process of delivering requests to a Basic Object Adapter is shown in **Figure 12.** The client object sends its request via the ORB Core. The ORB Core simply passes the request to the Basic Object Adapter which is the link to the implementation of the addressed server object. For the ORB Core, "there seems to be a server object behind the Basic Object Adapter", regardless of the existence of a Server. It is the Basic Object Adapter's task to select or create a Server and deliver the request to it.
As the Petri net indicates, the Basic Object Adapter has to perform one of four possible processes for request delivery. Each process is specific to the Server type and will be discribed in the following Chapters.

BOA Database

The data needed by the Basic Object Adapter to manage Servers and to deliver requests is stored in the *BOA Database*. It contains the following information:

- The (CORBA) *object references of server objects* being implemented on top of the BOA.
- *Information about existing Servers* and which server objects are implemented by them.
- For each server object, there is the *object type description* and the *implementation type description* available. (The latter one is needed when starting a Server.) Only references to descriptions are stored in the database, the descriptions can be retrieved from the *Implementation/Interface Repository.*
- For each server object, there is place to store some *additional data* which can be used by the Servers for their own purposes.

This can be used for saving the server object's data when a Server is removed, for example.

The terms "BOA Database" and "BOA Server Dispatcher" are only used in this report and are *not part of the CORBA terminology.* They are components of the system model which is shown here to explain the Basic Object Adapter.

4.7.1 Server Per Method

As shown in Figure 12, there is a subsystem that implements server objects. **Figure 13** presents this subsystem with the Servers being *Servers Per Method.*

Work Object

Inside the Server, there is a *Work Object.* A Work Object is the part of a Server that performs service operations requested by client objects. This is valid for the other Server types, too. Of course, a Work Object has the same type as the Server Object it implements.

The term "Work Object" is only used in this report and is *not part of the CORBA terminology.*

Skeleton

As the figure shows, *Skeletons* are still needed for unmarshalling incoming requests. They are now connected to the Basic Object Adapter which passes requests to the Server.

BOA Proxy Object

Work Objects can send requests to the Basic Object Adapter via the *BOA Proxy Object.* (The request types will be discussed later.)

The BOA Database contains the information needed by the BOA Server Dispatcher, see right part of Figure 13. For each server object, there is an entry in the database, *"Data per Server Object"*.

When a request is delivered to the BOA Server Dispatcher, a Server will be started, *even if there is already one for the addressed server object.* The whole process is shown in **Figure 14**.
First, the BOA Server Dispatcher has to find out what kind of Server has to be started. This is done using the implementation type description reference belonging to the server object.
In the next step, the Server will be started. During initialization, the Server will restore the server object's persistent data in its memory.
By starting the Server, the request is implicitly passed to it. Therefore, the server will execute the appropriate method right after initialization.

Method Execution

After returning the result – which is passed to the BOA – the Server initiates its removal. If neccessary, the Server will first save the server object's persistent data.

4.7.2 Unshared Server

A subsystem with Unshared Servers is shown in **Figure 15**. An Unshared Server handles requests for *one* server object. There will be no other Server if one is already available for that server object.

The process of preparing and starting method execution is shown in **Figure 16**. When receiving a request, the BOA Server Dispatcher has to know if there is a Server belonging to the addressed server object. This information is given by the *Server Existence Flag* inside the BOA Database.
If the flag is set, the Server is identified by the *Server Reference* – the request can be delivered to the Server.
If the flag is not set – i.e. there is no Server available – the BOA Server Dispatcher finds out, what kind of Server has to be started. After Server creation, the BOA Server Dispatcher saves the Server Reference to the BOA Database. When the Server is ready, the Server Existence Flag is set. Finally, the request is delivered to the Server.

Method Execution

Implementation Activation/ Deactivation

The starting of a Server is also called *Implementation Activation* – its destruction is called *Implementation Deactivation*.

Object Activation/ Deactivation

The creation/destruction of a Work Object is called *Object Activation/Deactivation.*

In case of an Unshared Server or Server Per Method, there is no difference between Implementation Activation/Deactivation and Object Activation/Deactivation because a Work Object exists as long as its Server.

4.7.3 Shared Server

Multiple Work Objects per Server

Figure 17 presents a subsystem with the Servers being *Shared* Servers. A Shared Server may contain *multiple* Work Objects. Unlike the Server types described above, Work Objects are built and removed *during the existence of the Server.* This is done by the *Agent for Communication and Creation/Destruction of Work Objects.*
Like Servers, Work Objects can even be removed (to free hardware resources, for example). Nevertheless, the related server objects still seem to exist for the client objects. Before removing a Work Object, the Server will *notify the BOA that the Work Object is no longer available.*

Server objects of different object types can have the *same implementation type.* For example, Work Objects can be C++ objects of different types in one program. Therefore, BOA Database entries for server objects of the same implementation type are grouped into a single database entry called *"Data per Implementation Type"*.

Because Work Objects can be created and removed inside a Server, there are *Work Object Existence Flags* in addition to the Server Existence Flag.

The process of preparing and starting method execution is shown in **Figure 18**. When receiving a request, the BOA Server Dispatcher has to know if there is a Work Object belonging to the addressed server object. This information is given by the *Work Object Existence Flag:*

- If there *is* a Work Object, the Server is ready to receive the request.
- If there *is no* Work Object, the BOA Server Dispatcher checks the Server Existence Flag. If no Server is available, one will be created as already described for the Unshared Server type (see Chapter 4.7.2.).
  Then, the Server is told to create a Work Object. (If the Server was just created, it does not contain a Work Object.)

Method Execution

Finally, the request is delivered to the Server (as identified by the Server Reference).
Choosing the right Work Object is likely to be done by the Server itself because this is specific to Server implementation details. For this purpose, a Server may store some kind of Work Object Reference in the *Additional Data for Work Object* (which belongs to a certain Server Object Reference).

4.7.4 Persistent Server

A Persistent Server is nearly the same as a Shared Server. The only difference is that the Server is *created independently from the Basic Object Adapter.* This means, that there must be a Server available when a request is delivered to the BOA Server Dispatcher. If there is none available, the BOA Server Dispatcher will return an error to the ORB Core. No Server will be created in this case.

4.7.5 Additional Functionality of the Basic Object Adapter

There are several, additional services provided by the Basic Object Adapter to Servers. The Basic Object Adapter can be accessed via a BOA Proxy Object.

Creation of Object References

An important service is the *creation of new object references.* A Server can use the Basic Object Adapter to ”create a new server object” or, being more precisely, create a new object reference, see **Figure C12**. When the Server wants a new object reference, it handles the intended object type and implementation type to the Basic Object Adapter. The Basic Object Adapter creates a new BOA Database entry describing the new server object. A Server/Work Object will be started by the Basic Object Adapter when the first request is sent using the new object reference.

Releasing of Object Refences

Of course, an object reference can be released, too. In this case, the related data in the BOA Database will be removed and the reference is no longer valid for the ORB Core.

Factory Object

The creation of new server objects is usually done by means of special server objects, called *Factory Objects:* A client object simply sends a request to a Factory Object which builds the requested new object. A Work Object implementing this Factory Object would use the BOA services described above.
More details about Factory Objects can be found in Chapter 5.4.

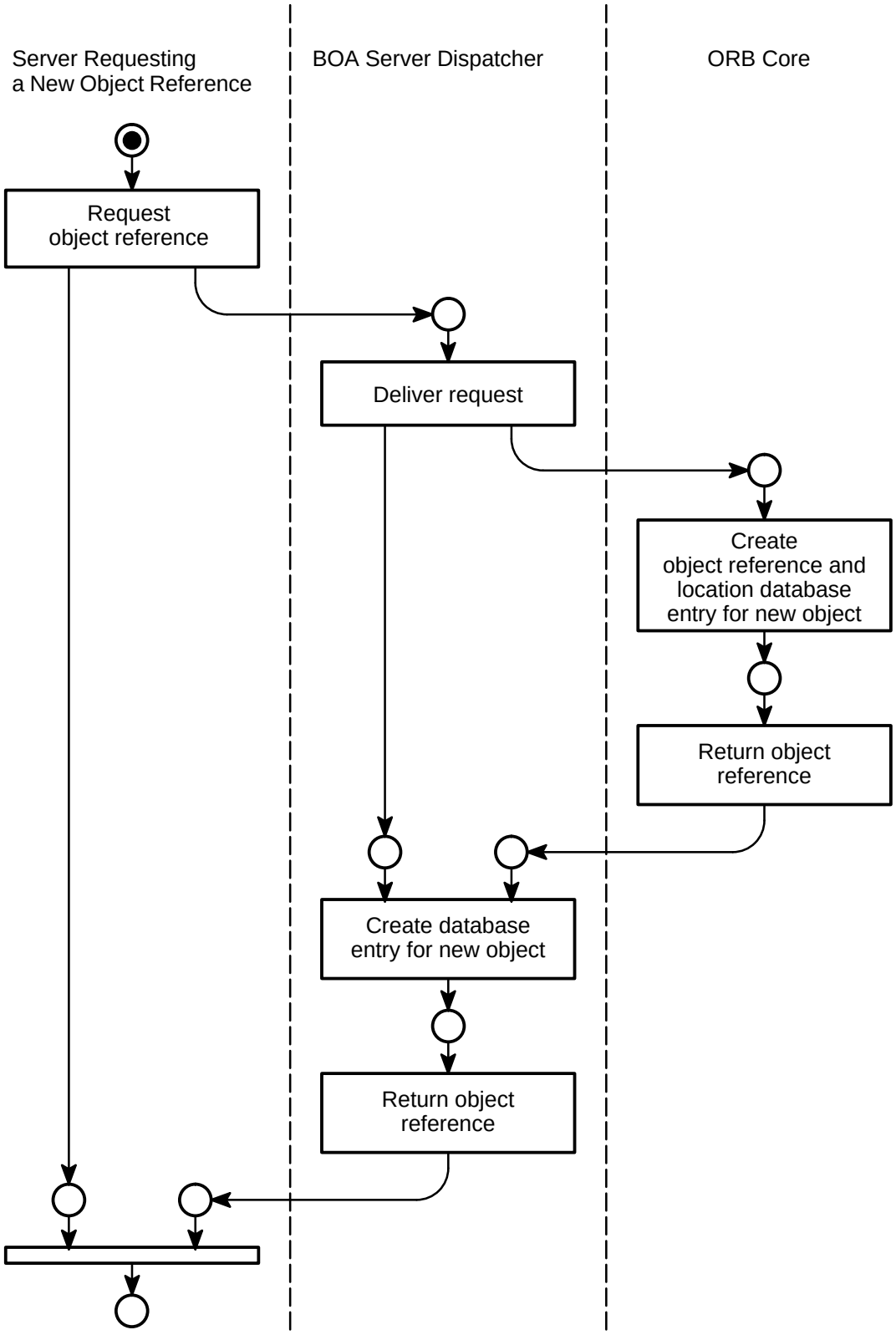


Figure C12 Creation of a Server Object (Reference)

Authentication

A Server can ask the Basic Object Adapter to return information about the client objects of a server object. This can be useful for finding out if a client is allowed to request certain service operations.

Changing the Implementation Type

The Basic Object Adapter can be sent a *new implementation type* (description) for an *existing server object reference*. This does *not* affect the object type or the object reference, both remain valid.

If there is no server running when the implementation type is changed, the next request will cause the creation of a Work Object/Server as defined by the new implementation type.

It is not defined what happens if the implementation type is changed with a Work Object/Server running.

5. Fundamental Object Services

Chapter 4 has presented the basic parts of the Object Management Architecture (OMA), namely the Interface Repository, Implementation Repository and the components of the Object Request Broker. These parts are neccessary for object communication and implementation.

Standardized Server Objects

Additionaly, basic services are provided by special, *standardized server objects*. These services are the Object Services, which are defined in the Common Object Services Specification (COSS).

In general, there are two types of Object Services:

- *Services to be used*
  If the programmer of a CORBA compliant application only wants to *use* a service, she or he can make use of *pre–built server objects*. These server objects are defined by the COSS – they are called *Specific Objects*.

Specific Object

  An example for this is the Naming Service, where server objects are available which provide services such as
  – the binding of objects to names
  – resolving a name
  – etc. ...
- *Services to be provided*
  If the programmer wants to *provide* services to others, she or he has to build server objects. In this case, the Object Services Specification defines object types the programmer should use as (super–) types for the server objects to be build. These object types – called *Generic Objects* – are also defined by the COSS. The use of predefined object types leads to a *standardized* set of services provided by the server object. Of course, the programmer still has to implement the appropriate functionality.

Generic Object

  An example for such a service is the Lifecycle Service. It defines the object type *Lifecycle object*. An object that is *derived from that type* understands *standard* requests for
  – creating a copy of its own,
  – move itself from one host computer to another,
  – etc. ...

The first part of the COSS – *COSS Volume 1* – defines four important Object Services. These will be described in the following chapters.

5.1 Naming Service

Hierarchical Name Structure

Using the Naming Service, objects can be given a name. Object names are structured in a hierarchy like file names in a file system, see **Figure C13**. The diagram shows an example of object names and their embedding in a naming graph.

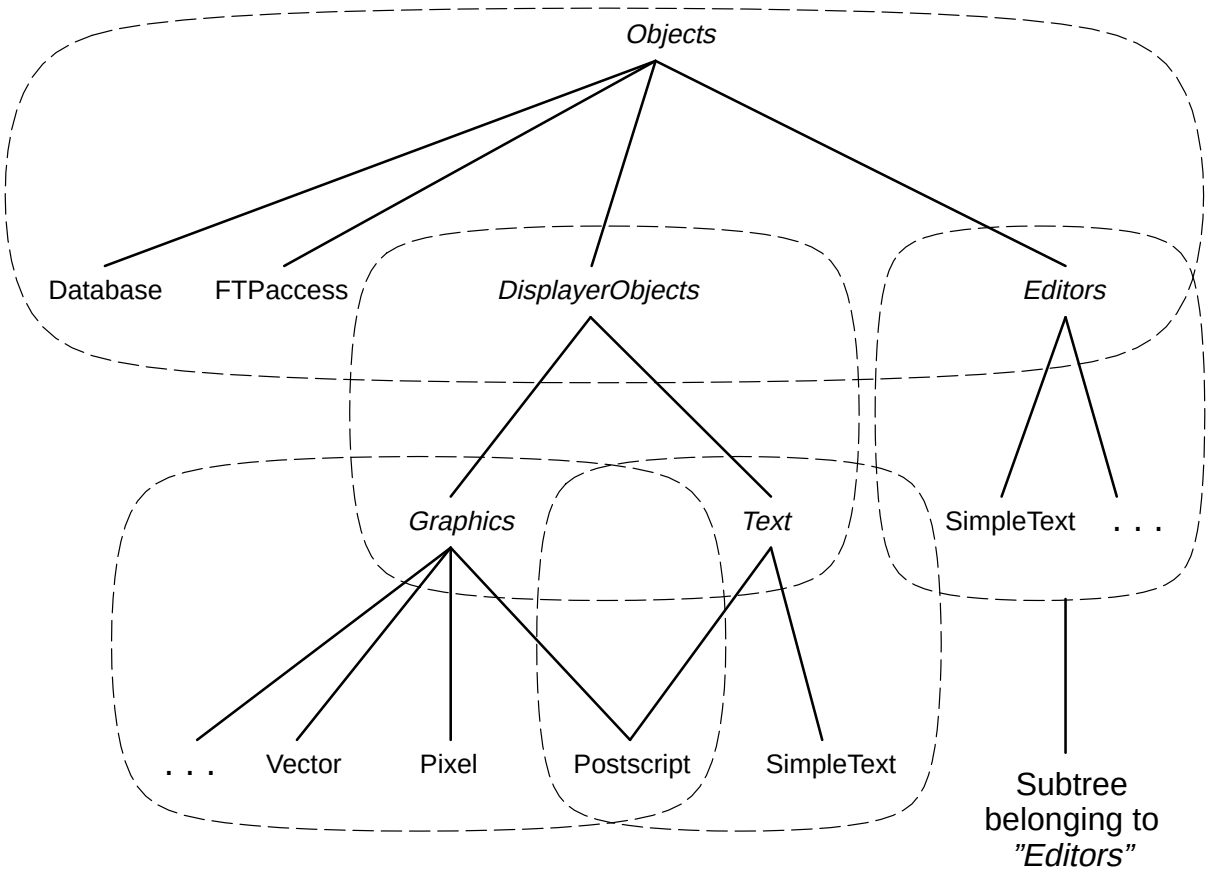


Figure C13 Naming Graph Example

Context

A leaf of the naming graph represents an object's name. A node in a naming graph is called a *context.*

Simple Name

Object names are unique within a subtree belonging to a certain context. For example, the name "SimpleText" is unique within the "Editors"–subtree and within the "Text"–subtree. Such object names are called *simple names.* An object may have multiple, different names within a context. It may also have the same name in different contexts – an example is given by the object name "Postscript" which is used *twice* within different contexts. Nevertheless, it identifies *one single* object for displaying Postscript data.

Compound Name

Objects can also be identified using *compound names.* A compound name consists of the names given along the path from a node to a leaf; for example *"Objects;DisplayerObjects;Graphics;*Pixel*"* which identifies an object that provides services for displaying pixel–based images.

Context Object

For using the Name Service, a client object must have access to a *Context Object.* A context object stands for a node in the naming graph. It allows access to the names/objects belonging to its subtree. For example, the context object named "Text" knows the objects named "Postscript" and "SimpleText".
By using compound names, the names/objects of all deeper subtrees/sub–sub-

trees etc. can be accessed. For example, the context object called ”Displayer-Objects” knows the names *”Graphics;*Postscript*”* and *”Text;*Postscript*”*.

A context object offers the following services:

- *Giving an object a new or an additional name (bind)*
  The object’s new name is added to the subtree belonging to the context object. The object given a new name may be an ”ordinary” object or a context object. An example for the latter one is the context object named ”Graphics”.
- *Delete a name (unbind)*
  A client can tell the context object to remove a name for a certain object. (The name is removed from the subtree.)
- *Resolve a name*
  A context object can connect a client to a named object. (i.e. a proxy object will be created for the client to access the named object.) For doing this, a client object simply passes a request (and the name of the desired object) to the context object.
- *List all known names*
  A context object can return a list of the object names in its subtree. (”Postscript” and ”SimpleText” for the ”Text”–subtree, for example.)
- *Create/delete a context object*
  When creating a context object, the name for the newly created context object is added to the subtree. The new context object can be used to build a sub–subtree. This is similar to the creation of a subdirectory in a file system. A context object (and the subtree it is responsible for) can even be removed. Before doing this, the subtree should be empty, i.e. the subtree’s names should be removed first.

