University of Bologna

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Engineering Bologna Campus

Class of

Computer Networks M

MIDDLEWARE - CORBA

Antonio Corradi

Academic year 2015/2016

MIDDLEWARE: CORBA

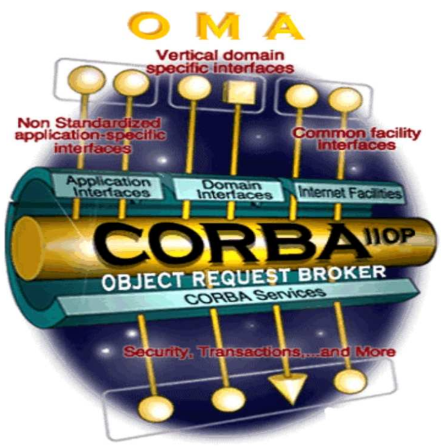
OMG- Object Management Group

CORBA started in 1989 with **440 company** Microsoft, Digital, HP, NCR, SUN, OSF, *etc.* with main objective to create a **use** and **management system** of a **distributed architecture**

Common **O**bject **R**equest **B**roker **A**rchitecture
CORBA standard v1 ⇨ 1991, v1.2 ⇨ 1992
v2 ⇨ 1996, v3 ⇨ 2000

Orbix *SunOS Solaris, Iris, Windows NT, HP/UX, AIX, OSF/1, UnixWare*
DSOM *IBM*

General specification of an Object (component) Middleware to use in heterogeneous distribute systems not tied to a specific language



MIDDLEWARE: CORBA

STANDARD OPEN SYSTEM based on OBJECT models with heterogeneous components to implement **mutual** and **complete interaction** and **integration** between **such components,** inside **distributed environments also objects oriented (C/S model)**

CORBA requires:

- definition of a **language as service interface**
- definition and support to **objects interaction**
- **integration bus** for **different environments** objects **(ORB)**
- **interaction** between systems with different managers
- **different deployment languages** (**language mapping**)

The objective is to allow **services support** without posing **limits** on user application **lifecycle**

ARCHITETTURA CORBA

Common **O**bject **R**equest **B**roker **A**rchitecture **CORBA**, as a **common environment**, **O**bject **M**anagement **A**rchitecture, for multi-architecture and multi-language scenarios, with an optimal integration with legacy systems and best support for differentiated projects for server and clients

Object Request Broker (**ORB**) is the **heart** of the **architecture** and acts as a **broker of communication**, to allow both **static** and **dynamic** links (!?) between entities

ORB behave as an always available enabler and allows:

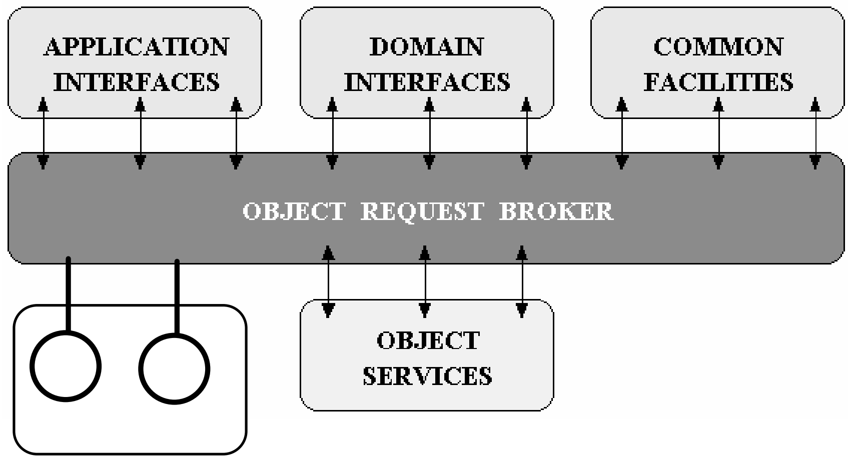
- control of **allocation** and **visibility** of objects
- control of **methods** and of **communication**
- control of **accessory services** always available inside OMA for every language mapping
- **simplified management** of every possible services

CORBA as third type middleware, infinite lifetime

CORBA come BUS

ORB is the **center** of **O**bject **M**anagement **A**rchitecture
ORB as a **bus center of an architecture** that aims at the integration among **every resources of an organization**

Every managed application objects can belong to **different environments** and must be able to **mutually communicate** without any need of **redesign**



Applications Object

Object Management Architecture

Other additional environment components

Common Facilities CF (horizontal)

Set of specific features

User Interface (client-site),

System Management, Information, Task (server-site)

Domain Interfaces (vertical)

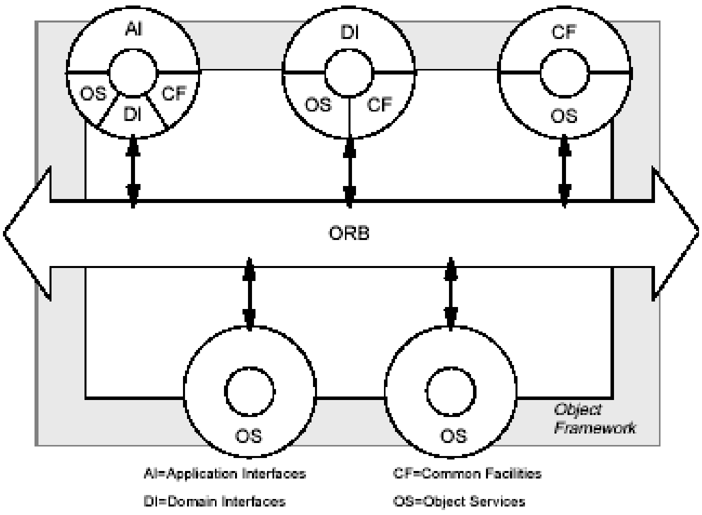
Features dedicated to application areas, for ex. manufacturing, telecommunications, electronic commerce, transportation, business objects, healthcare, finance, life science, …

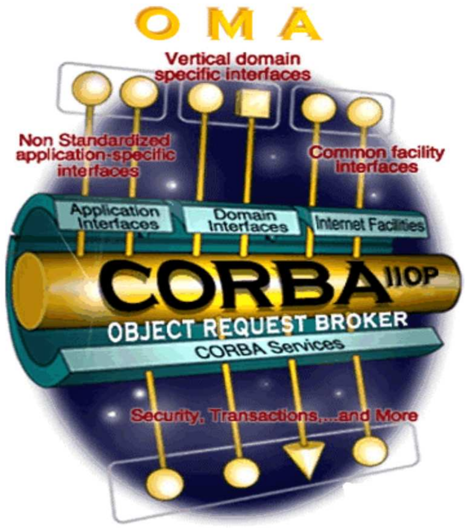
Application Interfaces

Non standard in any way and application-dependent

Object Management Architecture - OMA

Ambiente Object Framework





Object Management Architecture

Every component can connect to every other one, preparing link either before or during (if unknown before) execution, using the service of **one or more ORB (known dynamically)**

Set of **additional environment components**

Object Services or CORBA Services (*Common Mw Services*)

Some operations are basic for object

- ***naming*** and ***trading*** service (compatible with OO)
- ***event*** and ***notification*** service (less Object-Oriented)

In addition to further operations (or services)

For lifecycle management, relational, transactional, concurrency control, security, …

CORBA COMPONENTS

The essential components of OMA architecture, i.e., CORBA, associated to an ORB:

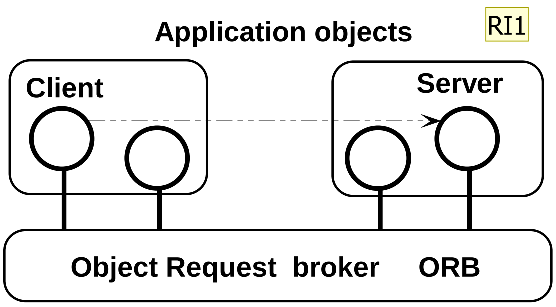
- **Object Request Broker**	(ORB)
- **Interface Definition Language**	(IDL)
- **Basic Object Adapter (e POA …)**	(BOA e POA)
- **Static Invocation Interface**	(SII)
- **Dynamic Invocation Interface**	(DII)
- **Interface e Impl. Repository**	(IR e IMR)
- **Integration Protocols**	(GIOP)

Those components are at very different level

ORB CONTINUOUS SUPPORT

Object Request Broker (ORB) must **coordinate invocation of local and remote services (dynamically)**

- Identify **implementation of an abject** as a servant to requests (object location)
- prepare the **servant** to receive the request - via *adapter* (object creation, activation & management)
- transfer **the request** from the client to the servant
- return **reply** to client

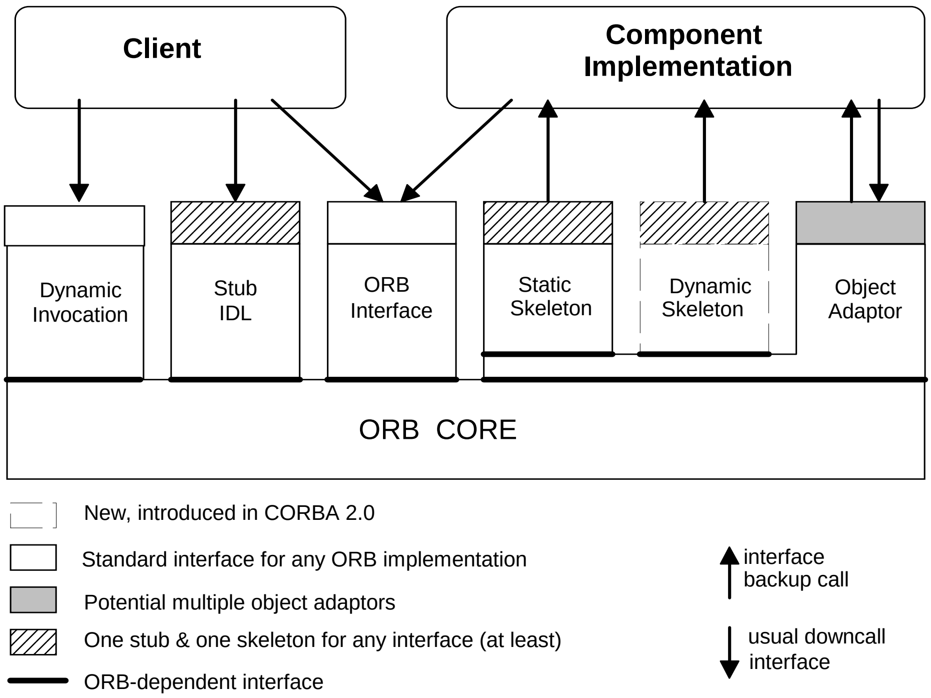


Diapositiva 10

RI1 Immagine con scritte in italiano
Raffaele Ianniello; 12/04/2016

CORBA: DYNAMIC VISION

Elements in action: overall user view



view of CORBA 1.x

not changed until CORBA 3

COMMON LANGUAGE in CORBA

Interface **D**efinition **L**anguage (**CORBA IDL**) must **identify** and **coordinate requested and offered services, local** and **remote (for** either **static** or **dynamic interactions)**

- Both **servants** and **clients** can **identify themselves** to **make themselves** mutually **known**
- Both **operations request** and **service** offers can be **optimally associated**
- CORBA reuse the experience from already developed and available **IDLs** for defining a general multi-language IDL

Unfortunately IDL prescribe predetermined identification and link and statically recognized (CORBA static binding)
And if we want bindings unknown at development time?

CORBA IDL for MULTILANGUAGE

Interface **D**efinition **L**anguage (CORBA IDL) **coordinates requested and offered services identification**, **with different languages**

```
interface Factory  //OMG IDL
  {        Object create(); // CORBA object or reference
  };
```

This interface permits to refer an object of type Factory (IDL) and to request the **create** operation (without **in** or **out** parameters) that returns a generic CORBA object (`type Object`, that is a reference to the object of interface `Object`)

IDL makes possible to define **new interfaces and new** general **types** and **abstract**, by need, to make them available and registered, and eventually concretely usable inside different language environments

CORBA does **not** provide any **object creation** (neither Factory): the creation is inside language environments and predefined there, outside CORBA scopes (the same as C does not provide any I/O)

CORBA IDL -> STUB E SKELETON

The **Interface Definition Language** (CORBA IDL) allows to generate **support component** (**stub** and **skeleton**), for communication and data, inside **different languages**

The **stub** enable working on the *message from the client perspective* (marshalling) and acting as client proxy

The **skeleton** collaborate with the ORB *accepting service request and adapting it to the server* (unmarshalling), by managing requests and responses

DEPLOYMENT

Typicsally, there is a **static link** between ***interface - client - servant*** *(not between* **client** and **servant**, *but between* ***client - service*** *and* ***service - servant****)*

The ***objects inside their different language environments*** *are bound to the stub and skeleton before execution*

(stub and skeleton are objects? no)

CORBA ADAPTER

Adapter (**Object Adapter**) **system component** to overcome **dishomogeneity** and **differences** among implementation of different **service environments** of different servants

(the Adapter does not connect with data presentation)

The Adapter is on the **server** side, with typical tasks of:

- object ***registration*** functions
- object ***external reference*** generation
- object and **internal process *activation*** *even on demand*
- *requests* ***demultiplexing*** to uncouple them
- ***send requests*** (upcall) to registered objects

Firsts adapters were Basic (**BOA**), then Portable (**POA**)

(OA are also CORBA objects? no, as OA are pseudo-objects)

INTERFACE REPOSITORY in CORBA

Interface Repository allows to know details about every **IDL data type** and to explore **interfaces**, exported from existent objects and available during execution

The interfaces are translated to different programming languages (**static binding**) where components are defined and compiled (**language mapping**)

IR allows to know and manage available interfaces **dynamically** and to **decide at runtime (dynamic binding)** what is available and convenient

Allows overcoming static approach: for example for a ***gateway*** *that allows access to CORBA interfaces of an environment and cannot be recompiled for every new interface*

IR service description system (it is not a naming system)

(IR is an object? yes)

ORB and IR in CORBA

In CORBA, **ORB is the middle enabler of any (remote) execution and operation request between different entities**

Every request **is always delivered** via the ORB and then server-side mediated BY the adapter

The ORB do not know about any **type information,** that are outside his scope and contained inside stub, skeleton and **language environment**

Interface Repository works as a **dynamic catalogue** of **interfaces** (not necessarily for **static** stub and skeleton),

And it is present for **dynamic explorations** at runtime**,** if it is necessary to retrieve information on dynamic interfaces

The interfaces must be always registered within the IR at their time of use and before consultation

In the **static case**, the IR is generally not needed (its function is plaid by proxies)

IMPLEMENTATION REPOSITORY (IMR)

The **Implementation Repository** is an internal tool of the architecture (not so application visible) to register and store lasting implementations of the servants

The IMR keeps track of every **servant implementation** and allows recovering and making available them in case of restart

- The interfaces are available inside IR, the **implementations are traced by IMR**

IMR allows to know and recover servants that provide specific interfaces (in a stable way) and allows to recover precisely the '**corresponding objects**'

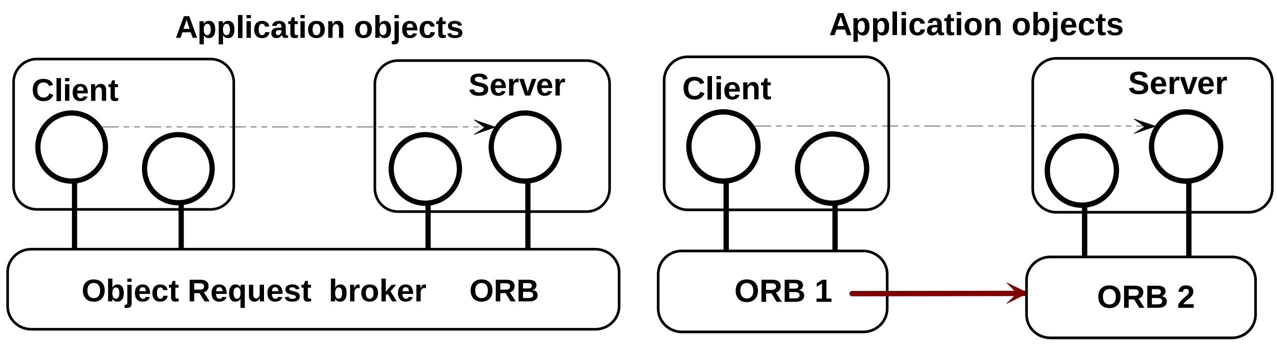
IMR is an **internal repository** for servant management (not a name system)

(IMR is an object? no)

DIFFERENT ORB SYSTEMS

ORB for communication of objects (intra-ORB) and also for communication between objects in different ORBs (inter-ORB)

In one CORBA system or in more CORBA systems managing different brokers



DIFFERENT CORBA SYSTEMS

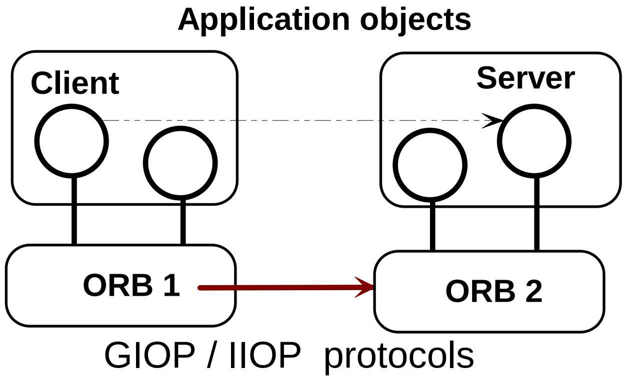
Definition of Inter-ORB standards to establish how to integrate different CORBA systems without problems

- Necessity of standard protocols **ORB-to-ORB interoperability**
- **G**eneral **I**nter-**O**RB **P**rotocol (**GIOP**) that prescribe a standard message format

CORBA specifies a protocol between different ORBs in terms of architecture and data exchange

Binary Communication protocol: data are optimized and non user-readable (no source)