Additional Type Information

A ”name” of an object consists of an *”identifier attribute”* and a *”kind attribute”*. The ”kind attribute” can be used to provide additional information about the object type.

5.2 Persistence Service

The Persistence Service defines the following services:

- *Giving Work Objects the means to save their persistent data*
  Before a Work Object is removed from the system, its data has to be stored in a persistent data storage. This is done using a standardized format which allows a later created, new Work Object to appropriately initialize its data – even if the new Work Object is implemented in a different way.
- *Allow client objects to control the persistence of a server object’s data*
  The Persistence Service defines operations for storing the data of a server object. These operations can be requested by client objects.

For example, a server object used to edit a document could be told to save its data – the document data – to a persistent storage when editing is finished.

5.3 Event Service

Event Channels The most important part of the Event Service is the definition of so–called *Event Channels.* These are objects which propagate events to all objects being interested in certain events.

Figure C14 shows an Event Channel and some objects being connected to it. (The ORB is not shown in the diagram, but objects are still connected via the ORB.)

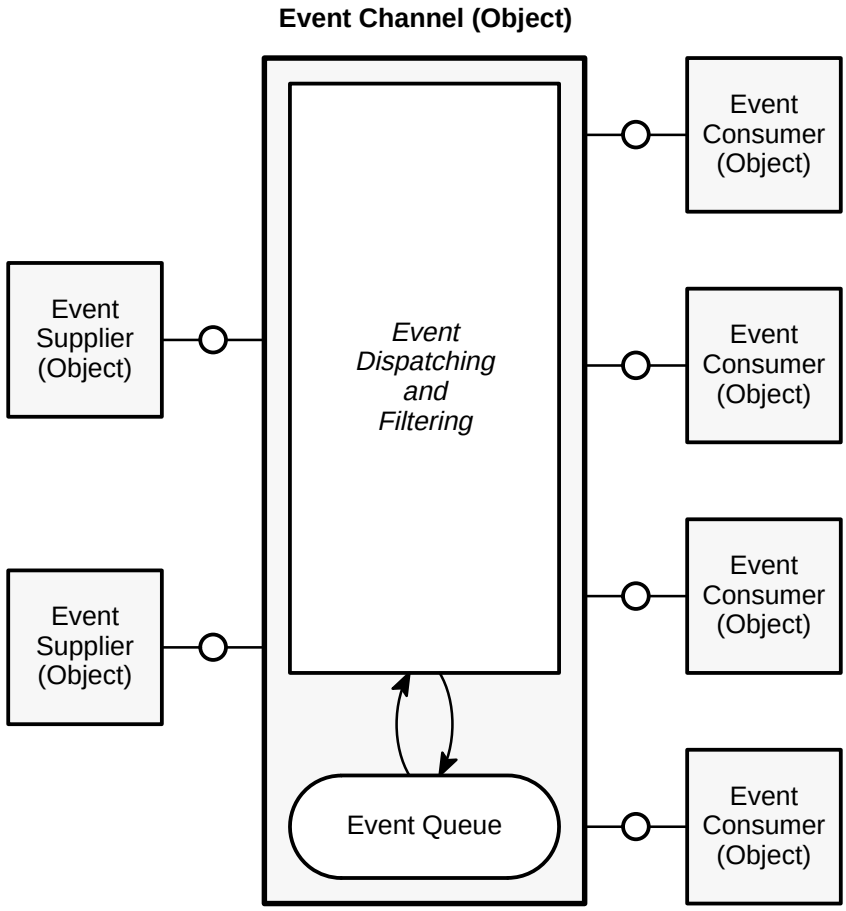


Figure C14 Event Channel

Event Supplier An *Event Supplier* is an object that sends events to others. (Sending events is done via special requests.) Consider an object that calculates a new stock market price, for example.

Event Consumer

On the other side, there are *Event Consumers*. These are objects which are interested in certain events. For example, objects which are part of an online–information system could be interested in the change of the stock market price (as mentioned above).
Note, that the Event Channel is both, an Event Consumer and an Event Supplier.

Pull/Push Style

There are two styles of event notification possible:

- *Push–Style*
  The *Event (Push–) Supplier sends requests* to the Event (Push–) Consumer to notify it of the occurence of an event.
- *Pull–Style*
  The *Event (Pull–) Consumer sends requests* to the Event (Pull–) Supplier to ask for events (event data). The Event Consumer can ask *if* there has been an event or it can wait for the next available event data.

If the Event Supplier does *not know who is interested in the events* it makes use of an Event Channel.
Before using an Event Channel, Suppliers and Consumers must register with the Channel. This is neccessary to inform the Event Channel about possible event types and about which objects are interested in which event types. The Event Channel provides the following services:

- *Event Queueing*
  Events sent to the Event Channel can be queued by the Event Channel when they are currently not needed by all Consumers (Pull–Style). When all Consumers have asked for the queued event data, it will be removed from the event queue.
- *Event Dispatching*
  If more than one Consumer is interested in a certain event, the Event Channel propagates the event to all interested Consumers. This frees the Supplier from the responsibility to inform all Consumers.
  If, for example, a stock market price is changed, *all* online–information systems (=Consumers) are automatically notified by the Event Channel.
- *Event Filtering*
  If some event types are not needed by any Consumer, events of that type will neither be propagated nor will they be queued – i.e. they will be ignored.

5.4 Lifecycle Service

Chapter 4 describes how objects can communicate via the Object Request Broker and how they are implemented via Servers and Work Objects. Nevertheless, the ORB does *not* offer a service which allows a client object to create CORBA objects. This capability was not given the ORB intentionally because the creation or destruction of objects is highly implementation dependent.

Factory Due to this, the creation of CORBA objects is done by special server objects, called *Factory Objects* or *Factories.* These objects are defined by the *Lifecycle Service.*

Generic Factory Factories are not defined in detail by the specification. There is only an object type – called *Generic Factory* – defined, which can be used as a pattern for implementing Factories.

Lifecycle Object In addition, the specification presents an object supertype called *LifecycleObject.* An object that is derived from that class is called *Lifecycle Object.* By having the same supertype there are certain services provided by all Lifecycle Objects.

The services provided by Factories and Lifecycle Objects are described in the following subsections.

5.4.1 Object Creation

Figure C15 shows the creation of an object by means of a Factory. The principle is simple – the client object sends a request to the Factory which creates the requested object.

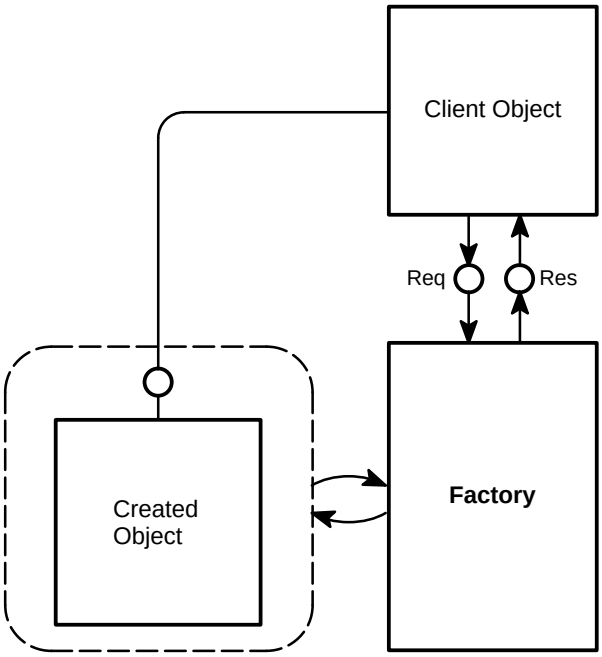


Figure C15 Object Creation

Key, Criteria Two parameters are sent to the factory for object creation, called *key* and *criteria.* The first must be sent whereas the second one is optional. Both parameters

are used to specify/constrain the type and implementation type of the object to be created. The specification does not standardize the semantics of these parameters.

5.4.2 Object destruction

A Lifecycle Object can receive a request that leads to its removal. **Figure C16** shows a scenario, where a client object sends a request to an object to be removed.
A request result is returned to the client which means an acknowledgement of the request. The destruction process itself is not defined by the specification. Usually, the object will pass the request for its destruction to an agent for removing objects (like a special Factory Object or the underlying operating system).

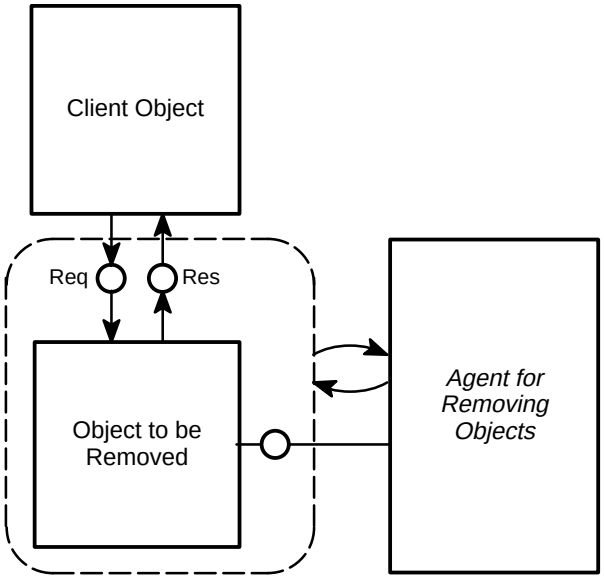


Figure C16 Object Destruction

5.4.3 Object duplication

A Lifecycle Object can be told to create a copy of its own. The result is a second object of identical type, which has the same state data and a *different* object reference. **Figure 19** illustrates this: In the diagram, there is a client object (see above) and an object to be copied (”Original Object”, see left.). The copying is not done by the object itself. Instead, a Factory is told to build a new object of the same type. After creating the new object (”Copied Object”), the Factory copies the object data.

Factory Finder The appropriate Factory is chosen by the Original Object. A list of available Factories is given to the Original Object by a *Factory Finder.*

Factory Finders are defined by the COSS, they provide only one service, namely returning a list of Factories. When asking for a list of Factories, a client (here the Original Object) sends a ”key” to the Factory Finder. This key is used to constrain the list of possible Factories.

On the other hand, the client object (see above) can influence the implementation type of the copied object by passing the reference of the Factory Finder and a ”criteria” parameter to the original object. The reference identifies the Factory Finder to be used by the original object. The ”criteria” parameter is usually passed to the selected Factory. The process of object duplication is depicted in detail in **Figure 20**.

5.4.4 Object movement

A client object can tell a Lifecycle Object to ”move to another location”. It is not defined by the COSS what ”moving” means in detail. In this context, ”moving” means *changing the implementation* of an object. The object reference, the object data, and the object type remain unchanged. Here are some examples:

- *change the physical location*
  An object could be replaced by an object implemented on another host computer, for example.
- *change the programming language*
  An object being programmed in C++ could be replaced by an object programmed in Smalltalk, for example.
- etc. ...

The movement of an object can be seen as an object duplication followed by a destruction of the original object. (The reference of the original object is assigned to the copied object.) Due to this, **Figure 19** shows both, object duplication and object movement. The diagram indicates that the Factory removes the original object only in case of object movement. (see comment ”only when moving object”.) **Figure 21** shows the process of moving an object in detail.

6. Interface Definition Language

The objective of the *Interface Definition Language* is to enable the description of an object type – also called interface – in a way that is independent of any programming language. The designers of IDL have chosen the definitional part of C++ as a model for their specification. So, IDL follows the same lexical rules as C++ and almost has the same set of keywords. Of course, some keywords had to be added.

IDL Definitions

Instead of giving a detailed description of IDL at this point, we want to illustrate the most important parts in an E/R diagram and by an example. The diagram is shown in **Figure 22**. There you can see that an *Interface Definition* comprises a set of definitions of different kinds: in the upper part, there are definitions of *Constants*, *Non–object Types*, and *Exceptions*. The non–object types include *structs*, *unions*, *enums*, *arrays*, *strings*, and *sequences* (i.e. a one–dimensional array with maximum size and a length that can be determined at runtime). Exceptions are like in C++ and can freely be defined in addition to a set of predefined CORBA standard exceptions.

Constants, Non–object Types, and Exceptions

Attributes

Two other kinds of definitions are shown in the lower part of Figure 22. One of these is the definition of *Attributes*. An attribute corresponds to a public member data of a C++ class, i.e. it contains data that can be accessed from clients. There are two types of attributes. First, some attributes can exclusively be accessed in read–only style. A read operation is automatically generated for these attributes during the compilation process of the IDL description. Second, we have those attributes that can be changed from outside. Here, an additional operation for writing this attribute will be created.

Operations

The lower left of Figure 22 depicts the most interesting kind of definition, the one for *Operations*. In the previous paragraph we have mentioned the set/get operations for attributes. These are shown here only for illustration purposes and are not really defined (written down in IDL). An application programmer can specify those *Other* operations that comprise the interface to the designed object.

Result and Parameter Types

Because this diagram depicts only those relationships of type textual containment of definitions (i.e. what type of definition can be made within another one), the structure of an operation definition is not shown in Figure 22. An operation is associated with a *Result Type*, i.e. an operation can hand back one particular value. If none is desired, this can be specified with type *void*. The result is not the only data that can be returned from an object request. *Parameters* of the request may be specified as *Out* or *Inout*. This means that the values of these parameters may be changed by the server during operation execution. The inout parameter can also serve as input parameter, just like the third type of parameter (*In*) does.

Scope

Implicitly, an interface serves as a name scope, i.e. types defined in different interfaces may have the same name without necessarily being identical. As an

explicit name scope the concept of a *module* is defined. Such a module can contain any kind of declarations. It is also possible to define types outside of the interface definition if they are intended to be visible more globally. Of course, this conflicts with the general objective of encapsulation.

Example

At this point, a simple example might be useful. The application area is the field of a warehouse that has to manage several goods. The interface of such a warehouse object will contain an operation to add an article to the warehouse. In IDL, this interface would look like the following:

```
interface warehouse {
          struct article {
                    string description;
                    int item_number;
          };

          exception FullStore {};

          boolean addArticle(in article newitem)
                    raises (FullStore);
};
```

Definitions

First, a struct *article* is defined that can contain information about the article description and item number. The second definition is an exception *FullStore* that shall be raised by the server if the adding of a new article is not possible because this would exceed the capacity of the store. The last element defines the operation *addArticle* that takes an article as argument, adds it to the store, and returns true if it was successful, false otherwise. Since the value false is not very much information for a client, the operation can also raise an exception to indicate that further adding of articles is not possible at the moment.

Compilation

After putting this IDL coding into a file *warehouse.idl*, it can be compiled by an IDL compiler producing the stubs and skeletons needed for the client subsystem and server subsystem. **Figure 23** illustrates this process. Client object code and stubs can then be compiled together using an ordinary compiler for the desired programming language yielding in the client subsystem executable. The same can be applied to server object code and skeleton(s) creating the server subsystem executable. In addition to the stub/skeleton coding, some more libraries may be added depending on the used CORBA implementation.

As you can see, making an application to a CORBA server object only takes the definition of the interface in IDL, compiling it and writing the code that is called by the skeleton. This is an easy way to make use of legacy software in an object environment. At the client side, all that has to be done is to get access to the reference of the server object and send a request via the generated proxy object to the server.

Portability

Since IDL is a mandatory part of every CORBA implementation, the design of object interfaces is portable among systems of different vendors. This means

that the design of the types can be reused. In contrast, the self–implemented part of a server or client may not be portable between various platforms.

C Language Mapping

In the remaining part of this section, the *C Language Mapping* of CORBA IDL will briefly be discussed. With this mapping, all IDL–defined types and data must be accessible from the programming language. Mechanisms that allow the issuing of object requests are also needed. The interested reader may be referred to the CORBA specification for a detailed description of the C mapping because a detailed description would exceed the limits of this report.

There is a predefined type *Object* in the C mapping where all the defined interfaces are mapped to. Operations of an interface are mapped to ordinary C functions. So this sample interface

```
interface foo {
          long bar(in long baz);
};
```

will be mapped to the following C definition:

```
typedef Object foo;
extern long foo_bar( foo o,
                     Environment *ev,
                     long baz);
```

The operation gets two additional parameters. The first is a reference to the target object because C does not have the concept of an object itself. The second parameter serves as connection to the CORBA environment. If the execution of a foo_bar operation raised an exception, information about this exception could be accessed via this environment parameter.

7. R/3 and CORBA

Question

After the detailed explanation of OMA and CORBA in the previous chapters, the remaining part will investigate the question ”how can R/3 and CORBA share their functionality?”. There are several approaches that could be chosen. We will discuss them in the following sections.

Client/ Server

Section 7.1 deals with R/3 as a CORBA client. It tries to answer the question of how could parts of R/3 issue CORBA requests in an easy way. The other direction, i.e. ”how can R/3 be requested by clients connected to a CORBA environment ?”, is examined in Section 7.2.

7.1 R/3 as a CORBA Client

R/3 CORBA Object Requester

If parts of R/3 are to be enabled to issue CORBA requests, a new functionality is necessary that can act as a CORBA client, accept calls from R/3, and delegate these calls to an ORB. Calls from R/3 to this new part could be accomplished via RFC in a first approach. The new component providing this functionality is called the *R/3 CORBA Requester* in **Figure C17**. It acts as an RFC server that sends incoming calls to an ORB via stubs or DII. The first choice is only possible for certain predefined services. The second possibility provides more flexibility so that any available CORBA object may be requested.

A tighter coupling between the requester and R/3 is also possible. This kind of connection, however, would cause more design and implementation effort. ABAP statements could be created that give a programmer the chance to issue CORBA requests (e.g. CALL METHOD OF CORBA OBJECT ...). The processing of these statements would involve a C function of the R/3 kernel that acts as the desired requester or as an RFC in the case of the OLE client.

7.2 R/3 as a CORBA Server

For R/3 to act as a CORBA server, two questions must be answered. The first question is ”what kind of objects should be provided?”. This topic is dicussed in Subsection 7.2.1. Second, a decision has to be taken as to what CORBA component R/3 should be connected two. This is examined in Subsection 7.2.2.