Common Data Representation (**CDR**) standard



INTER-ORB PROTOCOL: GIOP e IIOP

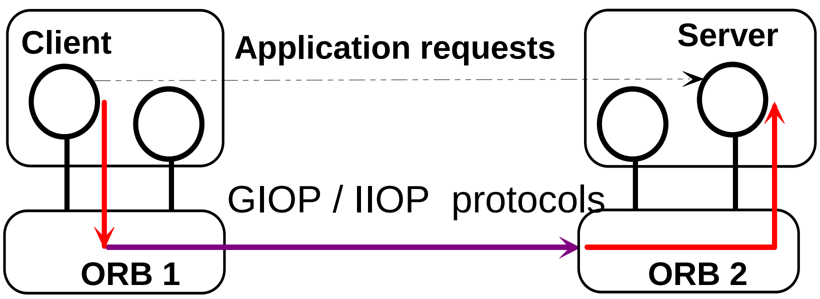
Definition (since version 2) of Inter-ORB Protocols to precisely the interaction between different CORBA systems

ORB interoperability protocol

General **I**nter-**O**RB **P**rotocol (**GIOP**) - **Binary protocol**

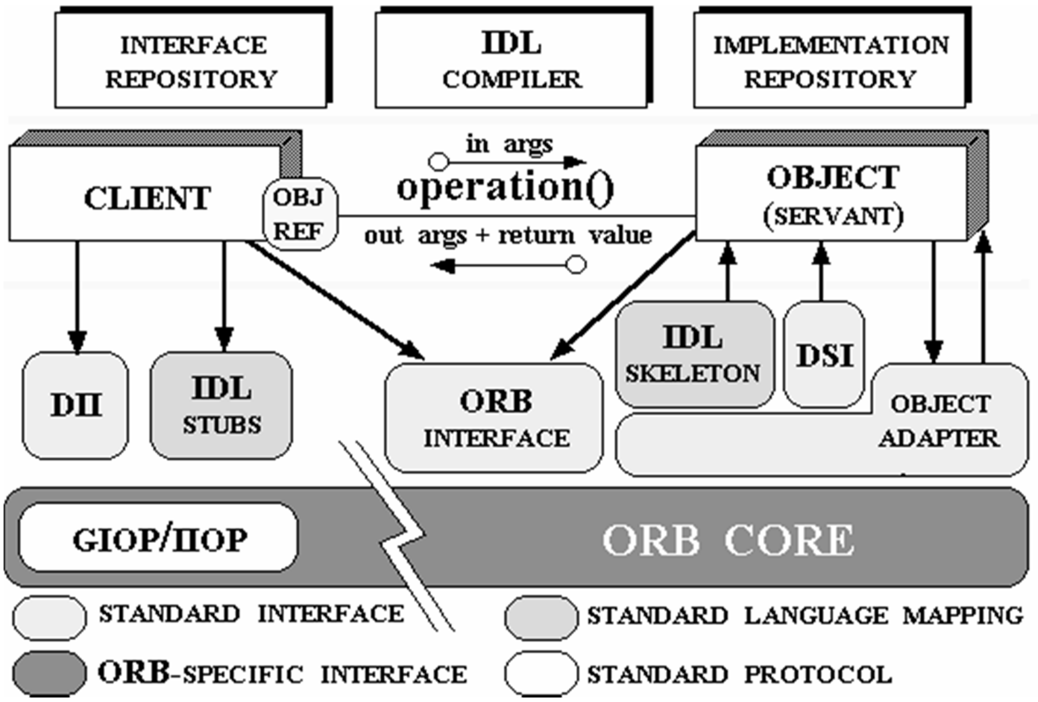
Common specification of data representation, data format, interaction with transport messages (semantic assumptions: reliable, connection, …)

for Internet using TCP/IP - **I**nternet **I**nter-**O**RB **P**rotocol (**IIOP**)



CORBA ARCHITECTURE

Overall picture of a communication between ORBs



CORBA: PSEUDO-OBJECTS

Support components and pseudo-objects

Stub generated from IDL interface for a specific language
Skeleton generated from IDL interface for a specific language

These components realize the **Static Invocation Interface SII**

The SII consists also of other architecture component, such as **IDL interfaces** (to generate stub and skeleton), (interface and implementation) **repositories** to find component specifications and implementation, and **object references**

The dynamic part is implemented in other **pseudo-objects**

DII, **D**ynamic **I**nvocation **I**nterface, or *Request* object
introduced for client dynamic invocation

DSI, **D**ynamic **S**keleton **I**nterface, or *ServerRequest* object
introduced for server dynamic invocation

ORB base functions

ORB acts as a coordinator, as an enabler, and as a manager of services available on the system

> CORBA applications produces **objects** that become part of the system beyond **application lifetime**
>
> The **applications** and the **objects** are developed using **different environments** to represent **stable resources** that can act to request **methods** and **execute operations**

ORB intermediates any interaction and

- **coordinates requests from client objects**, **transparently** from the position and the implementation of remote objects
- **facilitates** and **manages communication through the**use of **references** to existing **servant objects**
- **supports** and **controls** the whole **interaction**

ORB functions

ORB is a fully object interaction enabler, by suggesting a default blocking synchronous interaction

ORB limits its interaction responsibility by delegating individual language environments for final execution

CORBA is **not responsible for object creation and moving**

CORBA employs **external remote references** that are **externally created** by language implementation environments that must define their service objects (**servant**)

CORBA obtains **remote references** via:

- conversion of **string references** and vice versa (objects referred and translated into strings - stringification, and vice versa)
- use of **objects directory,** by using name services (Trading e Naming service)
- **Passing of reference parameters** to **servants**

CORBA IDL

INTERFACE DEFINITION LANGUAGE (**OMG IDL**) has been introduced to grant flexibility over heterogeneous platforms

IDL are **declarative languages** to **specify interfaces** and **involved data** (for API parameters)

Many common IDL are **procedural**

* OSI **ASN.1** / GMDO
* ONC **XDR** (SUN RPC)
* Microsoft **IDL**

CORBA IDL is an **object-oriented** language (***derived from C++***)

Obviously, different IDLs are **not compatible** with each other, even if often are different only for **syntax** and **identification systems** and **entity names**

CORBA IDL

CORBA IDL is a purely **description language for data** and **method interfaces**

- description of **interfaces definition**
- **interfaces** as set of method and attributes
- **multiple inheritance** of interfaces
- **exception** definition
- automatic management of **attributes**
- **mapping** for **different languages** and environments

The compiler can obtain automatically stubs for clients/servants even using different languages

We must consider different **language mapping** **for references to servant objects** (in different languages)

CORBA IDL EXAMPLE

```
module Stock
{exception Invalid_Stock {}; exception Invalid_Index {};
 const  length = 100;

 interface Quoter {
   attribute float quote; readonly attribute float quotation;
  long get_quote(in string stock_name) raises (Invalid_Stock);
 };
  interface SpecialQuoter: Quoter {
   attribute float quotehistory [length];
   readonly int index [length];
   long get_next (in string stock_name) raises (Invalid_Index);
   long get_first(in string stock_name) raises (Invalid_Index);
 };
 interface CancelQuoter: SpecialQuoter {
   long cancelhistory (out float cancelledquote [length])
 };
}
```


CORBA IDL SUPPORT

For any attribute, an automatic access function is provided suited for permitted operations
(_get for readings and _set for writings)

```
attribute float quote;
float    _get_quote ();
void     _set_quote (in float q);
readonly attribute ind index;
float    _get_index ();
```

For any exception, the state (completion_status) provides information on behavior semantics

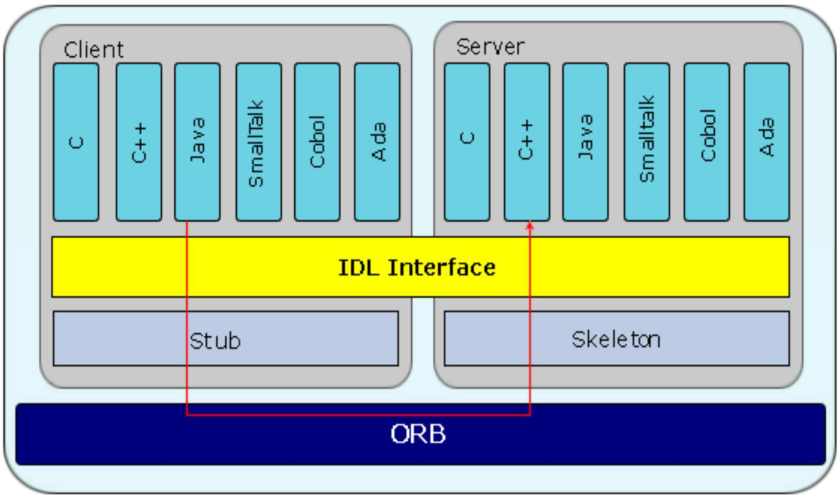
```
COMPLETED_YES,
COMPLETED_NO,
COMPLETED_MAYBE
```

CORBA IDL

Language to define CORBA interfaces, independently of a **specific programming language**

Naturally it is necessary **pass** from the abstract **CORBA level** to concrete **specific languages** (**language mapping**)

CORBA specifies the need of **mapping environments**
Servant creation is a responsibility of each language mapping



CORBA IDL ENVIRONMENT

CORBA is an **environment** where **we use remote references and do not move objects (static objects)** because of **the heterogeneity of single deployment environment**

Remote references allow to request operations to other components with known CORBA interface

Every object has **an interface** (**coarse granularity**)

Interfaces define: **attributes**, **methods**, **exceptions**
(*attributes accessed through* ***get*** *and* ***set*** *operations*)
(*operations with* ***in*** *or/and* ***out*** *arguments*)

The interfaces use **multiple inheritance**

The **interfaces** can be grouped also within **modules**
(for logical aggregations)

OTHER CORBA IDL EXAMPLE

```
module BankAccount {
struct transaction { string data; float amount;};
exception RedException {string message;};
typedef sequence <actions> list_ops;
interface Account {
  float balance(in string cc);
  list_ops bankStatement (in string cc);
  void withdrawal (in string cc, in float amount,
     out float balance) raises RossoException;
  Account accountTwin(); // returns an object };
};
```

Parameters passed by value (**CORBA** objects **by references**)

Problem of parameter handling in **out** and **in out**

DATA in CORBA IDL

Types in CORBA

Object Reference (references to **objects or interfaces**)
vs. even with inheritance between CORBA objects

Value (values copy) and **Exceptions**

Basic values short, long, ushort, ulong, float, double, char, string, boolean, octet, enum, Any

Constructed values Struct, Sequence, Union, Array

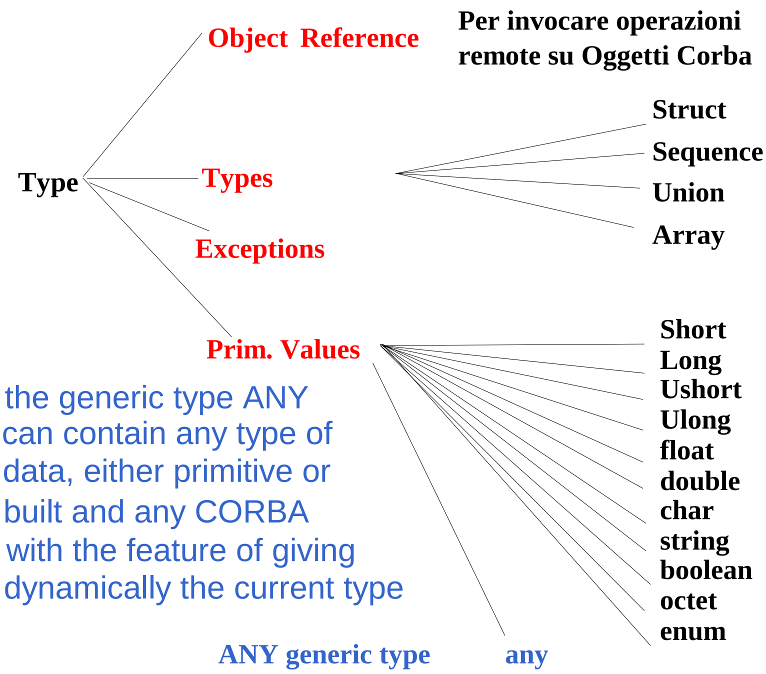
Any as general type that contains any type, primitive or from CORBA interface (analyzable during execution)

Object by value *(CORBA 3)*

*Objects that **cannot** be accessed remotely but only passed **by copy** from an environment to another one overcoming heterogeneity of different environments (no remote reference to them)*

TYPES in CORBA IDL

Types in CORBA IDL



Types of CORBA IDL RI2 **are than translated into types of different programming languages obtained for different language mapping**

Type Object (IDL) represents any type of CORBA object without any information of the specific type

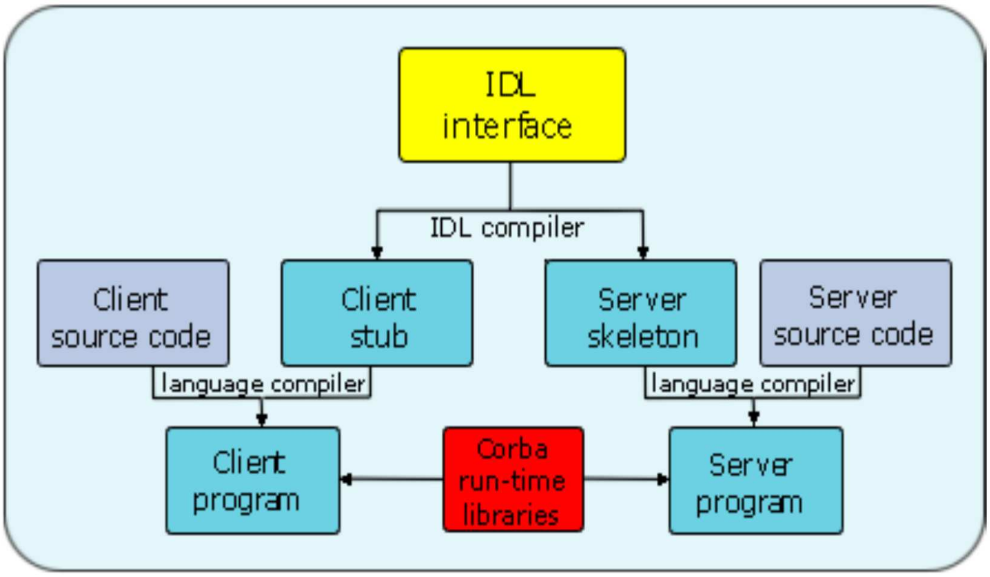
Diapositiva 34

RI2 immagine con testi in italiano
Raffaele Ianniello; 13/04/2016

Form CORBA IDL to Languages

Tools allows to build from CORBA IDL different components, essential to the project and to execution in **different language mapping**

stub and **skeleton**
+
file helper
and **of other help** (**holder**)
+ other operations



CORBA Language mapping

CORBA defines

interfaces (with inheritance), **exceptions**, **methods** with **objects as parameters** of different types and with different **modes** (**`in, out, in out`**)

Different languages must add **concepts,** to harmonize their **structures** to obtain **interface conformance** and guarantee **run-time operations (OO languages must integrate inheritance)**

Strategy for consistency of concrete language types and possibility of integrating with the CORBA model

various transformation functions provided automatically management of types, to put together structures in simple way,

Apart from many other support functions (naming, trading, and suggested development methodologies) usable by user

CORBA vs LINGUAGES: HOLDER

Use of **holders** in JAVA as language where are output parameters

for example

public final Class **BalanceHolder …**

{public float value;

public **BalanceHolder**() {}

float _**read**() {return value;}

void _**write**(float value) {this.value = value;}

};

for **`out`** and **`in out`** parameters (also other helps: helper)

In general, every **language** must create anything that is necessary to foster development inside its environment

CORBA HELPER

Helper use for Language mapping: in Java functions to

- **harmonize and treat language types and CORBA types**

*In Java the **CORBA Object** type is mapped in org.omg.CORBA.Object*

functions of **narrow**-ing that transform from the CORBA Object type to the one defined inside the interface

functions used for managing transformations from abstract CORBA type for the specific concrete type of interest

- **implement various utility functions**

functions for **reading** and **writing** a type on an object stream (associated to CORBA interface), to **treat type dynamically** during execution, …

Every **language** must guarantee interoperability with CORBA

CORBA ENVIRONMENTS AVAILABILITY

Widely used and still rising

`Object Broker`	`DEC`
`ORB`	`HP`
`DSOM`	`IBM`
`Orbix`	`IONA`
`Visibroker`	`Borland`
`(DOM Facility)DOE`	`Sun Studio Sun`
`PowerBroker`	`ExperSoft`
`JacORB, ...`	**`Open source tools`**

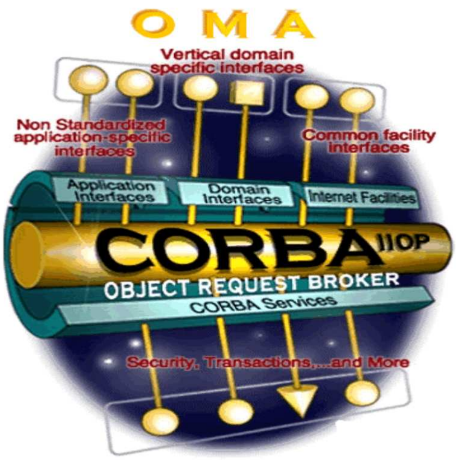
Even if the learning curve is high and there is overhead in performances