7.2.1 What Objects to export?

Accessible Objects

If the R/3 system has to act as server for CORBA requests, the first thing to decide is which objects have to be accessible from outside. There are two pos-

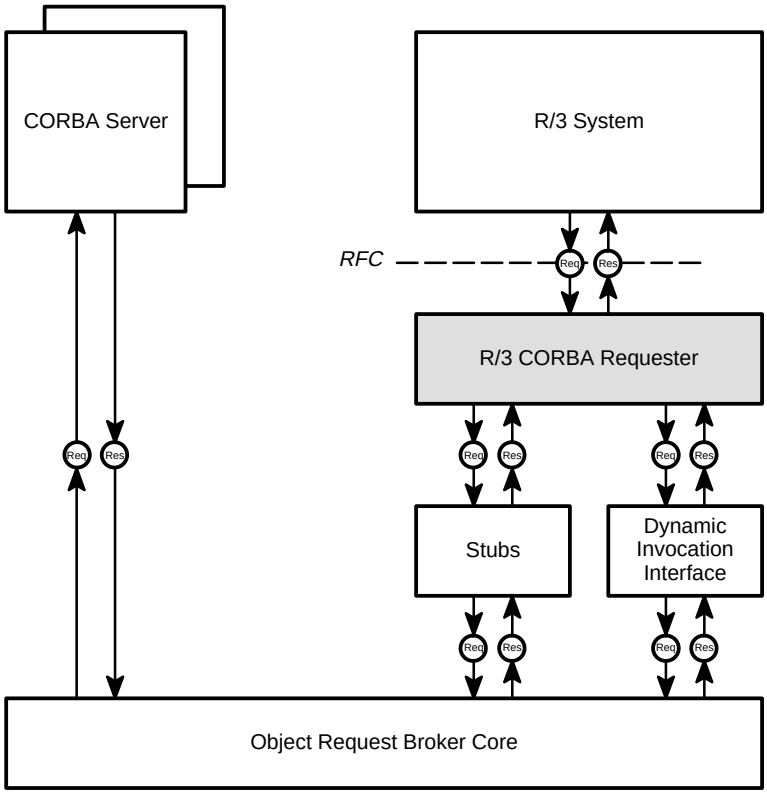


Figure C17 R/3 as CORBA Client

sible ways to do this. The first way is illustrated by **Figure C18**. The R/3 System exports its conventional system functionality as CORBA objects. One of these could be an object *R/3* providing core system functionalities as operations, e.g. *logon, call_report,* or *call_function_module*. Other exported objects could include an *enqueue server object*, a *batch processing object*, etc.

Business Objects

The other way to export functionality of R/3 via CORBA is to provide *Business Objects*, as shown in **Figure C19**. These business objects are constructed with R/3 internal means and implement business functionality. An example for this is a *customer object* containing an operation that hands the stored data of a certain customer to the requester. Other possible business objects could include *invoices, documents, suppliers*, *delivery notes*, etc.

The idea of object orientation suggests the second solution. With the first way, object requests to R/3 would be on a very low level. This would lead to a very procedural–like programming style for the clients. It could, however, be necessary to provide a small set of these system functionality objects in addition to the business objects.

Interface Description

After deciding what objects are to be provided, interface descriptions of these objects must be provided for client programmers. Especially in the case of providing a considerable number of business objects, it is necessary to generate

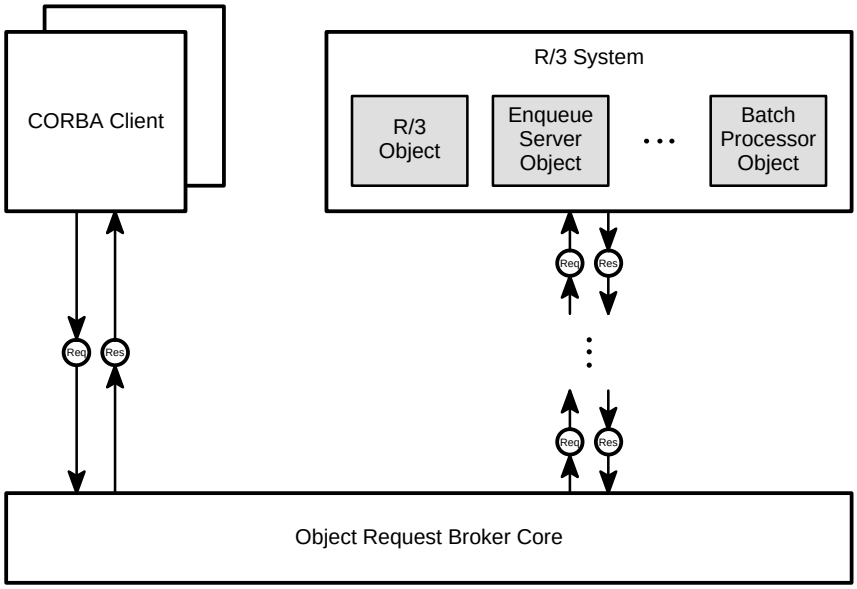


Figure C18 R/3 exporting System Functionality

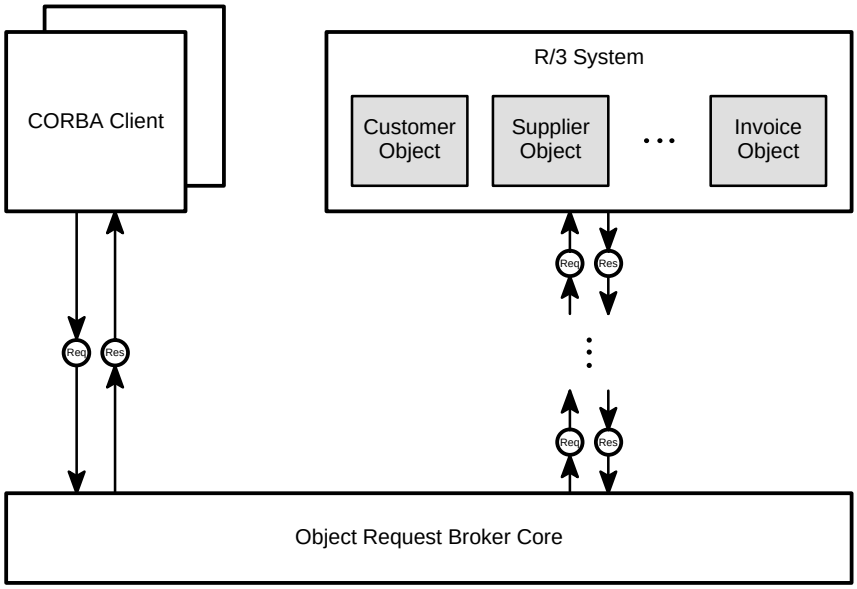


Figure C19 R/3 exporting Business Functionality

automatically the interface descriptions from the information that is stored within the R/3 System (in an object repository). The generated descriptions should be in CORBA IDL to enable the client developers to easily compile them to the necessary stubs.

7.2.2 R/3 – CORBA Connection Possibilities

Possible Interfaces

A critical question with regard to implementation and porting effort is ”where do we R/3 connect to?”. Taking a look at the Common ORB Architecture, we can identify three interfaces where R/3 could be coupled to CORBA. First, R/3 could be connected on top of an object adapter. Connections to different adapters may be possible (BOA, OODB Adapter). Another alternative is the direct connection to an ORB via object adapter/ORB interface. A third solution uses the recently adopted *Internet Inter–ORB Protocol* (IIOP). These solutions are discussed in the following paragraphs.

R/3 on Basic Object Adapter

Figure C20 shows the first coupling approach. The R/3 system acts as a persistent server connected to the Basic Object Adapter via skeletons. As in the case where R/3 acts as a client, a new functionality must be developed that is capable of mapping incoming CORBA requests to calls processible inside R/3. In Figure C20, this is called the *R/3 CORBA Request Receiver.* Inside this request receiver a proxy object that forwards the incoming CORBA requests to its real R/3 object would be implemented, for example via RFC. This approach only allows the provision of R/3 objects that have an interface known at construction time of this request receiver, because the skeletons used to reach these objects have to be created during compile time. Therefore, this solution demands a certain, static number of object types to provide the necessary functionality. No objects of new types could be exported without the new construction of the request receiver, e.g. during a patch or upgrade.

Dynamic Skeleton Interface

The new CORBA 2.0 specification includes a new component, called the Dynamic Skeleton Interface (DSI). In analogy to the DII on the client side, this provides the means to build a server on top of BOA for requests to objects with unknown interface during construction time of the server. (see also Chapter 4.5.3.2)

Instead of using the BOA, linking R/3 via an OODB Adapter could also be a possible solution. The R/3 System (with some functionality added to it) bearing business objects could act as an OODB. No examination of this kind of adapter was possible, yet.

R/3 Object Adapter

Another way for a connection between R/3 and CORBA could be the implementation of a new, different Object Adapter, as shown in **Figure C21**. Its task would comprise the mapping of requests coming from the ORB Core to requests that can be processed by R/3 work processes, e.g. via RFC. Of course, in this case the interface to the CORBA system would not be within the scope of OMG specifications. An *R/3 Object Adapter* would therefore be very ORB–dependent and difficult to implement. Providing connection to ORBs of different vendors would cause considerable porting effort. This solution is practical only for the support of one selected ORB or if the ORB providers would give much support to the R/3 Object Adapter implementors. In the best case, ORB providers themselves would develop an R/3 adapter for their systems.

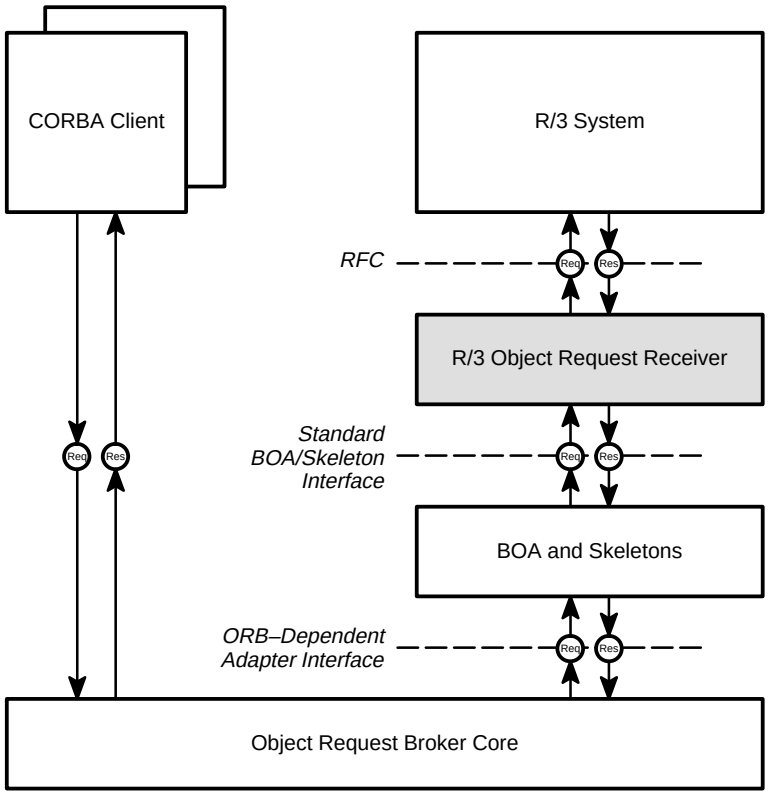


Figure C20 Linking R/3 via BOA

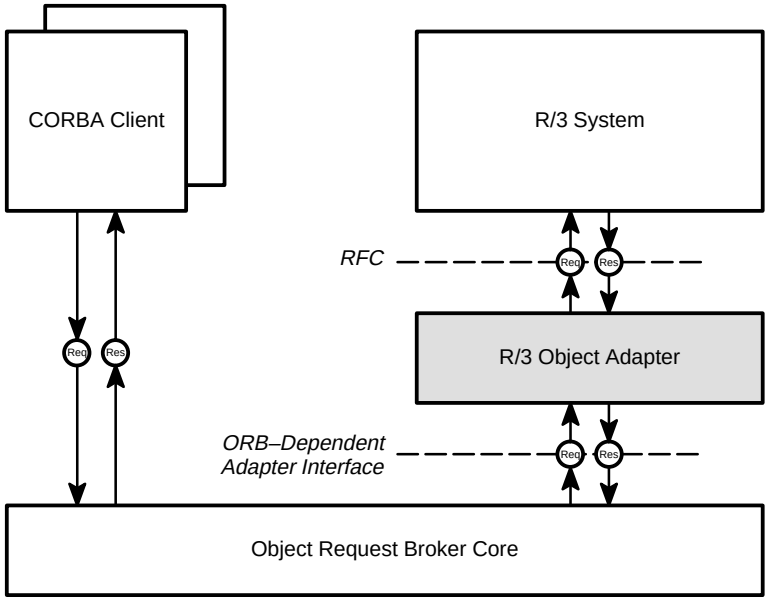


Figure C21 New R/3 Object Adapter

R/3 as an ORB

The last approach for an R/3–CORBA connection is an own ORB functionality added to the R/3 System. **Figure C22** shows this *R/3 ORB* linked to another ORB system via *Internet Inter–ORB Protocol*. This protocol serves for communication purposes between different CORBA environments and is standardized for the new CORBA 2.0 specification. The development of this solution might cause the highest effort of the proposed approaches. But it would also lead to the most flexibile solution because any CORBA system could be reached and no restrictions were imposed by a certain kind of adapter, skeleton, etc.

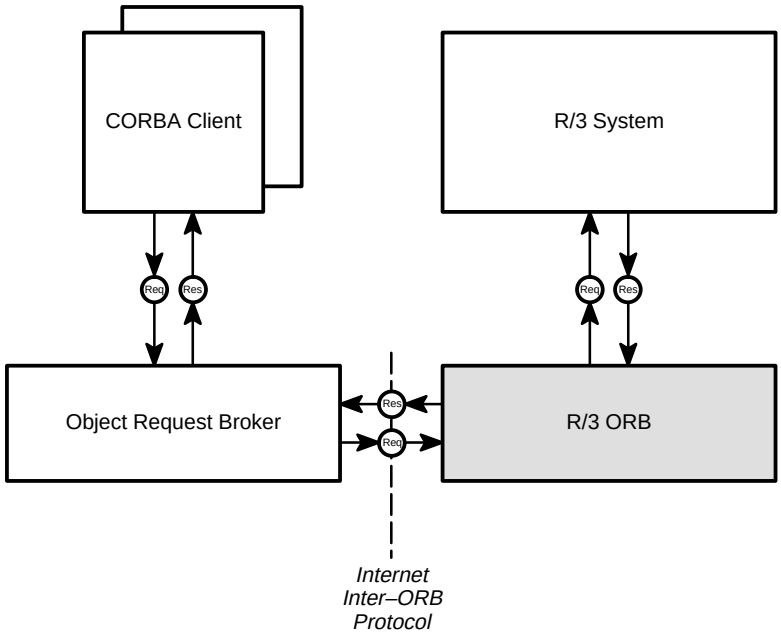


Figure C22 Linking R/3 via Internet Inter–ORB Protocol

Appendix

A. Bibliography

OMA
Object Management Architecture Guide
Revision 2.0, Second Edition, Sept. 1, 1992
Object Management Group
Richard Mark Soley, Ph.D. (ed.)
OMG TC Document 92.11.1
John Wiley & Sons, Inc
ISBN 0–471–58563–7

CORBA
The Common Object Request Broker: Architecture and Specification
Revision 1.1
Object Management Group
OMG TC Document 91.12.1
John Wiley & Sons, Inc
ISBN 0–471–58792–3

OSA
Object Services Architecture
Revision 8.0
Object Management Group
09 Dec. 1994
OMG Document Number 94–x–x

COSS 1
Common Object Services Specification Volume 1
Revision 1.0
First Edition
March 1, 1994
Object Management Group
Jon Siegel, Ph.D. (ed.)
OMG Document Number 94–1–1

COSS 1a
Addendum to Common Object Services Specification Volume 1
Revision 1.0
First Edition
March 1, 1994 and October 12, 1994
Object Management Group
Jon Siegel, Ph.D. (ed.)
OMG Document Number 94–10–7

COSS 2
Common Object Services Specification Volume 2
Object Management Group
OMG Document Number 95–x–x

ORB 2.0 RFP
ORB 2.0 RFP Submission
Universal Networked Objects
OMG TC Document Number 94–9–32

B. Modeling Techniques Summary

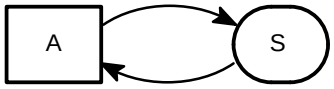
This chapter gives a short overview of the elements in the structure plans used by the Basis Modeling group. For detailed information see the report **Description Method – Modeling Principles and Structure Plans**.

B.1 Block Diagram

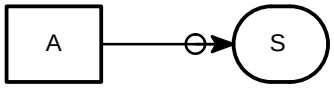
Block diagrams show the structure of a system, i.e. its *components* partitioned in active and passive parts and the *connections* between them. They show the *data flow* in a system.

Agent A is an active system component. An agent can be refined with any complex block diagram (i. e. can contain agents and storage places).

Storage place S is a passive system component. A storage place can be refined with a number of sub–storage places.

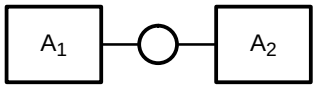


Agent A can write to and read from the storage place S. This means always an access to the complete contents of S.

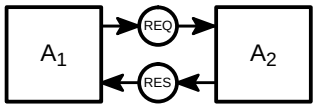


Agent A writes part of the storage place S. This symbolizes a selective write access.

Communication channel C is a non–saving, passive component. It can temporarily hold data and events for the duration of a communication.



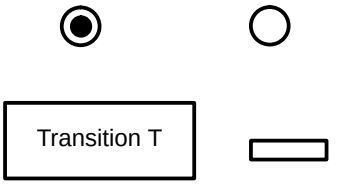
Communication between agent A_1 and agent A_2. The kind and direction of the communication are not specified closer.



Refinement of the communication channel between two agents. Agent A_1 sends requests (REQ) to agent A_2 and receives its responses (RES).

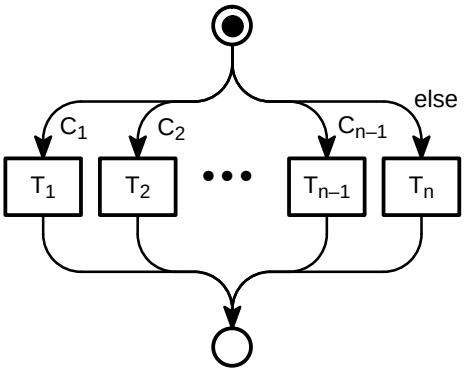
B.2 Control Flow

System states and events leading to state transitions are shown with *state transition diagrams*. Petri nets are extensions to state diagrams especially suitable for describing concurrent actions. They show the dynamical aspects of complex systems and consist of *places* and *transitions*. The places can hold *tokens* – they are marked – and the distribution of those tokens over the places describes the state of the system. A *firing* transition changes the system state. The firing of a transition occurs if and only if all input places are marked and all output places are free. The effect of the firing transition is that all output places will be marked.

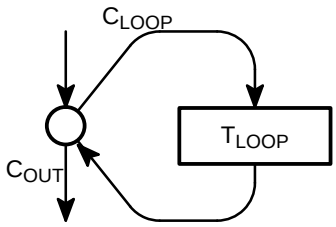


Marked and unmarked place. ”Start places” (no incoming arrows) are marked in Petri nets to show the beginning state.

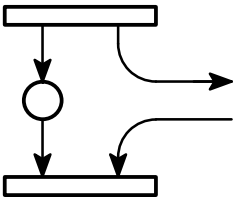
Transition T. The text in the rectangle describes the *activities* that are executed during firing of T. Transitions can be refined with sub–Petri nets. The empty transition on the right characterizes a null transition (i.e. nothing is done during firing, only the tokens are moved). Often used for syntactical reasons only.



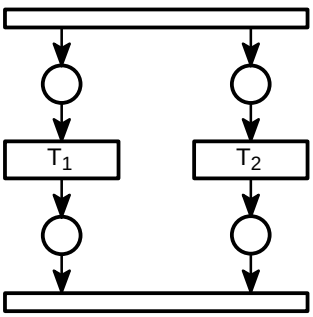
Conditional state transition (IF or SWITCH construct). If the system state fulfills one of the conditions C_1 – $C_{n–1}$ then the corresponding transition is executed; otherwise (else case) the transition T_n is processed.



Looped state transition (FOR, WHILE or REPEAT construct). Depending on the condition C_{LOOP} and C_{OUT} the transition T_{LOOP} is processed several times. Often only one of C_{LOOP}, C_{OUT} is given in the Petri net; the other one must be assumed ’else’.



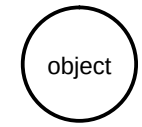
Upper transition causes jump to another process with a guaranteed return to the lower transition (e.g. subroutine call).



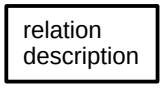
Processing of *concurrent* transitions T_1 and T_2.

B.3 Entity/Relationship Diagram

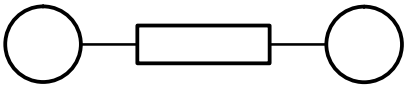
E/R diagrams visualize *object classes* (often called *entity types* or *object sets*) and *relationships* between them.



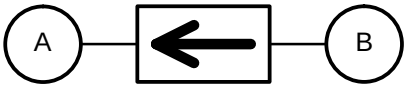
Set of objects. The text describes one element of the set.



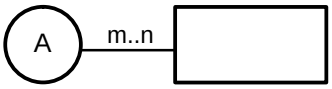
Relation symbol. Optional text describes the relation.



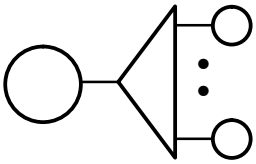
Relation between two object sets.



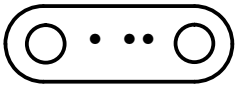
1 : n relation between object sets A and B. Each B is associated with exactly one A (often called hierarchy relation or parent–child relation with the elements of B as the children).



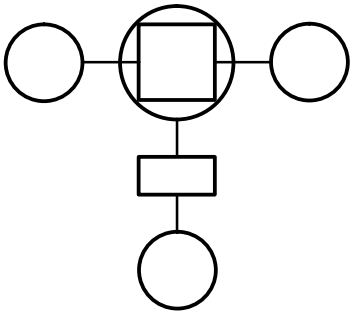
An element A of the object set occurs at least m times and at most n times in the relation.



The object set on the left side is divided into the disjoint sub–sets on the right side (*partition*).



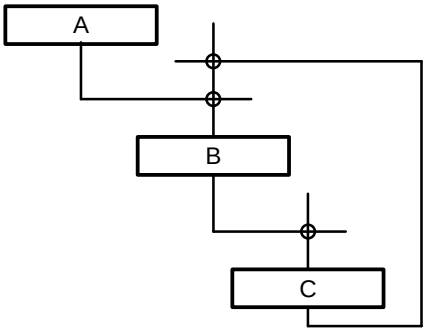
Another representation of a partition of an object set.



Objectification of a relation. The relation between the two upper object sets is viewed as an object. It has a relation with the lower object set.

B.4 Layered Structure Diagram

This type of diagrams describes a certain relation between objects of *one* class, e.g. import/ export, uses/provides or calls/is called relation between modules, functions or other components.



A, which might be a module or a function, requests services from B. The vertical line on the lower side of a rectangle means a service request and the vertical line connected to the upper side of a rectangle symbolizes the providing of a service. The circles connect pairs of correlated objects. The example shows recursion: If the diagram is to be interpreted as a function call layering, then function A calls function B. Function B in turn calls function C which recursively calls B.

Index

F

G

H

I

L

M

N

O

P

R

S

T

U

W

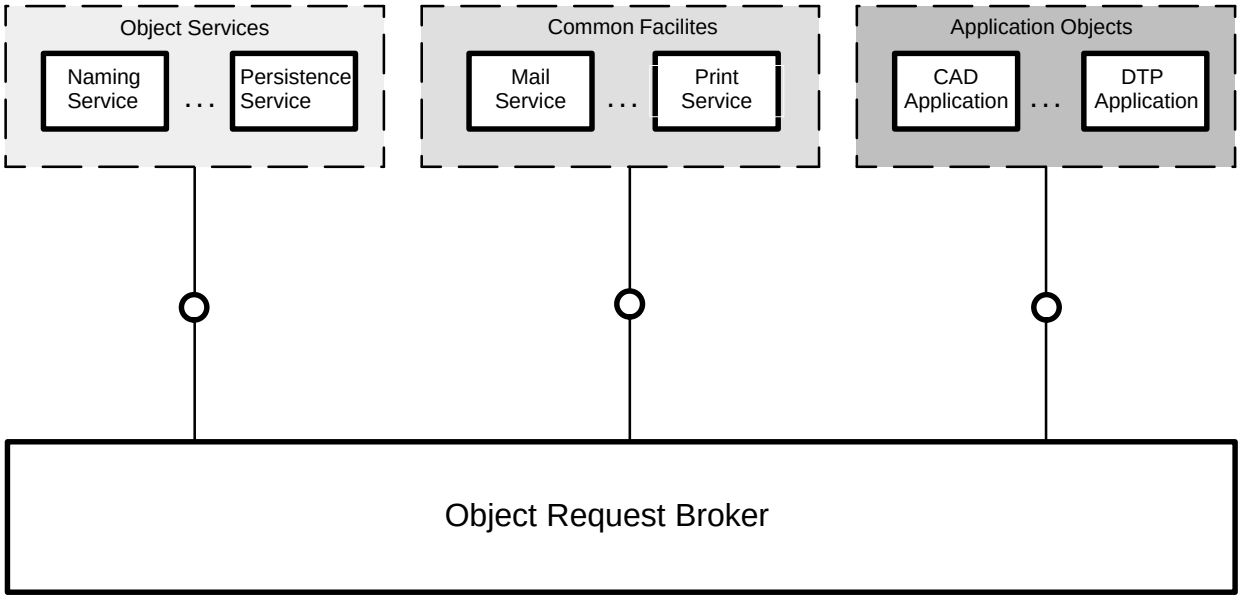


Figure 1

Block Diagram **Reference Model for the Object Management Architecture**	
OMG Concepts – OMA and CORBA	
Author: ra	Date: Apr 10, 1995

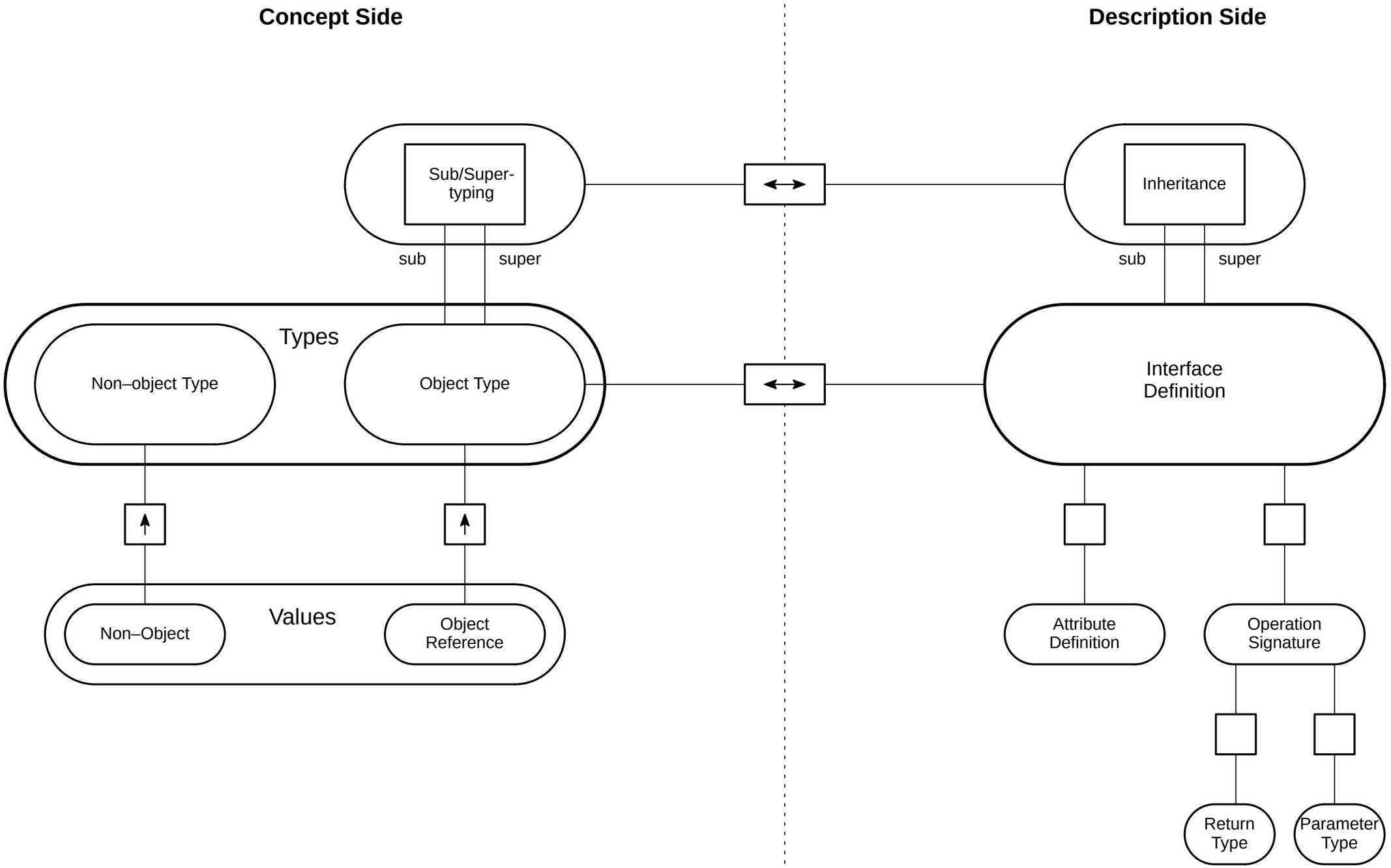


Figure 2

ER Diagram **Object Model**	
OMG Concepts – OMA and CORBA	
Author: ra	Date: April 1995

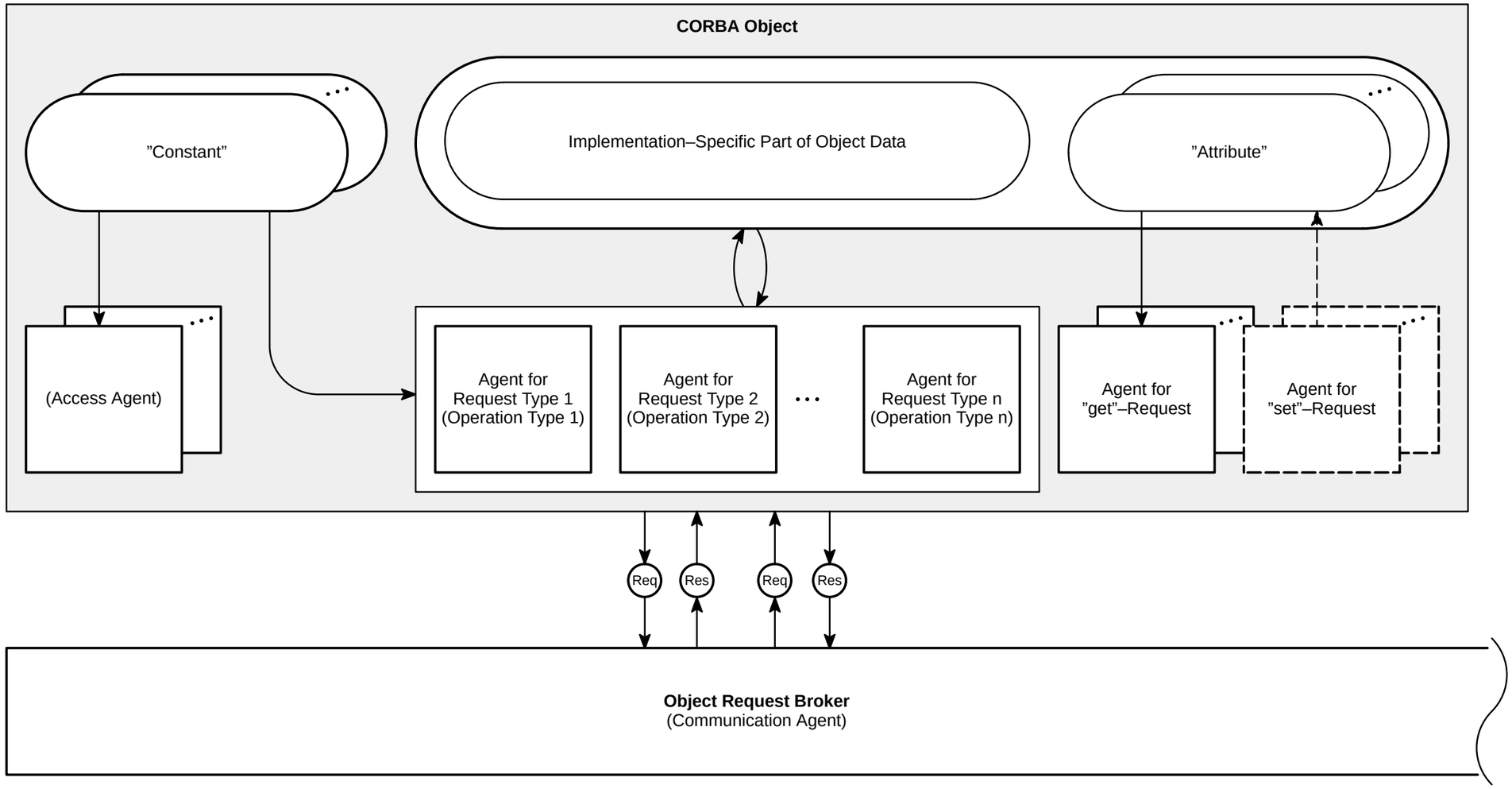


Figure 3

Block Diagram **Structure of a CORBA Object**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: May 1995

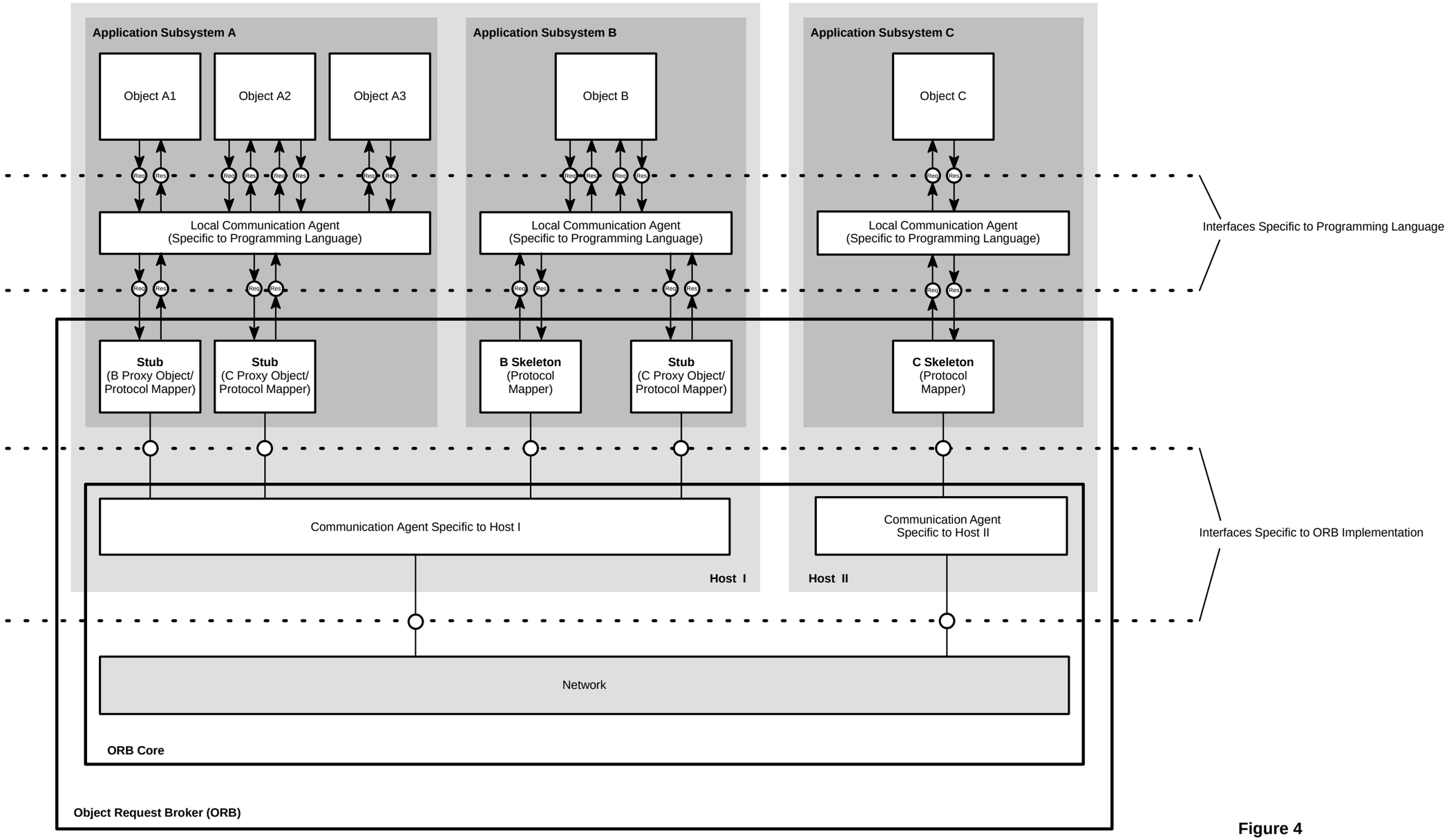


Figure 4

Block Diagram **A CORBA System**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: May 1995