Middleware

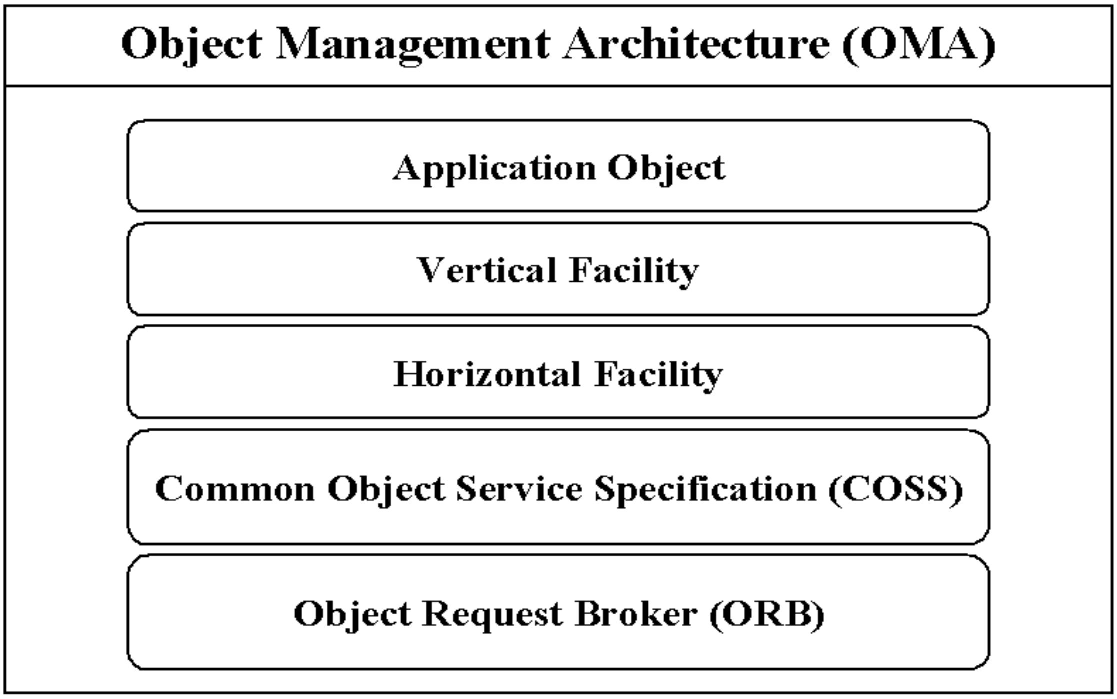
Most used middleware must provide answers even to in the small needs and further details

CORBA users expect to :

- design **quickly new components**
- **integrate** old **legacy** components
- use **existing tools** and **available assistance** components
- **integrate applications** with new **available facility**
- have a middleware **capable of host services without interruption (QoS) and without lifetime limit**



CORBA ARCHITECTURE



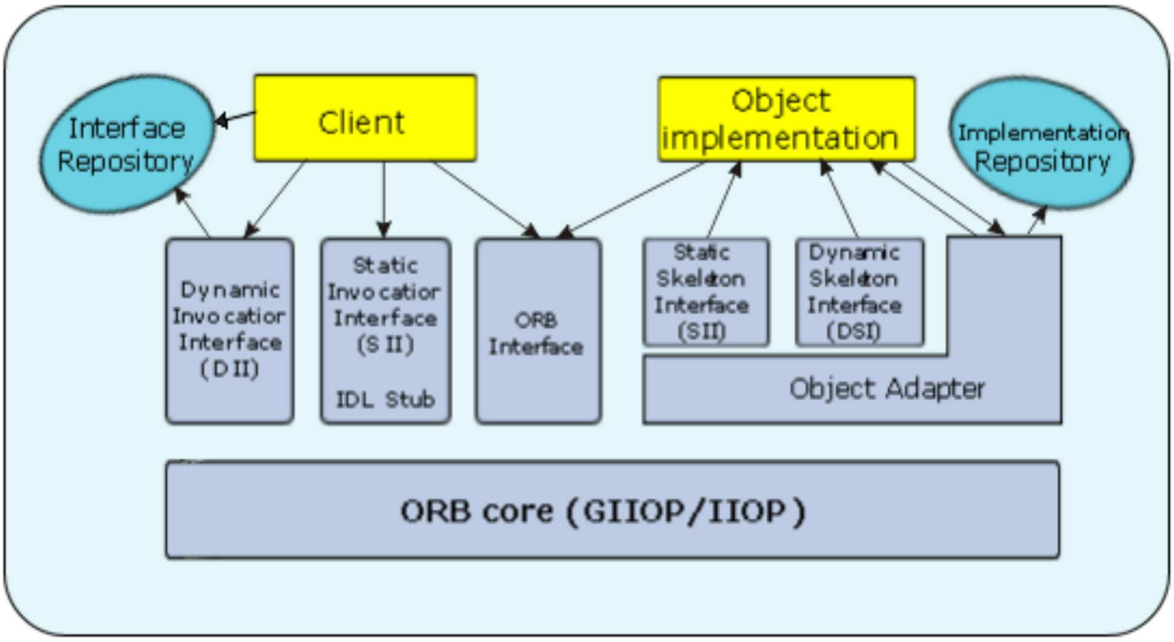
CORBA COMPONENTS

CORBA essential components

*	**Object Request Broker**	(**ORB**)
*	**Interface Definition Language**	(**IDL**)
*	**Basic (e Portable) Object Adapter**	(**POA**)
*	**Static Invocation Interface**	(**SII**)
*	**Dynamic Invocation Interface**	(**DII**)
*	**Interface e Impl. Repository**	(**IR e IMR**)
*	**Protocolli per Integrazione**	(**GIOP**)

CORBA ARCHITECTURE

Global view of the base architecture of service support



CORBA APPLICATION DESIGN

Common interfaces must be specified by clients and servers

After the generation of **stub** and **skeleton**

Server must implement servant **classes**

The servant **must register itself**

Client must implement **classes**

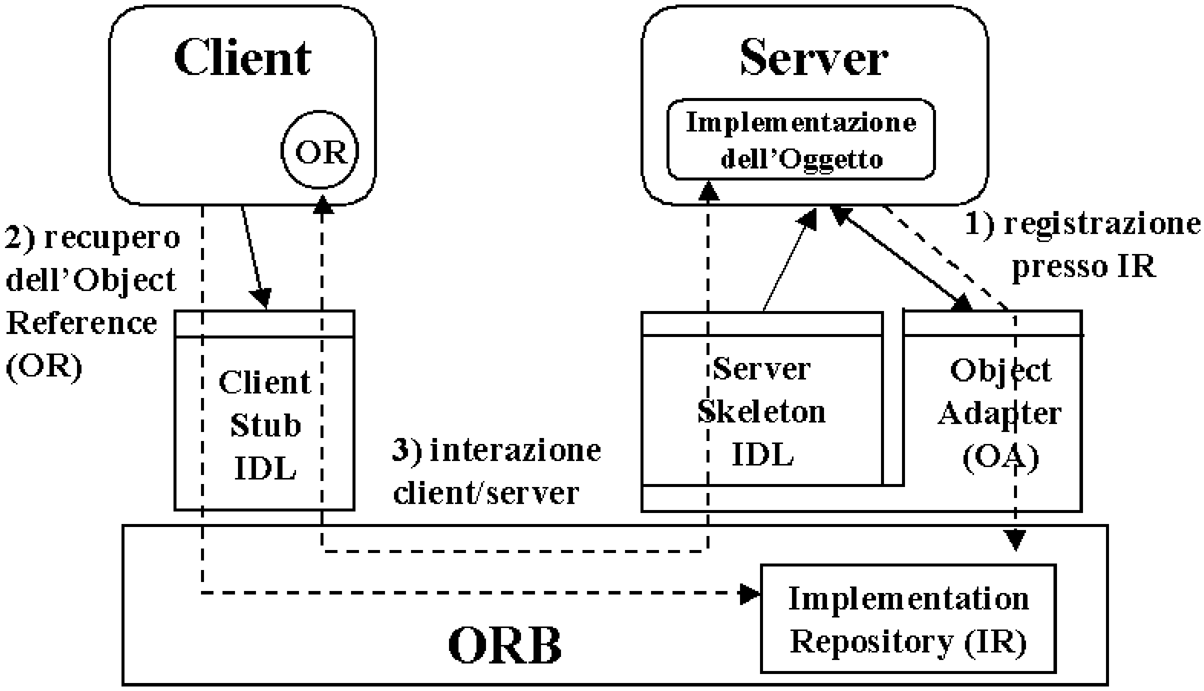
The **execution** starts

Client needs **remote references** to find the **server**, the **components**, …, and, in general, the entire **support entities**

Deployment: how many ORB? Where? How can be reached?
Or on **every node** (local API), or centralized **ORB**, or **more** servers

With which QoS? And which fault-tolerance?

CORBA COMPONENTS



RI3 Immagine con testi in italiano
Raffaele Ianniello; 15/04/2016

ORB INTERFACE in CORBA

ORB can be intended as a set of class that permit a **good remote reference support**

Various conversion functions

functions for transform **ObjectReference (or Object Interface)** into strings (to maintain them easily) and vice versa

```
Interface ORB {
    string object_to_string (in Object obj);
    Object string_to_object (in string str); }
```

With stringification, we can pass from a form to another even in different environments

Also functions for initializing different OA, to find necessary services, base functions, etc.

ORB INTERFACE in CORBA

ORB Various functions

ORB **Initialize** for booting

```
ORB ORB_init(inout arg_list arguments,
    in ORBid ORB_identifier)
```

to initially connect to the ORB all users (clients, servants, …)

Also a set of functions to find **default context** and obtain references to **base services** (IR, naming service, …)

```
typedef string ObjectID;
typedef sequence <ObjectID> ObjectIDList;
ObjectIDList list_initial_services ();
Object resolve_initial_references (in string ObjectID);
```

KNOWN INITIAL OBJECTS in CORBA

The function to obtain base objects (ObjectReference) in general allows access, for example:

`Object resolve_initial_references (in string ObjectID);`

CORBA support objects founded through Initial Services

“**RootPoa**”, “**POACurrent**”, “**InterfaceRepository**”,

CORBA support services

“**NameService**”, “TradeService”, “NotificationService”, …

current CORBA policies

“ORBPolicyManager”, “TransactionCurrent”, “PolicyCurrent” …

An object reference into string could be:

IOR:000000000000001949444c3a696f722f53696d706c654f626a656374
3a312e300000000000000001000000000000003000010000000000a737
465656c7261696e00079e00000018afabcafe000000023bd4cf8d000000080000000000000000

Object Reference in CORBA

One Object Reference allows to refer to **an instance of a remote service** (**a stub**): OR are **opaque** and **not** internally visible by users that can only pass around; only the ORB can manipulate them (they potentially integrated with persistence management)

Object References refer to CORBA **Object** instances

The operations provided by the ***Object Interface*** are many to permit to work viably in a transparent way

```
get_implementation, get_interface,
is_nil, non_existent, is_a, is_equivalent,
hash, duplicate, release,
create_request, get_domain_manager,
get_policy, set_policy_overrides, …
narrow, this, …
```

Object Reference in CORBA

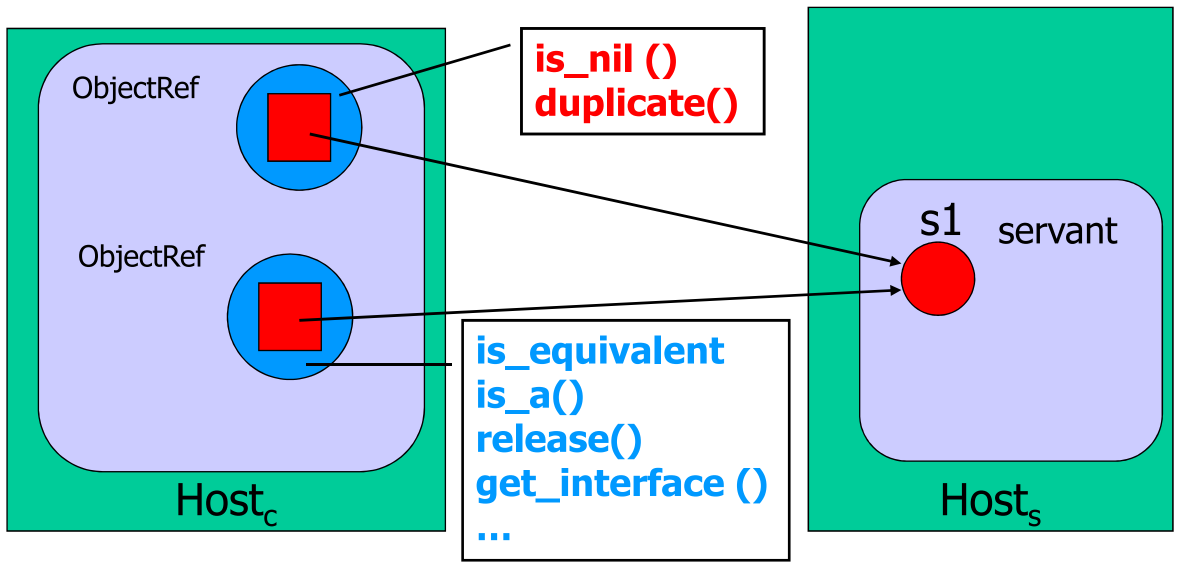
Object Reference of CORBA inherit from

CORBA::Object interface

```
interface Object {
// operations for objects management
Object duplicate (); void release ();
// operations for know the object
Object get_implementation ();  Object get_interface ();
// operations of existence and reference
boolean is_nil ();
boolean non_existent ();
boolean is_equivalent (in Object other_object); // same obj?
boolean is_a (in string repository_id); //implements?
Object create_request (in Object); // create request object
// ...
}
```


Object Reference in CORBA

Object Reference are opaque but **right practical operations and management functions** must be available



CORBA VERSIONS

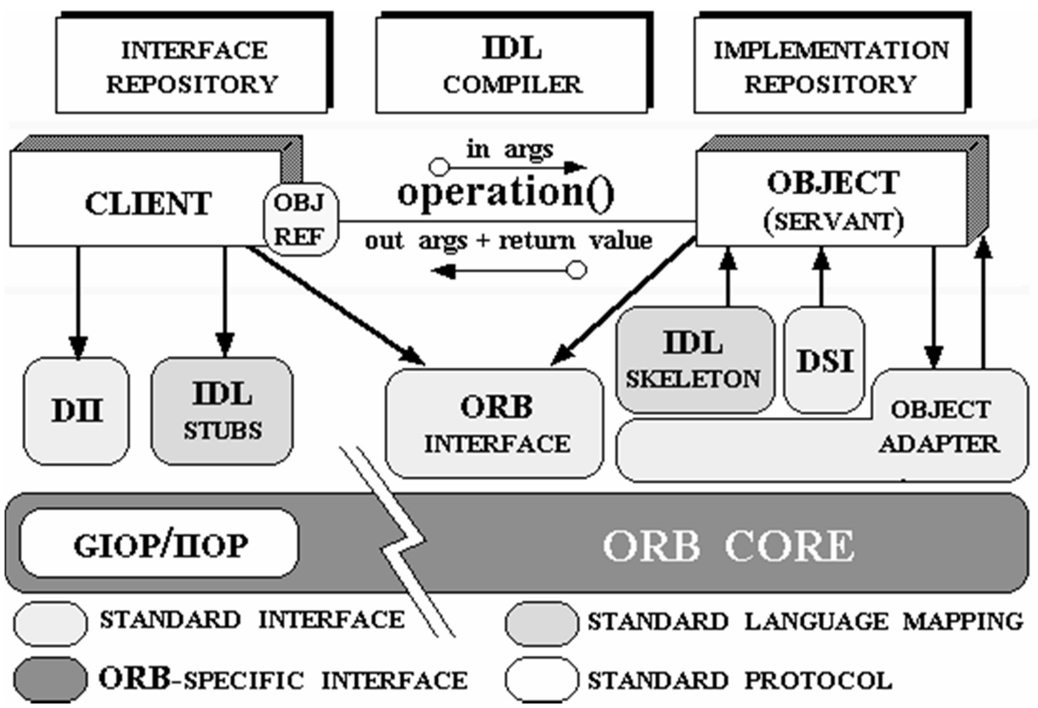
CORBA maintains **essential components** even in its evolution but enriches itself with ***tools*** and ***components*** to *deal with* ***new problems*** and to *provide a* ***better support***
Essential components always the same goals

- Interaction between different languages environments
- Helps to use different languages environments
- Tools to obtain **QoS** in different languages environments
- New general utilities and for specifics domains
- New realizations and integration with different existing development environments

CORBA 3 (2000, 2005 -…)

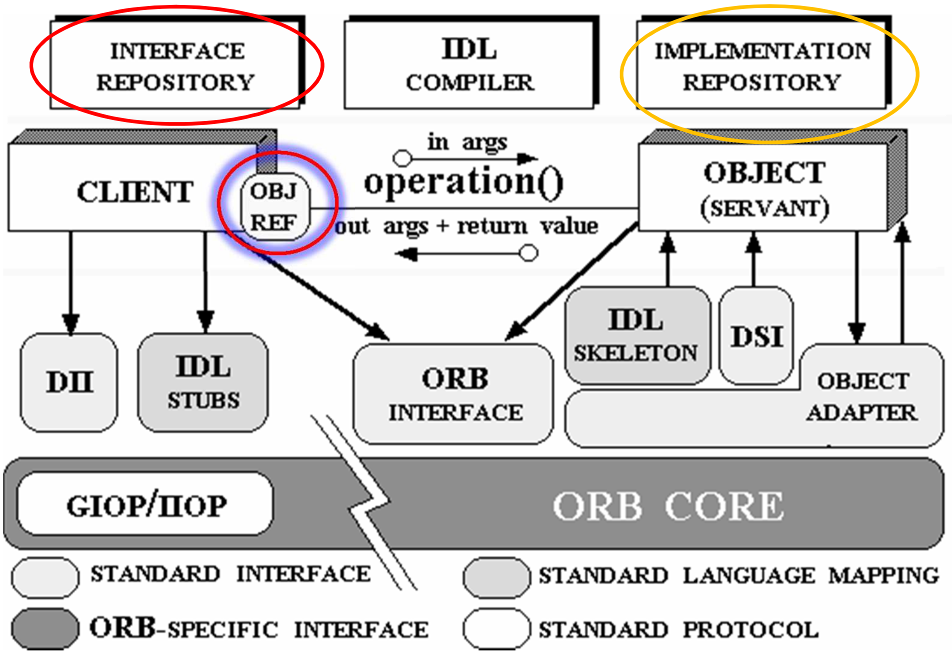
CORBA ARCHITECTURE: details

Global view of architecture implementation



CORBA REPOSITORY

IR must register interfaces of every available service, **IMR** servants code; **Object references** allow to obtain services



STATIC BINDING in CORBA

Static Invocation Interface (SII)

*The compiler and the tools enable the call before execution, by creating **stub** and **skeleton***

Every invocation is safe and verified in advance

No dynamic control on the interface is done given that proxies are generated statically

The **client** *binds itself to the stub* and send the request using the reference after connection to the ORB (**synchronous invocation**)

The **servant** is *bound to the skeleton* and is activated by the object adaptor (POA) for the requests

There is no connection between **client** and **servant**: subsequent requests can go to different servants, but with the same interface

In case the (none) **servant is not active, POA activate it and sends the request**

CORBA SEMANTICS

Normal mode is **blocking synchronous**

In case of malfunction or problems, the client receive an exception expected from the interface

At-most-once semantics

In CORBA the **synchronous invocation** is based on static proxies as mediator

Obviously that can be limiting

The **synchronous** static invocation has a 'very limited' cost (if operation with coarse granularity are foreseen) but can also produce delays

Are other modality necessary?