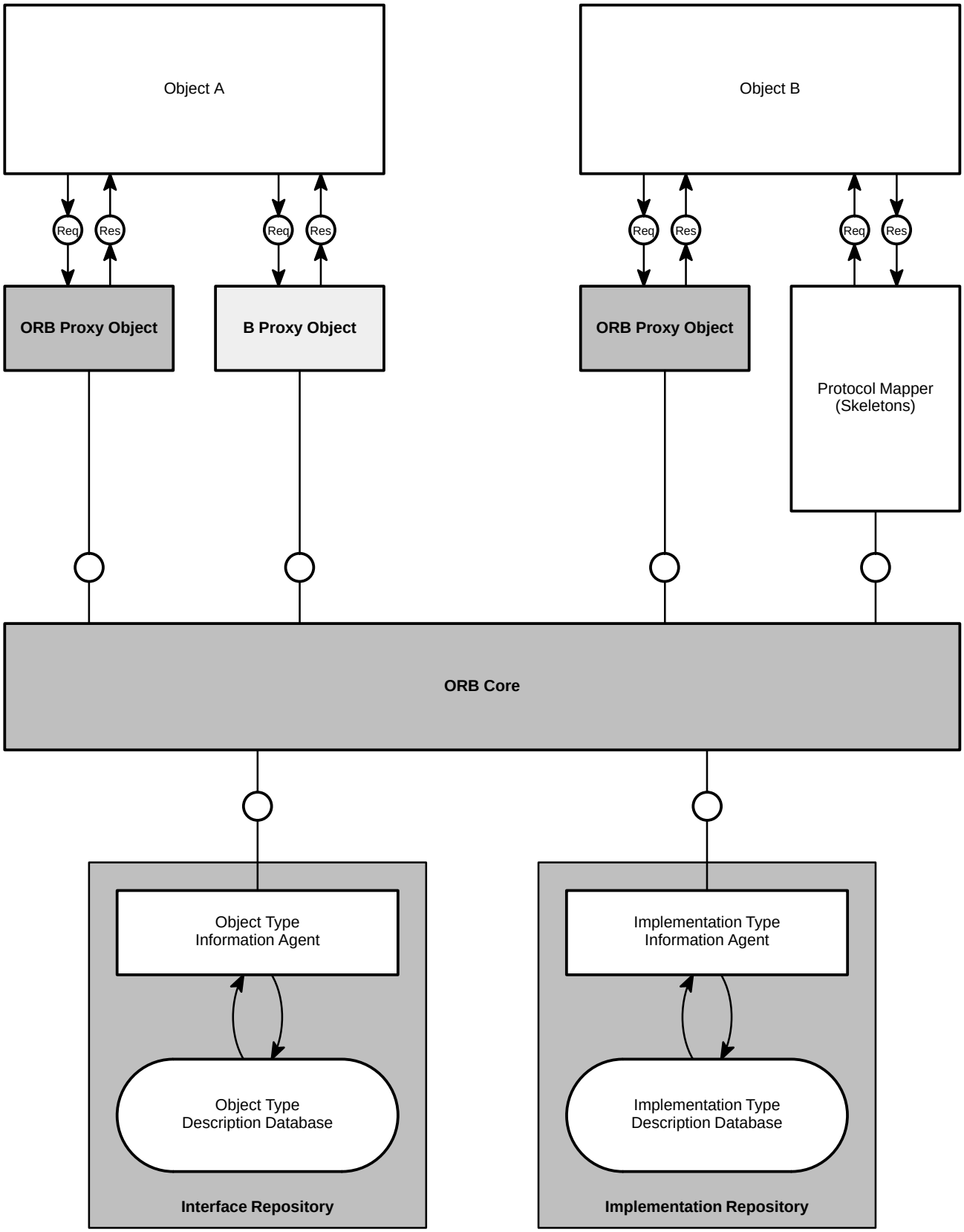


Figure 5

Block Diagram **ORB Interface**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: March 1995

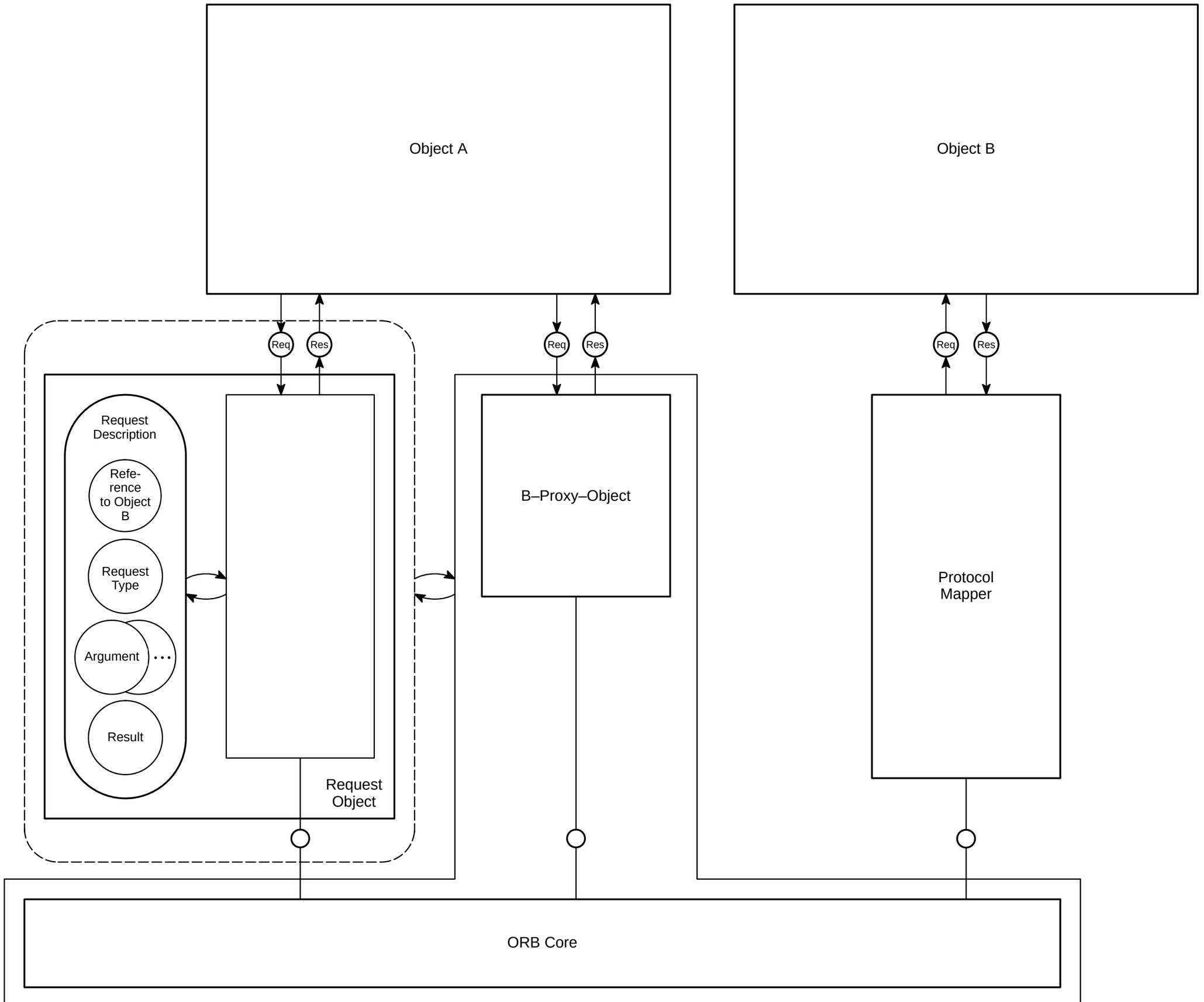


Figure 6

Block Diagram **Dynamic Invocation:**	
OMG Concepts – OMA	
Author: ta	Date

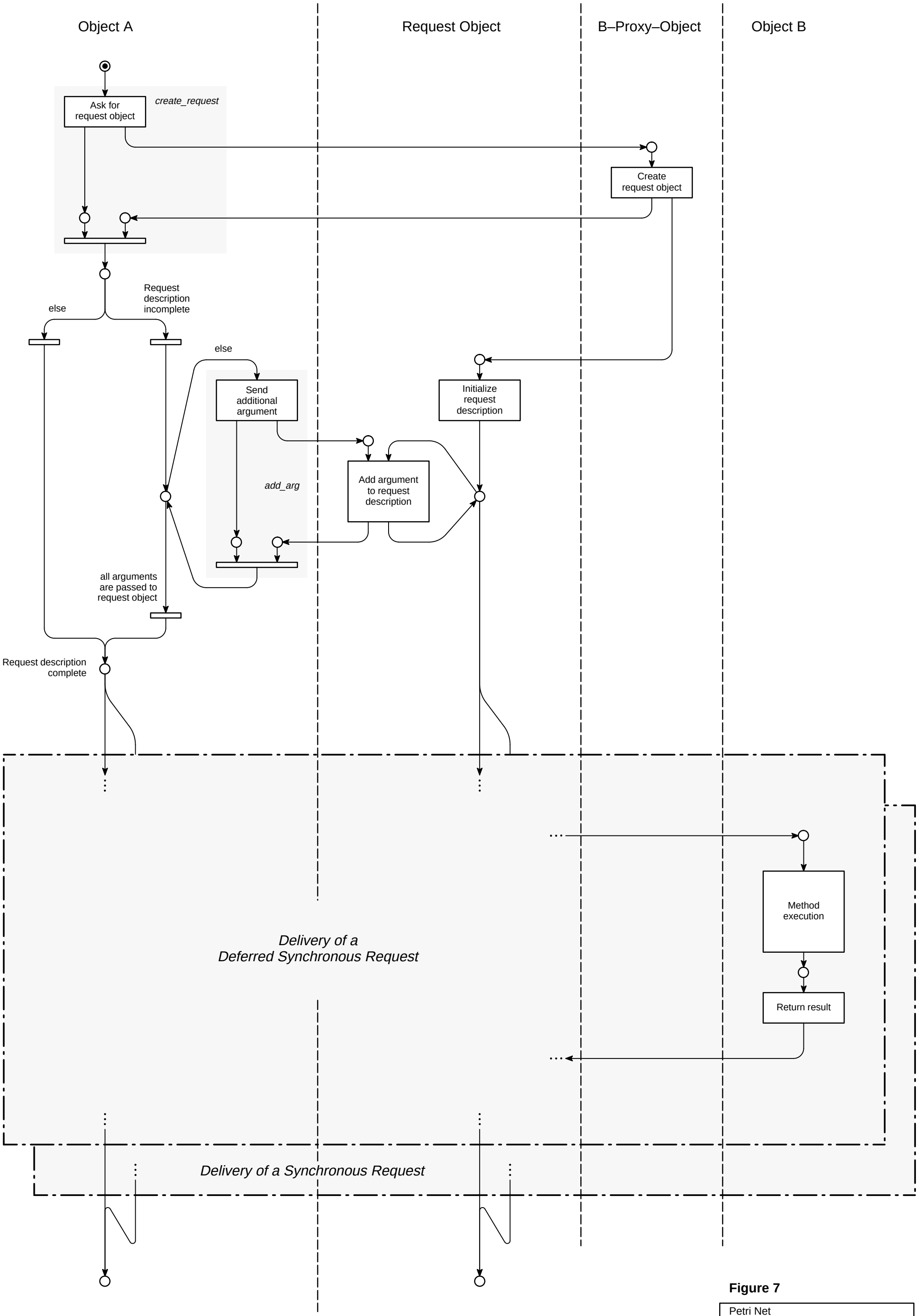


Figure 7

Petri Net **Dynamic Invocation: Overview**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

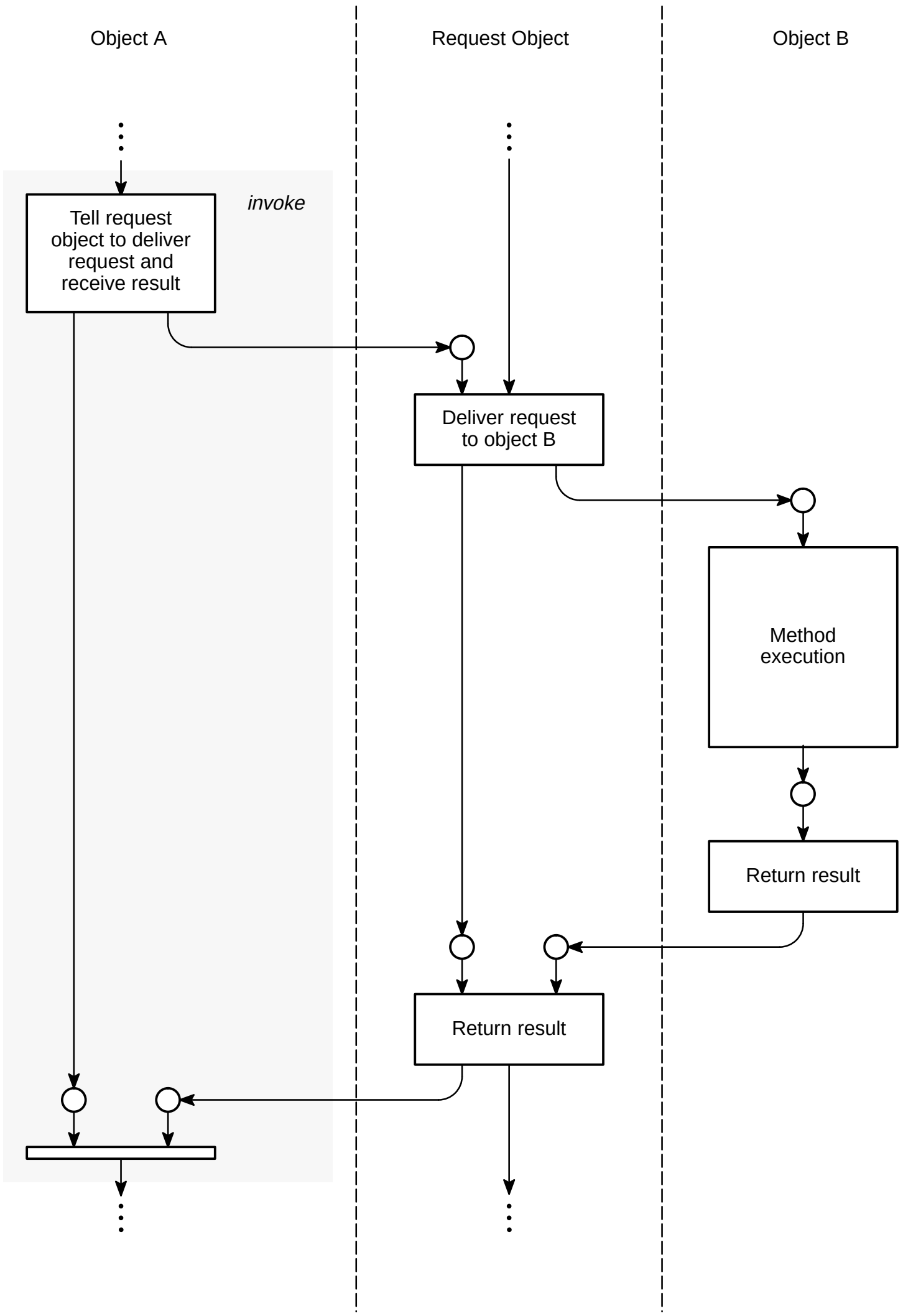


Figure 8

Petri Net **Dynamic Invocation: Synchronous Request**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

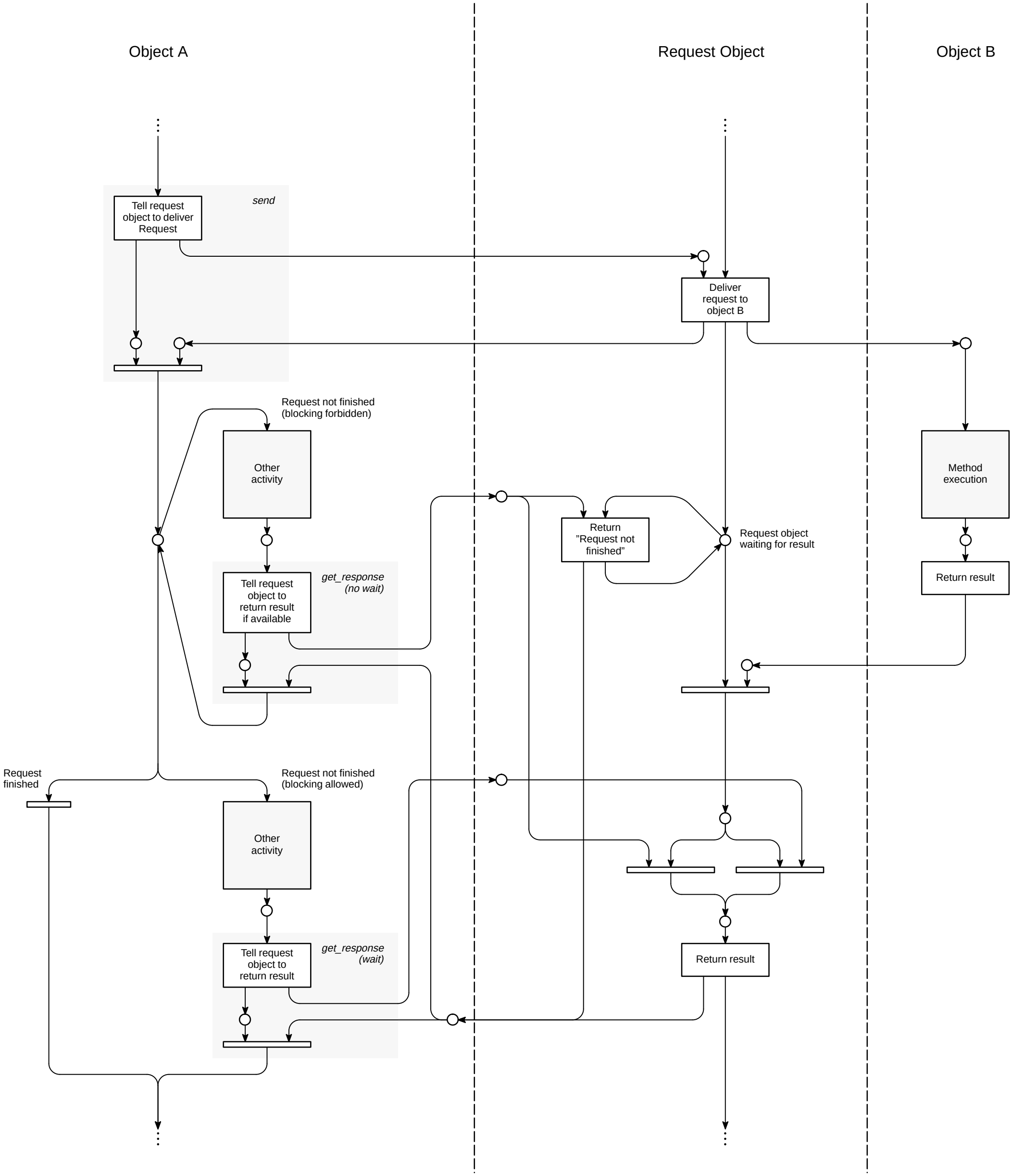


Figure 9

Petri Net **Dynamic Invocation: Deferred Synchronous Request**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