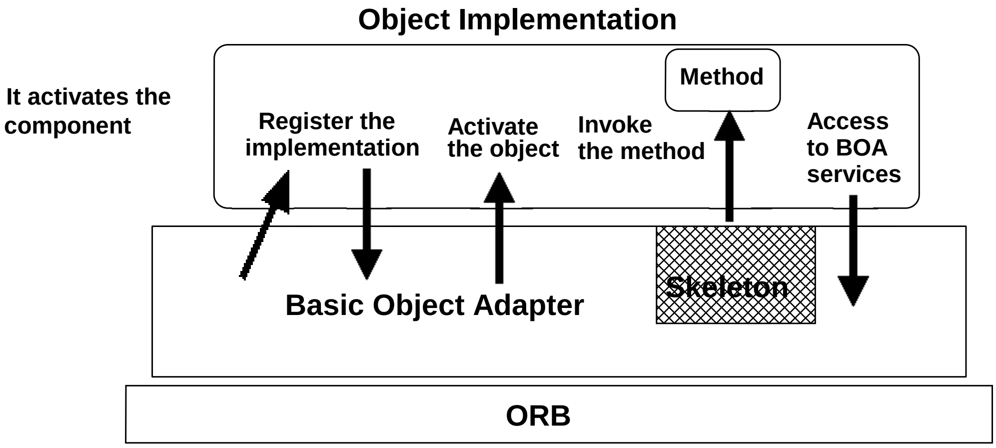
Oneway in IDL: *no response (best effort)* *deprecated*

CORBA COMPONENTS: ADAPTER

The **Adapters** are the components responsible for CORBA flexible scheduling

The **various adapters** must reach the implementation of different servants and control and manage the real execution

We call **servant** the **passive entities** that embody the real object server functions



ADAPTER functions

The **Adapters** supervise **operation execution** inside servers through the concept of **servant**

SERVANT USE

A **servant** is the **object part that makes available the code to execute on the request of a client** (entity that is highly dependent from the programming language and from the servant specific programming environment)

The **real service implementation** within a language

The POA has the assignment to compose the image of the CORBA object server

A POA (on a node) could control:

- **a unique servant**
- **also many different servants to provide to different requests**

A **POA** decide **its servants** and its **management policy**

CORBA COMPONENTS: ADAPTER

The Adapters control the execution **of abstract server** via **real servants that work on service code**

MANY ACTIVATION WAYS INSIDE SERVER

activation for every request (**thread_per_request**)

a process is created inside the object for any service

initial pool activation (**pool of threads**)

every object receive its process from a process pool initially created, without paying any creation cost at run-time

per-session activation (**thread_per_session**)

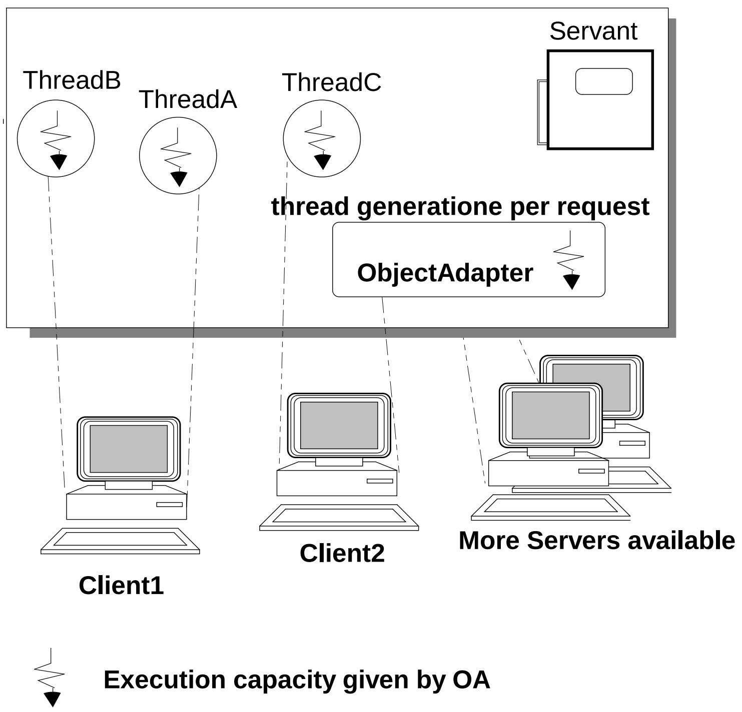
every client has a process dedicated to interaction

Also other modes: a *thread per servant* (thread_per_servant)