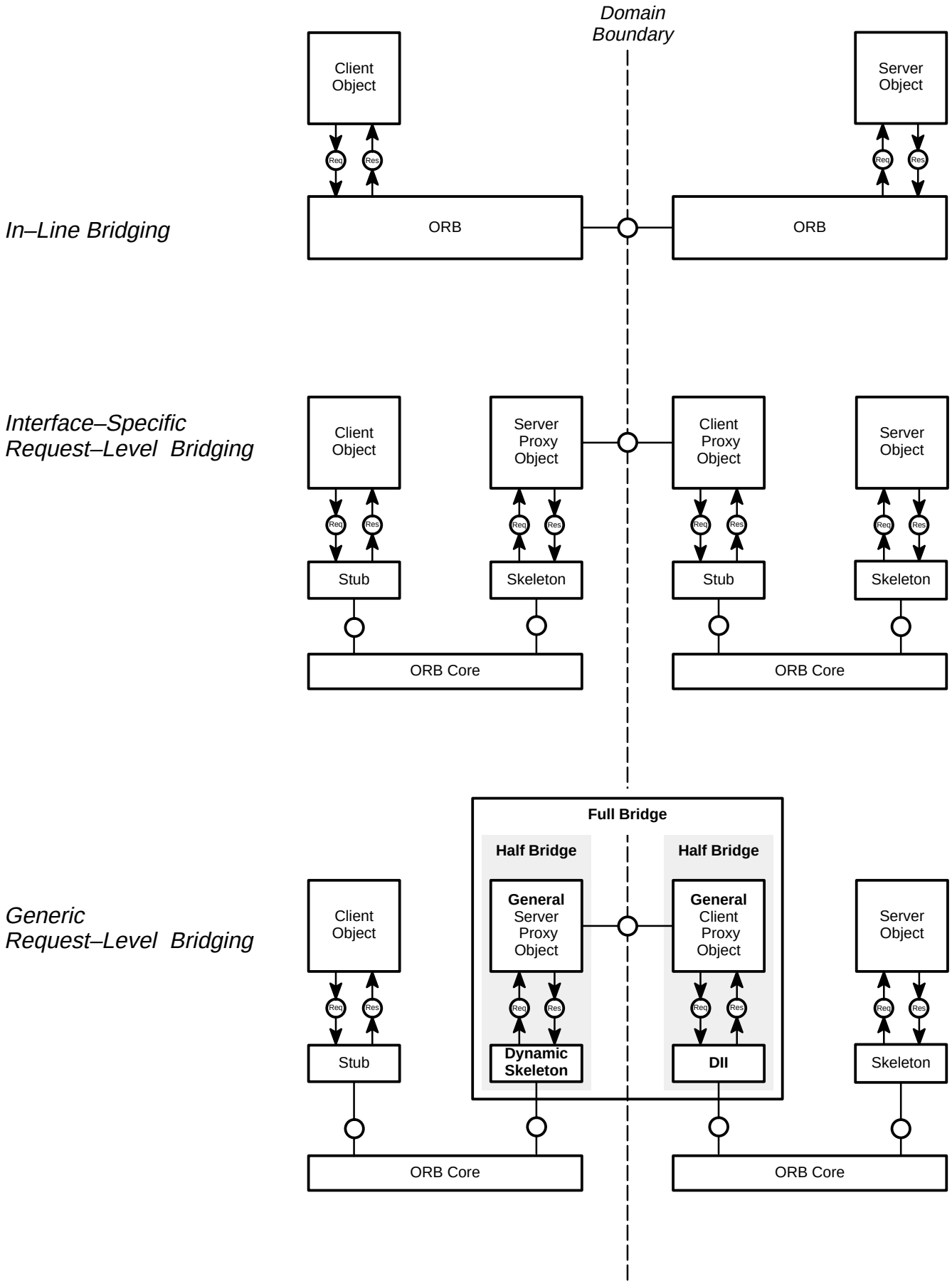


Figure 10

Block Diagram **Bridging Possibilities**	
OMG Concepts – OMA and CORBA	
Author: Ta	Date: Feb 2, 1996

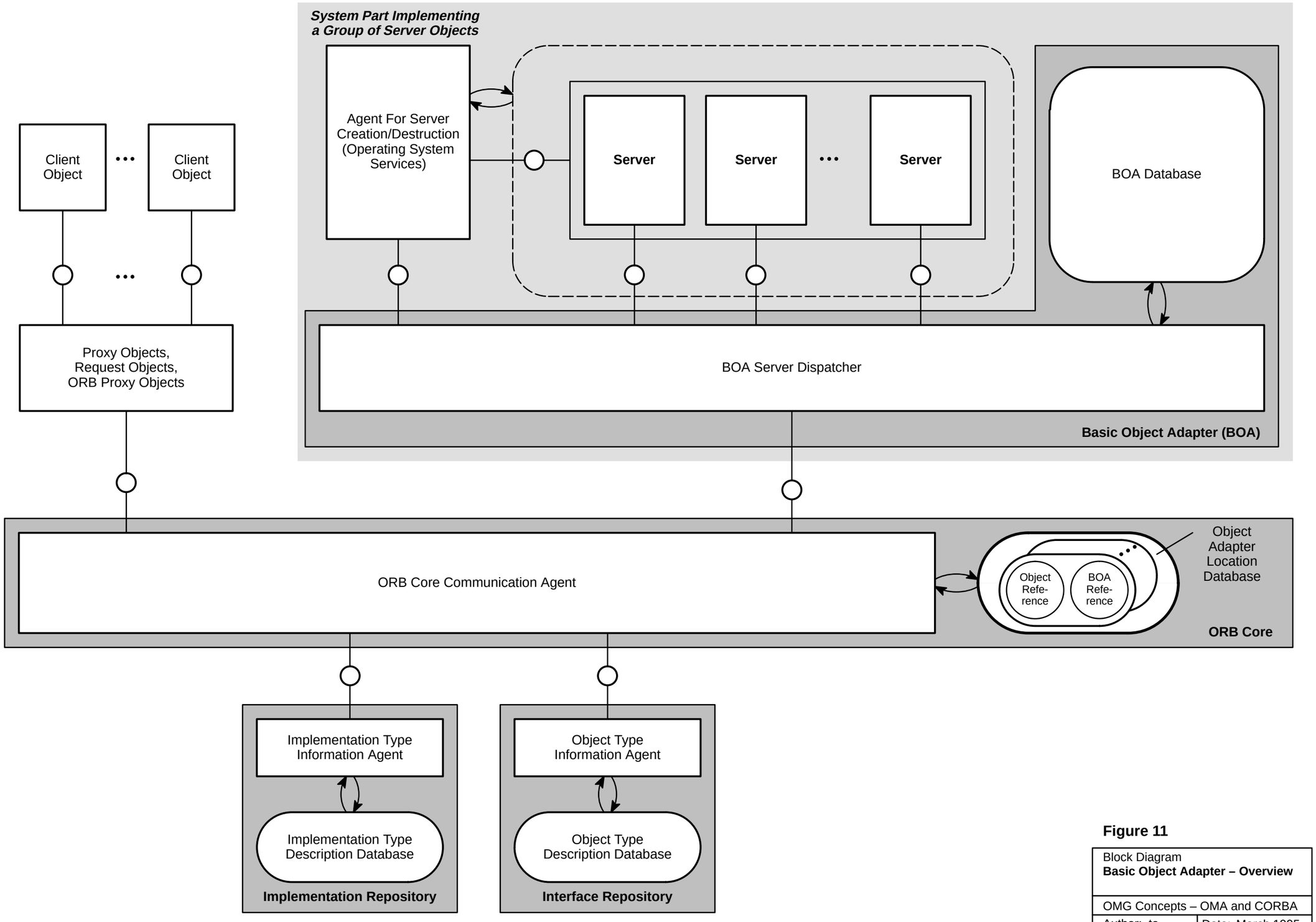


Figure 11

Block Diagram **Basic Object Adapter – Overview**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: March 1995

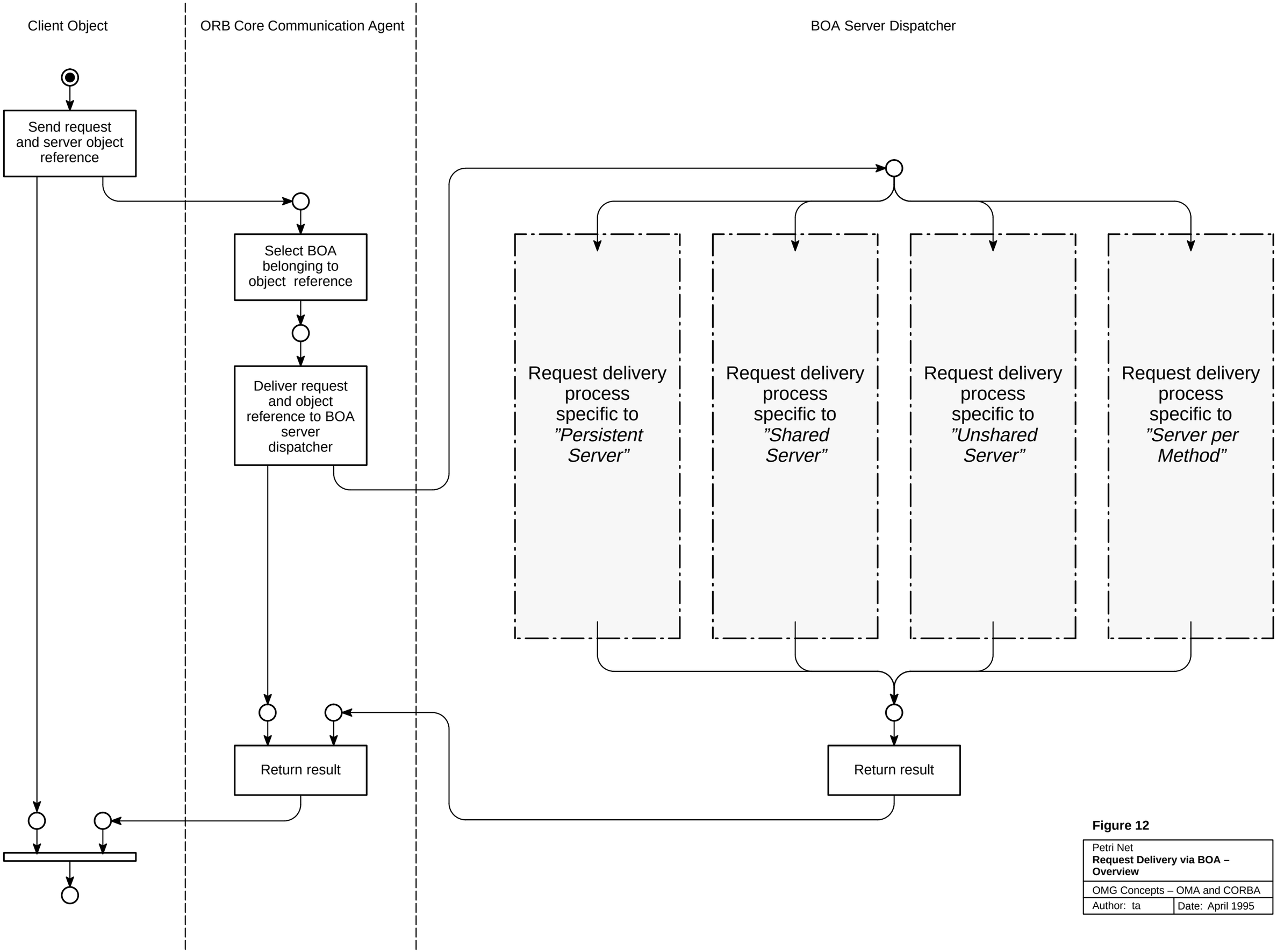


Figure 12

Petri Net **Request Delivery via BOA – Overview**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

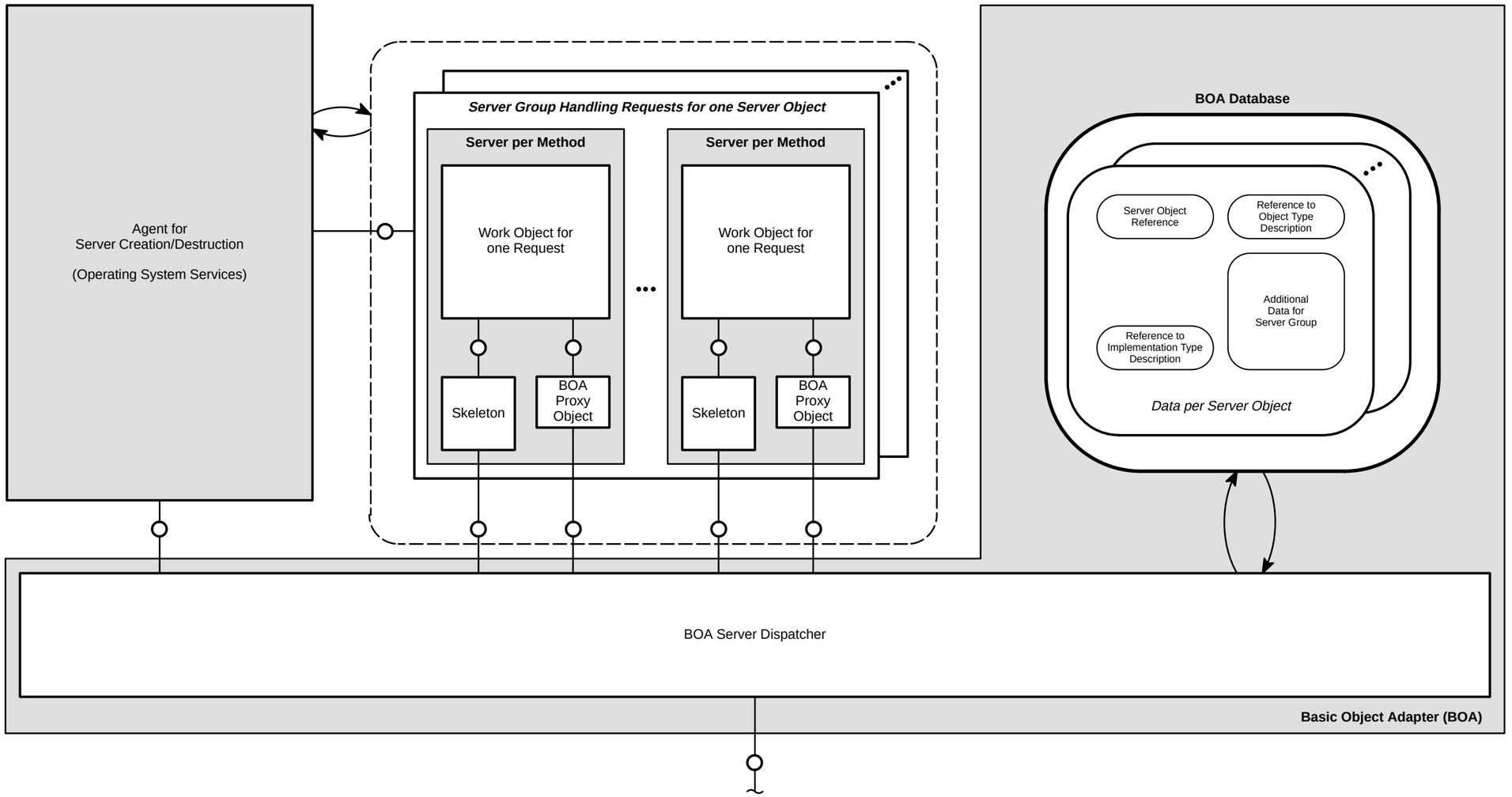


Figure 13

Block Diagram **BOA: ”Server per Method”**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

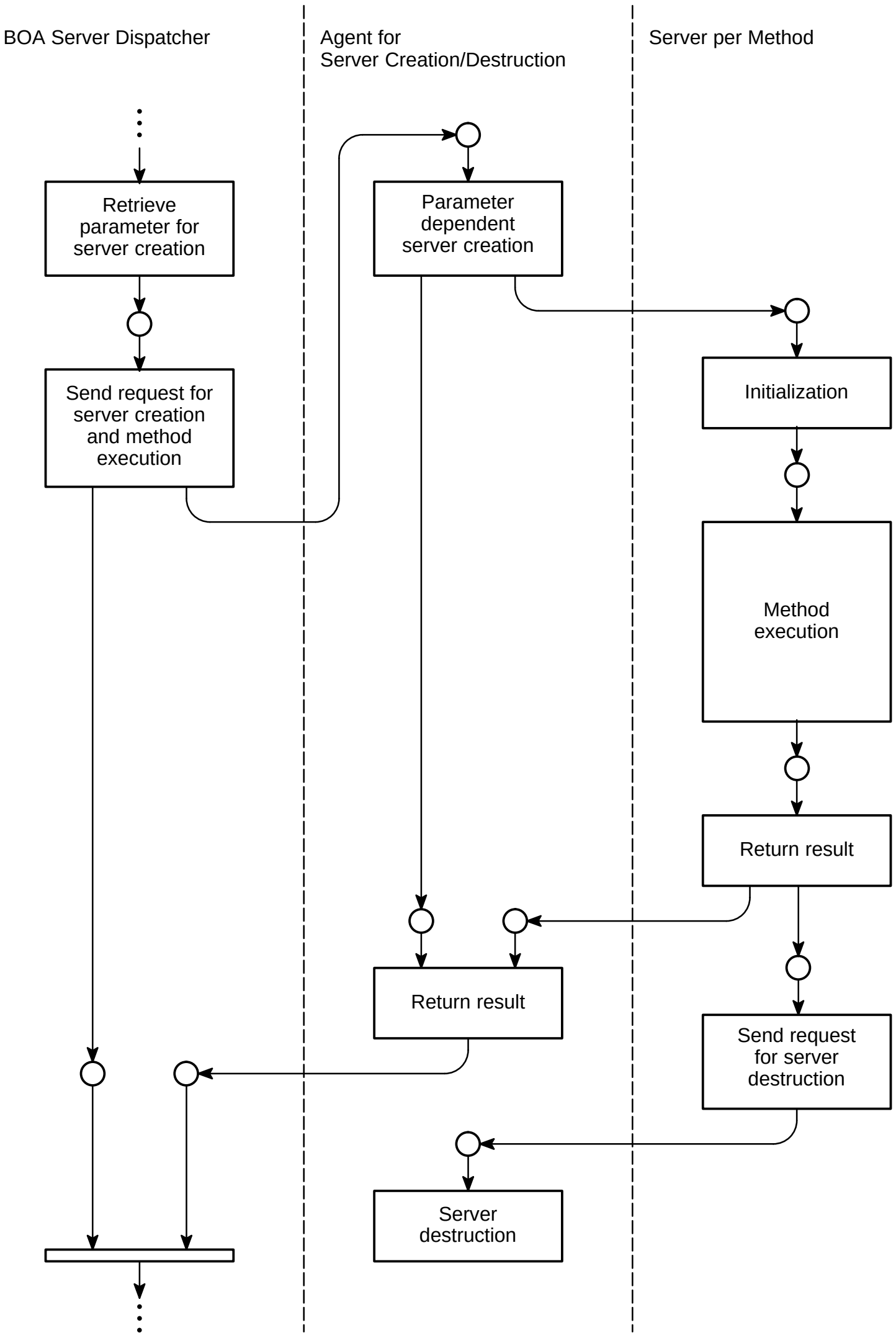


Figure 14

Petri Net **Server per Method: Preparing and Execution of Methods**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

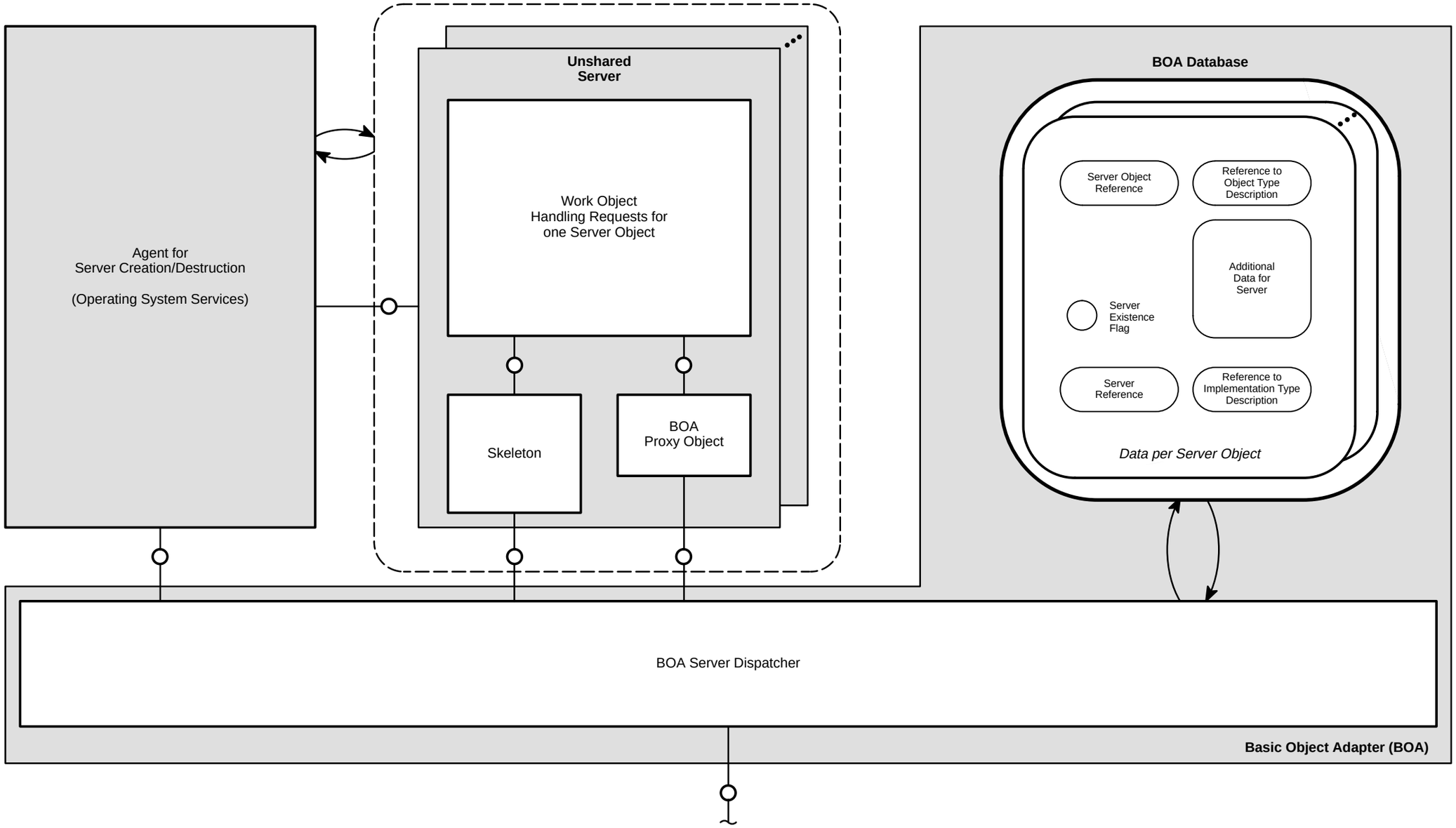


Figure 15

Block Diagram **BOA: "Unshared Server"**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

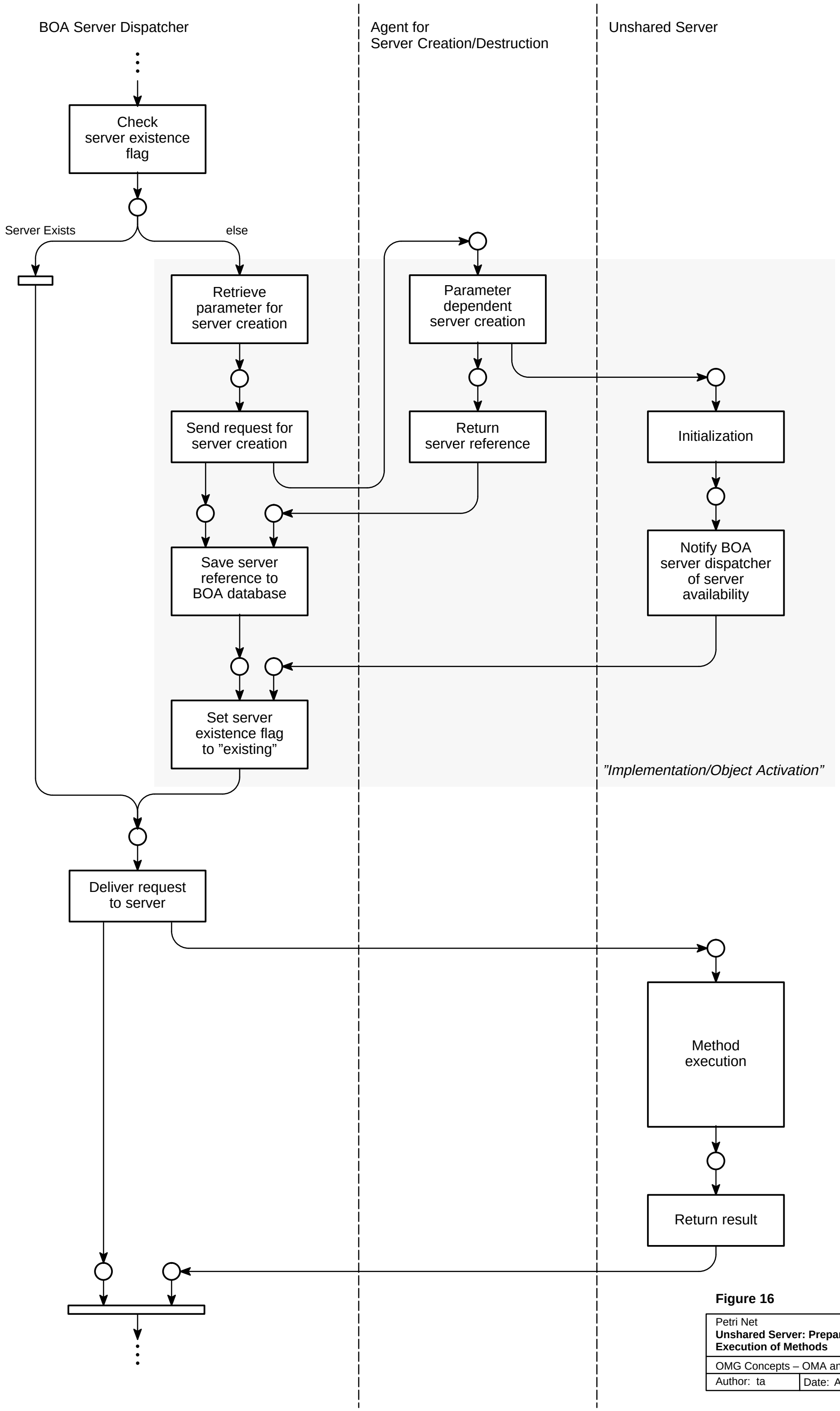


Figure 16

Petri Net **Unshared Server: Preparing and Execution of Methods**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

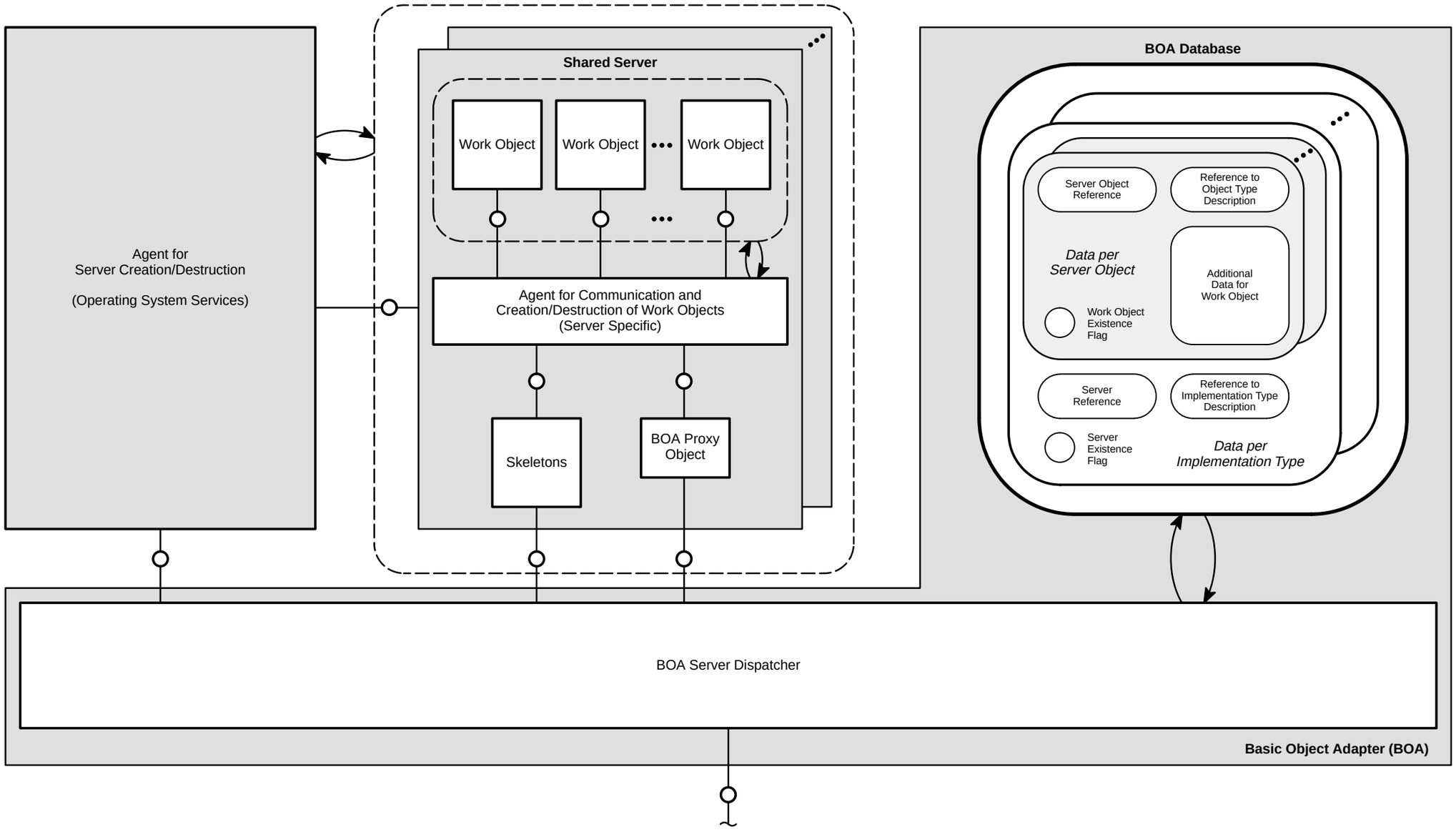


Figure 17

Block Diagram **BOA: ”Shared Server”**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

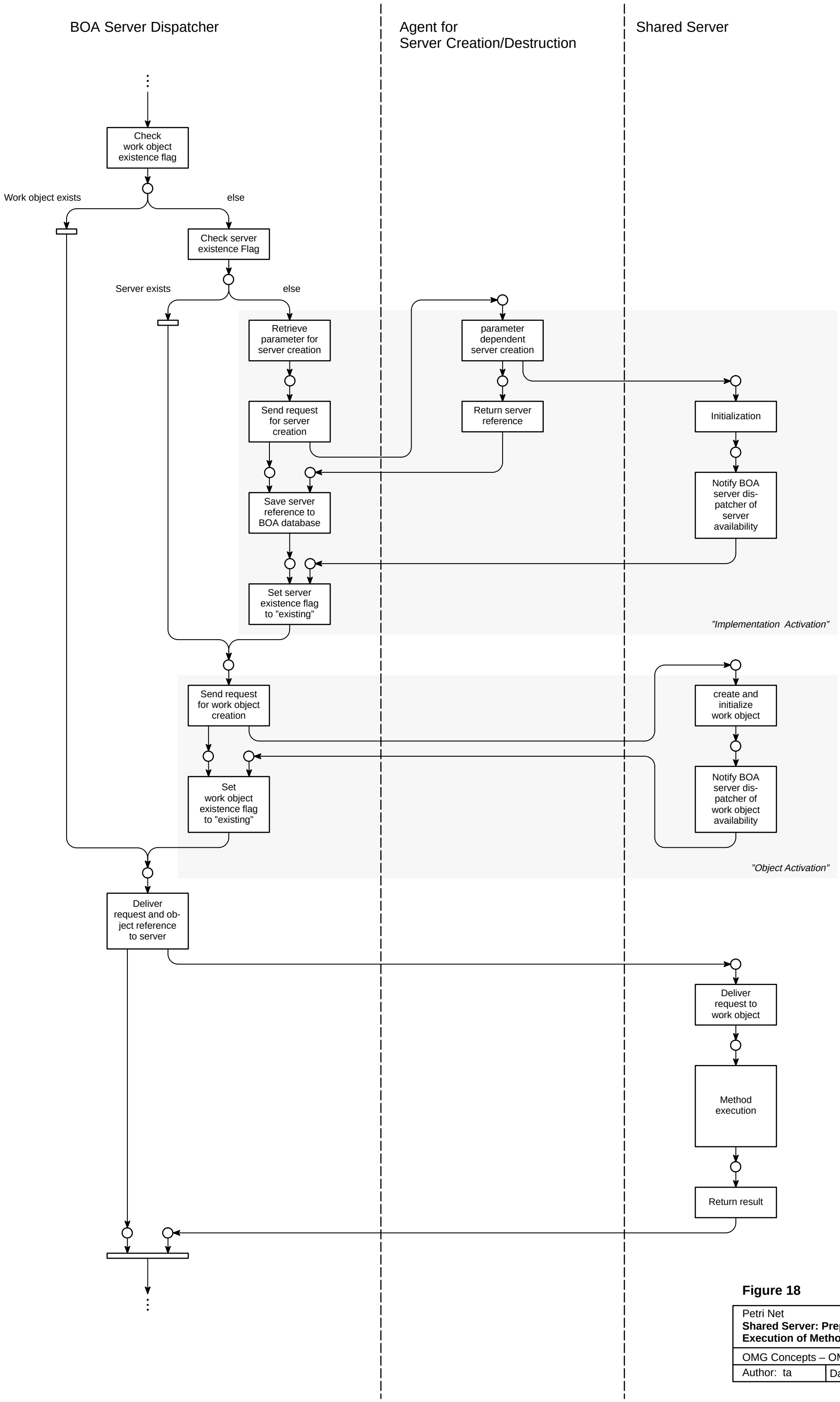


Figure 18

Petri Net **Shared Server: Preparing and Execution of Methods**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

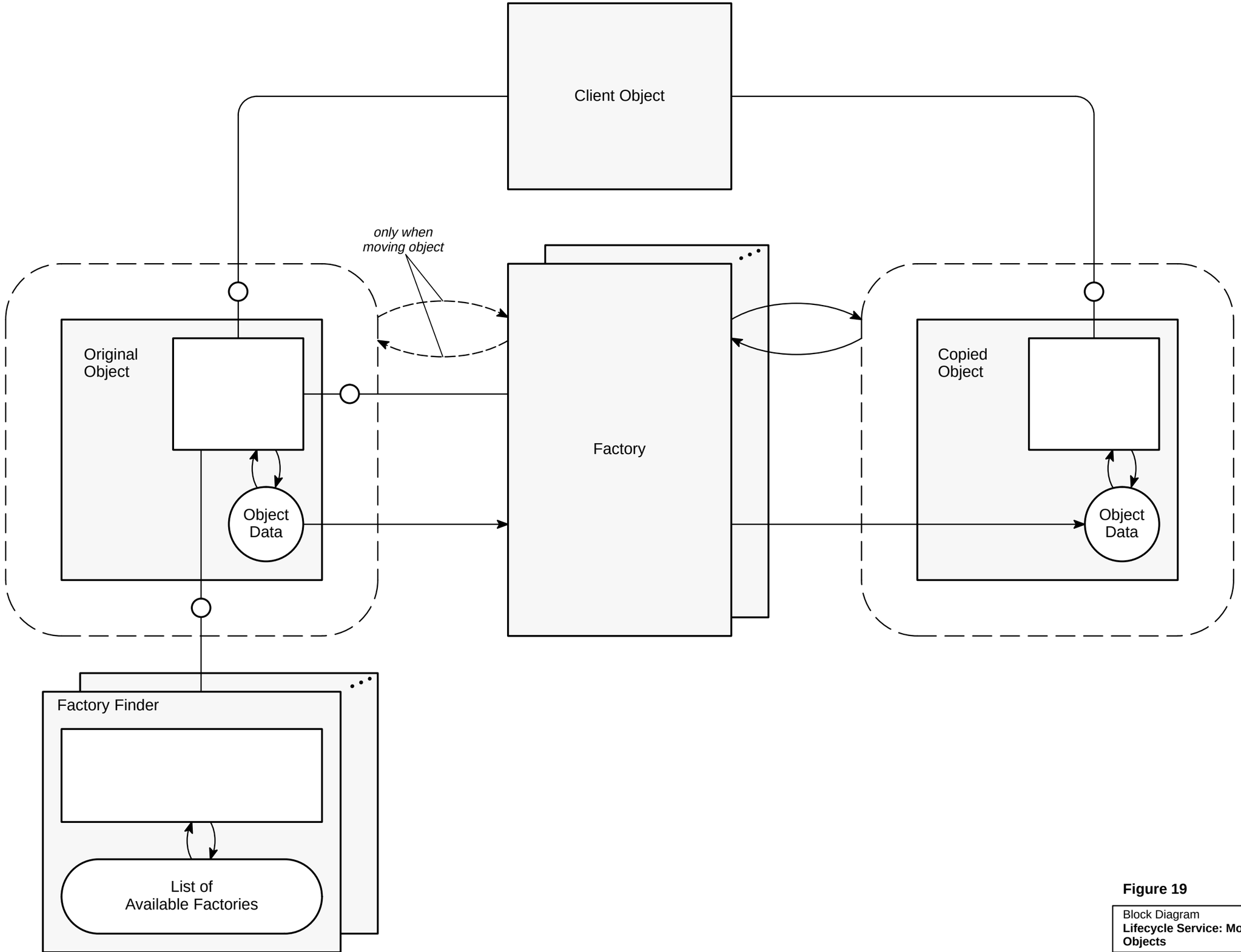


Figure 19

Block Diagram **Lifecycle Service: Moving/Copying Objects**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: March 1995

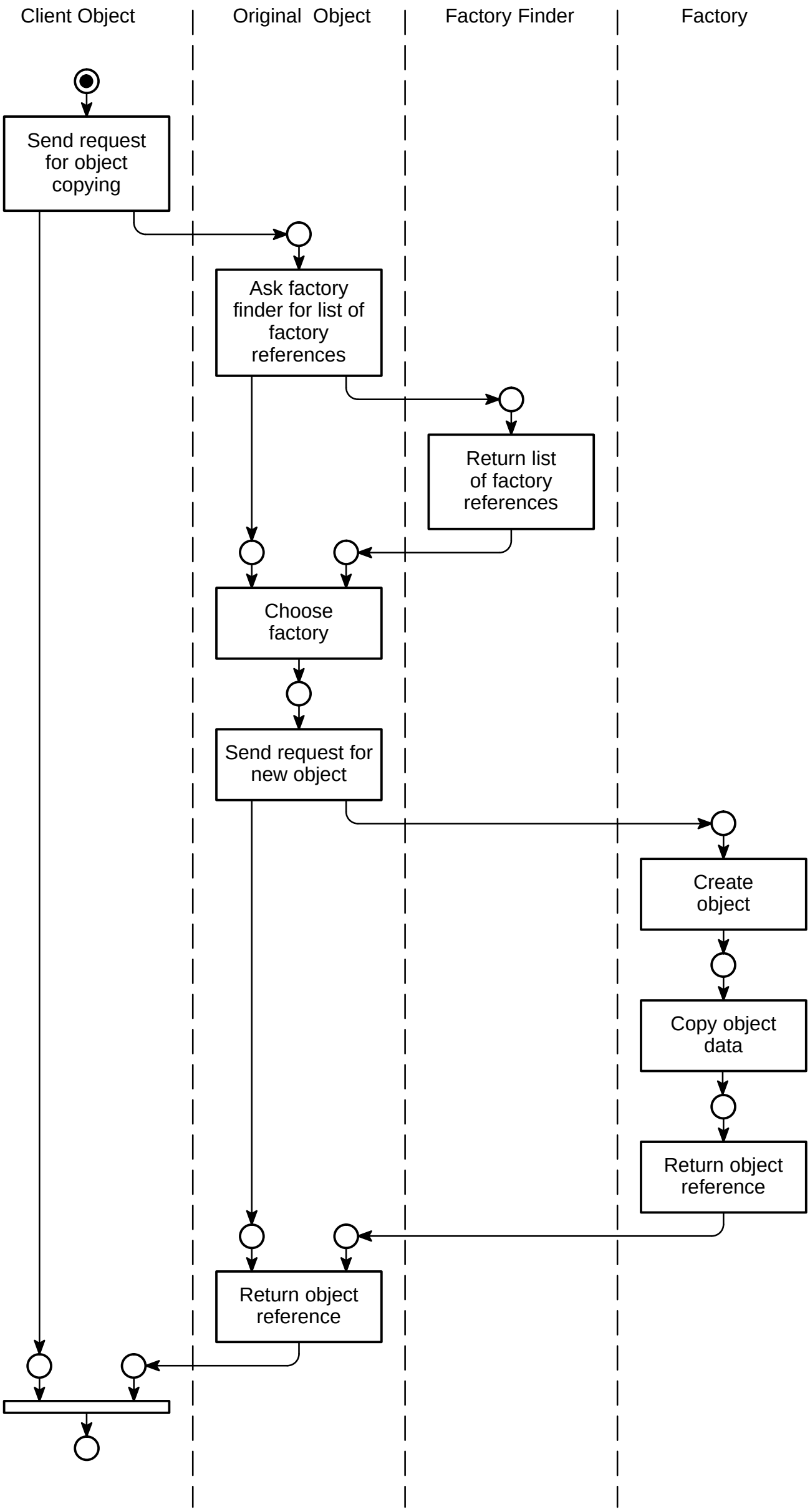


Figure 20

Petri Net **Lifecycle Service:** **Object Duplication**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

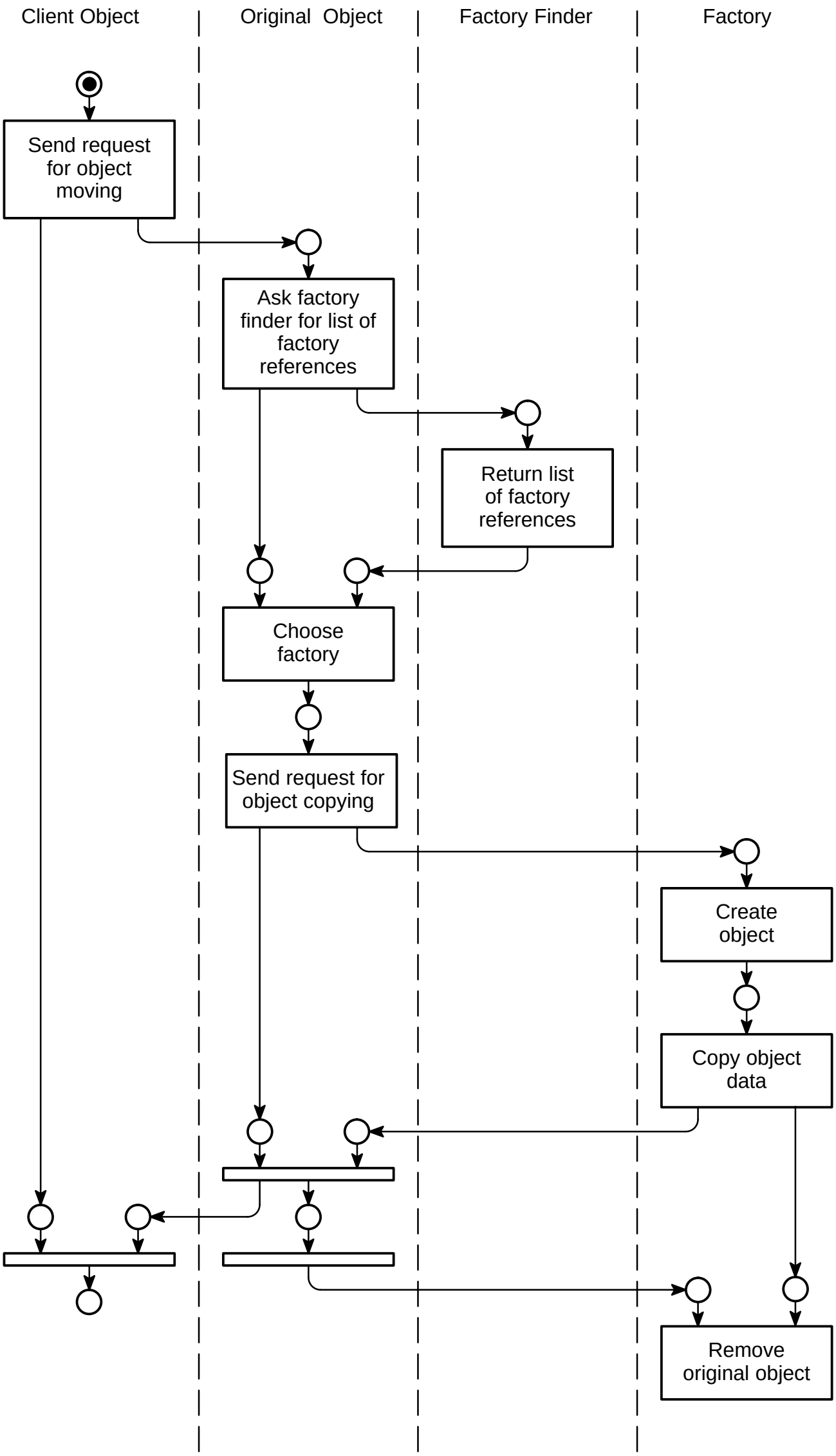


Figure 21

Petri Net **Lifecycle Service: Object Moving**	
OMG Concepts – OMA and CORBA	
Author: ta	Date: April 1995

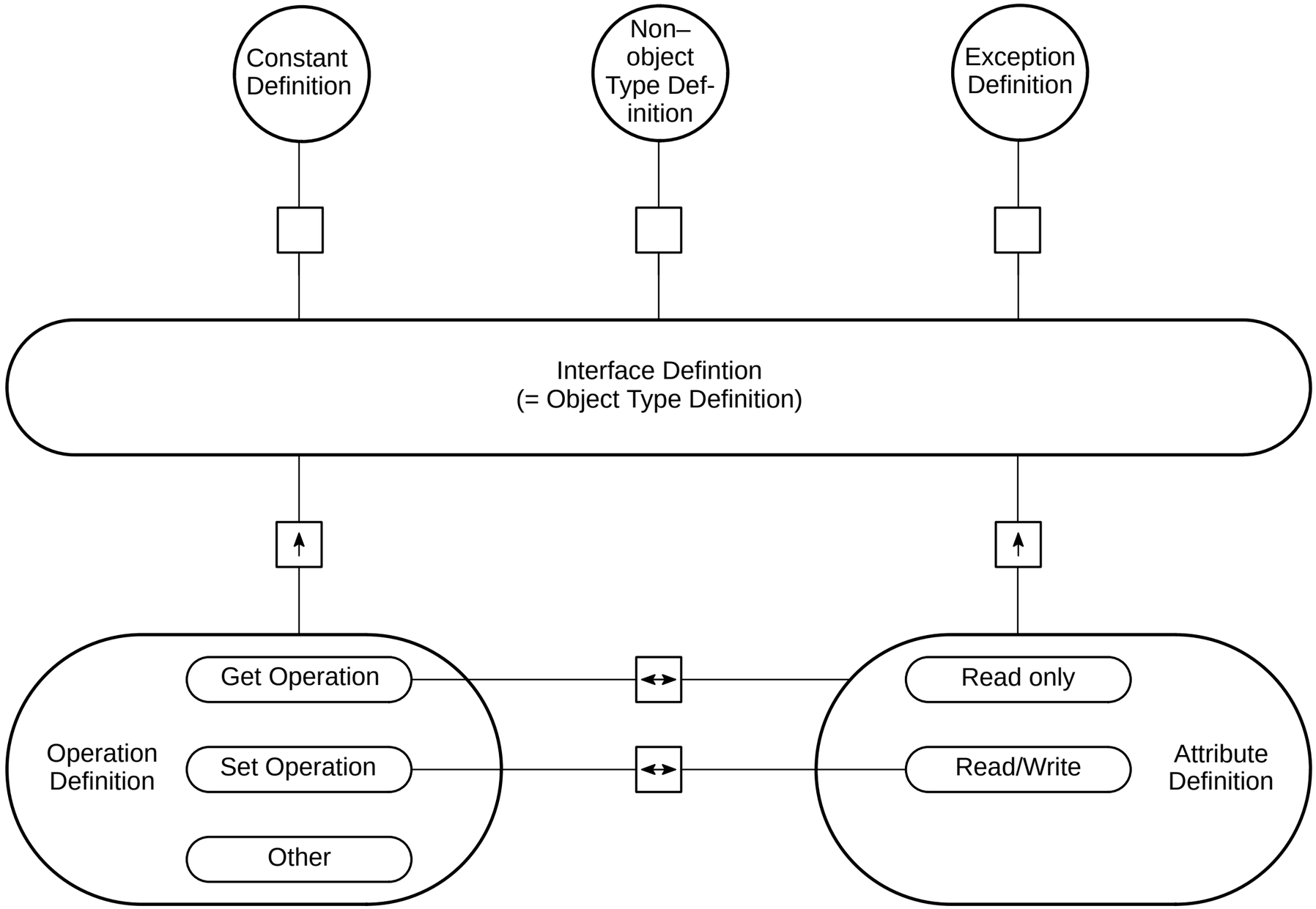


Figure 22

ER Diagram **IDL Definitions**	
OMG Concepts – OMA and CORBA	

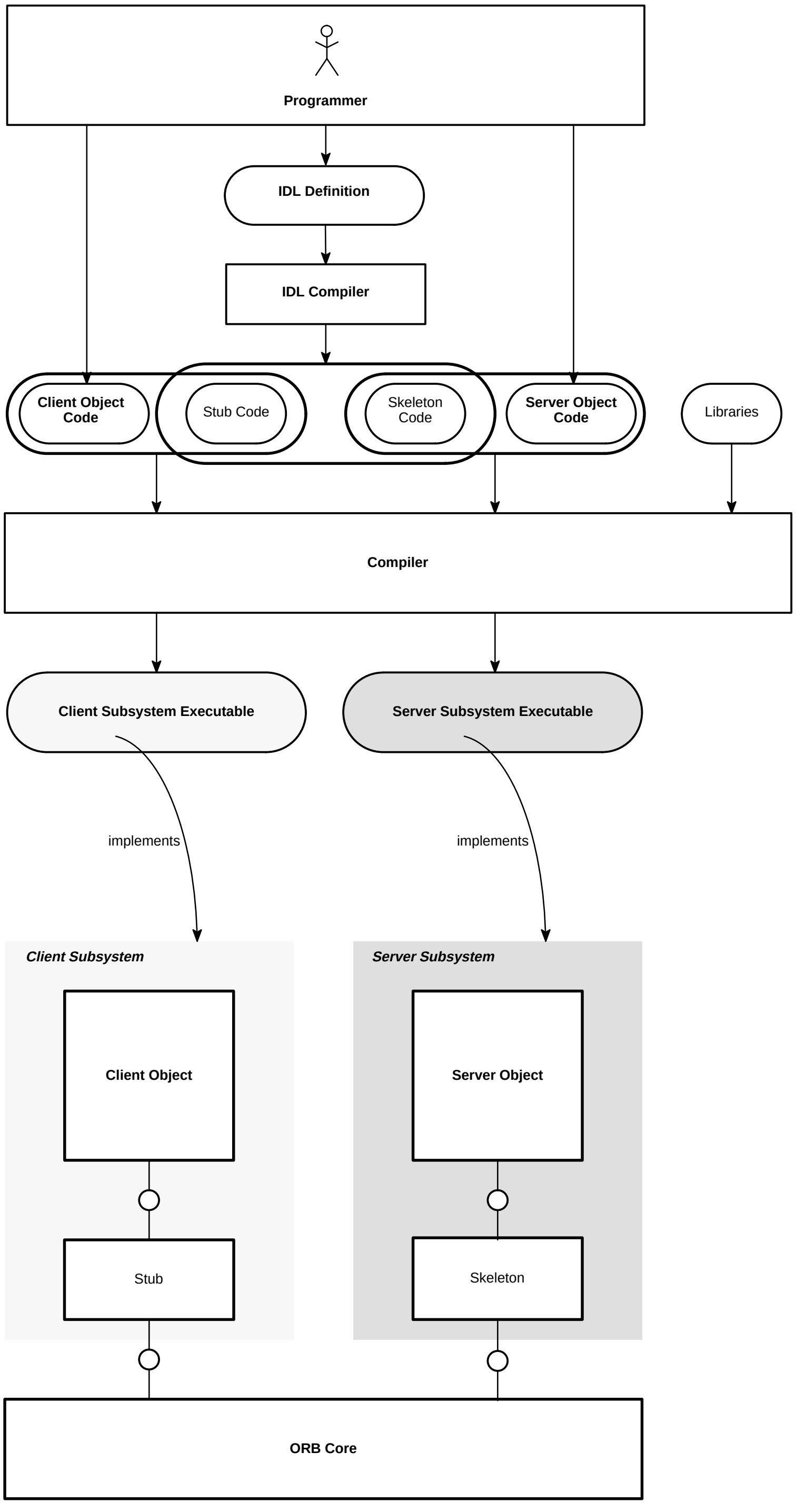


Figure 23

Block Diagram **IDL Compilation Process**	
OMG Concepts – OMA and CORBA	
Author: ra/ta	Date: May 1995