a unique activation for more server objects

(shared server) simultaneously

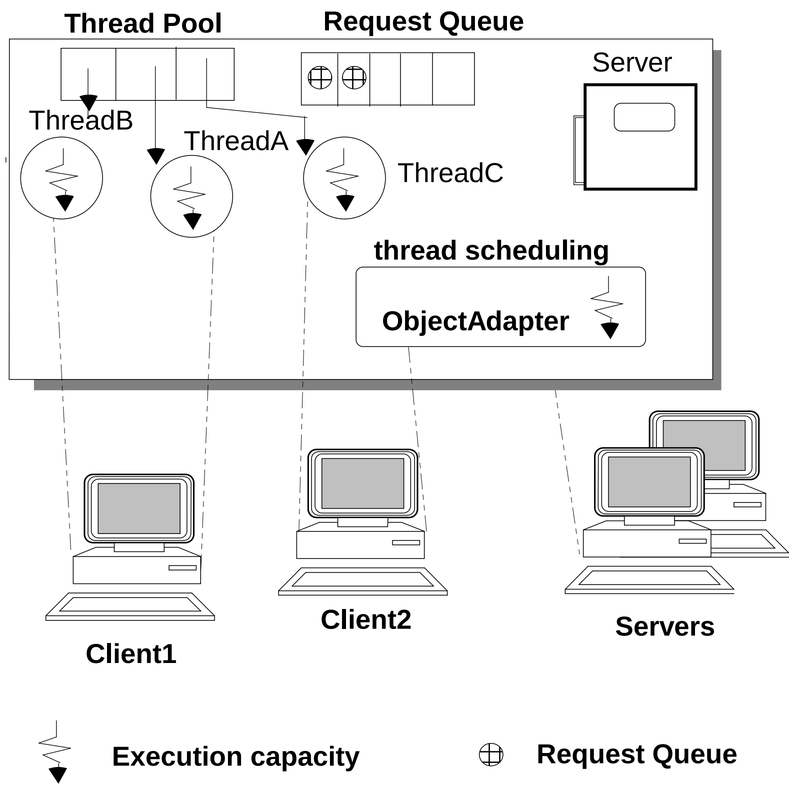
THREAD-PER-REQUEST Activation



Every request obtains a thread activated by-need

High activation cost that intrudes on every operation

THREAD-POOL Activation

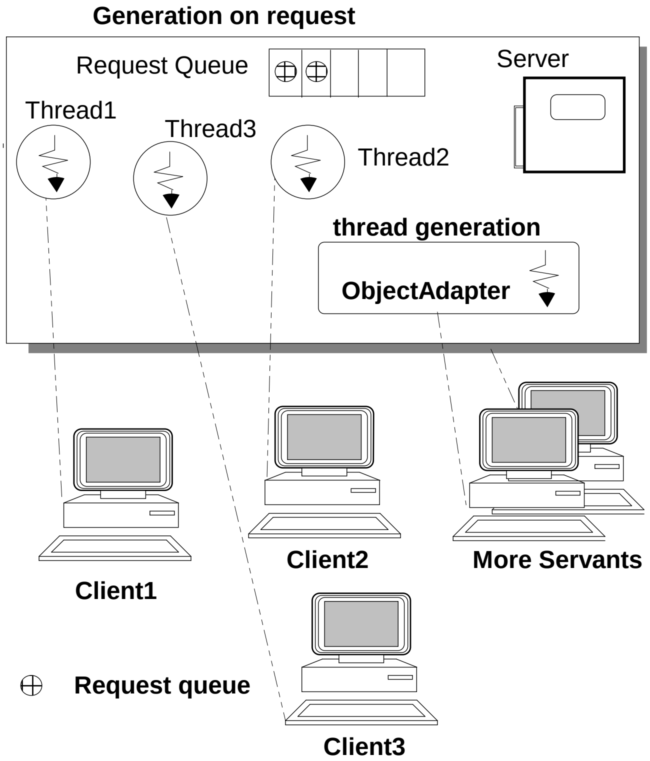


Every request receives a thread from a pool of pre-created processes

In case they are not available, it waits until one is freed

Less cost, but long wait in case of high traffic

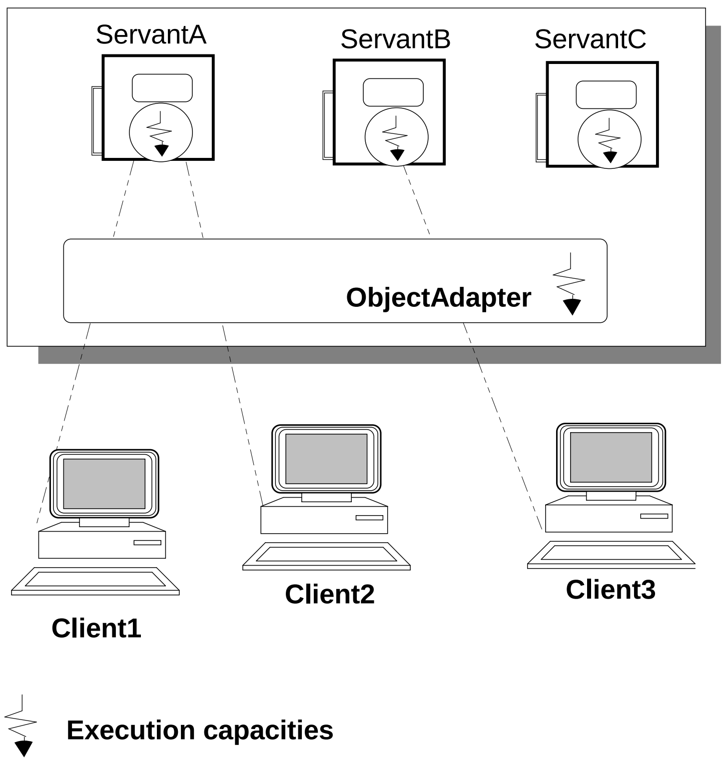
THREAD-PER-SESSION Activation



Every client receives an active thread at the beginning of working session

Service parallelism is limited

THREAD-PER-SERVANT Activation



Every object is embodied by a servant that expect a thread from first activation and answer only through that

Parallelism is limited based on the servant number

MIDDLEWARE for CONTINUITY

CORBA is a **middleware** used to support and enable infinite life cycle and time to organization resources and try to ultimately support that perspective

Middleware for service continuity

The infrastructure stresses the idea of an **interface based contract** and can optimize implementation resources based on policy defined on specific user defined indicators

The middleware can **balance servant workload for a service to obtain a better throughput**

The middleware can **route requests towards other resources, in case of malfunction (no downtime), or to enable other strategies (**e.g., of **localization, load balancing),**

The middleware can also **direct load towards servants in other CORBA systems for differentiated service**

CORBA COMPONENTS

Essential CORBA components

* **Object Request Broker (ORB)**
* **Interface Definition Language (IDL)**
* **Basic (e Portable) Object Adapter (POA)**
* **Static Invocation Interface (SII)**
* **Dynamic Invocation Interface (DII)**
* **Interface e Impl. Repository (IR e IMR)**
* **Integration Protocols (GIOP)**

DYNAMIC BINDING in CORBA

Dynamic Invocation Interface (DII) and

Dynamic Skeleton Interface (DSI)

necessary to operate without **static links** to the **interface**, namely to connect *with* ***interfaces that do not exists at compile time (but only defined lately)***

DYNAMIC behavior

In general, the **dynamic behavior** allow an application to adapt to **situations not forecast** during development, or better to interfaces unknown at development time (so to extend **application lifetime**)

In this case, the client and server can bind to not forecast interfaces at the application start

DYNAMIC BINDING in CORBA

Dynamic Invocation Interface (DII) and

Dynamic Skeleton Interface (DSI)

DYNAMIC behavior

Client and server, that did not provide any proxy, must use at run-time **pseudo-objects** for **viability**

The duty of run-time checks for type safety are left to the user code and no support is provided

The interfaces used in the dynamic case must be registered and available for the dynamic usage

Interface repository allows to discover any detail of knowledge of the interface (that must be present before its usage)

DYNAMIC CLIENT in CORBA

DYNAMIC CLIENT behavior - DII - the client

- receives a dynamic **`ObjectReference`**, for which no proxy has been statically produced
- creates an object **`Request`** after discovering its interface and tailored to that interface
- uses the object **`Request`** as an interaction mediator with the servants to ask methods from them

The **`Request`** can be used for **many dynamic requests** with the **same interface it has been created for**

In the client case, we can assume invocation less synchronized and constrained between client and server (different forms of asynchronicity)

DYNAMIC BINDING in CORBA (DII)

ORB allows to create, and manage a **dynamic request** and **invocations** via a **pseudo-object Request**

```
pseudo typedef long ORBstatus;
ORBstatus create_request {// Pseudo IDL
  in Object          obj // operation object
  in Context         ctx // operation context
  in Identifier      operation // operation name on object
  in NVList          arg_list // operation arguments
  inout NamedValue   res // operation result
  out Request        req // created request for the operation
  in Flags           req_flags // operation flags
}
```

It is always possible to prepare a **Request** oriented towards all the operations for an Interface to request their invocation

When the request is available, the client can use it for the methods the client wants to invoke

DII: REQUEST for DYNAMIC BINDING

The API to compose requests by using a **pseudo object Request to incarnate the DII**

```
pseudo interface Request {// Pseudo IDL
  Status add_arg (in Identifier name, in TypeCode arg_type,
                  in void *value, in long len, in Flag arg_flag);
  Status invoke (in Flags invoke_flags // invocation flag);
  Status get_result (in Flags flags // result extraction flag);
  Status send_oneway (in Flags flags  // invocation flag); …
  Status send_deferred (in Flags flags  // invocation flag);
  boolean poll_response (in Flags flags  // invocation flag);
  Status get_response (in Flags response_flags //response flag);
}
```

The **invoke** allows to ask for the execution of methods; results can be waited or more asynchronously managed (see send_deferred)

The request is prepared and the **dynamic call** involve a higher cost compared static synchronous operations

PSEUDO OBJECT in CORBA

CORBA architecture provides some support entities called **pseudo-object: Request** is one of them

The pseudo-objects are entities necessary for user to obtain viability without becoming CORBA objects; they:

- do not have a CORBA reference *(not objects)*
- *are confined inside specific ORB*
- *helper, holder, etc. in different language mapping are not produced*

Pseudo-objects are enablers that have the same CORBA description of application entities (while being system defined) and an application can use for its purposes

Pseudo objects can be mapped differently inside different languages and available only in some language environments

DYNAMIC BINDING in CORBA (DII)

A user **must operate** through a **Request object by submitting it to the ORB** for operations execution. The steps are:

- *request* creation
- setting/check of **in *parameters*** (name, type, value)
- set of *answer* (type)
- set of possible *exceptions*
- set of possible *contexts*
- **real invocation** (also **oneway** o **deferred**)
- *verification of possible exceptions* (after completion)
- *extraction* of all request result *information*:

*parameters of **out**, **in out**, **return value***

The remote reference allows to find and explore **the interface through the IR repository**

INVOCATION SEMANTICS in CORBA

The standard CORBA mode is **blocking synchronous**

In case of malfunction or problems, the client receives an exception from the interface

At-most-once semantic

The static **synchronous** invocation introduce less steps that corresponding **dynamic**

New introduced modes only for **dynamic invocation**:

Oneway Invocation: no response (best effort semantic)

Deferred Synchronous: the answer is expected but it is to be found later (at-most-once)

use of **get** and **poll** for the answer

It is possible to mix static and dynamic modalities?

DYNAMIC INVOCATION SEMANTIC

Different modalities of action on pseudo-object Request

Blocking Synchronous

`invoke()  … get_result()`

Oneway Invocation

`send_oneway()`

Deferred Synchronous

`send_deferred()…`

```
/* many operations */     poll_response() …
/* get result      */     get_response ();
```

In case of dynamic invocation, all guarantees of the static control (stub) are not present

Clients are responsible at invocation for all necessary checks: correct parameter types, exception, etc.

DYNAMIC SERVER in CORBA

DYNAMIC SERVER behavior - DSI – The server

- decides to implement a **new interface** where there is not a statically generated skeleton
- uses a **pseudo-object dynamic `ServerRequest`** as a mediator with two fundamental tasks
  - **To register** its **service** interface to the POA and the ORB (in a preliminary way to any invocation)
  - To work in mediating **single request service invocations** (that are driven by a general server function named **`invoke`** registered to the POA). The **`invoke`** must check dynamically parameters and correctness

The **`ServerRequest`** can be used as an enabler for every **interface method**

DYNAMIC BINDING in CORBA (DSI)

A server that intends to provide a **dynamic implementation of operations** must define a **Dynamic behavior** via **Dynamic Skeleton Interface (DSI)**

The servant uses, at run-time, the POA to register itself as a possible and valid implementation of the interface of interest

- A **pseudo-object ServerRequest registers** itself as an implementation to the POA and **allows the POA to consider it** as an allowed and manageable servant
- The dynamic operation requests a **higher correctness checks** (if possible) of parameters that have not been statically checked from language support

Every invocation, static or dynamic, can be sent to the new servant *registered with DSI, that uses the* **pseudo-object** to intermediate parameters and result

DYNAMIC BINDING in CORBA (DSI)

To provide its implementation, the server must use a **ServerRequest** for the **dynamic offer of operations**

```
pseudo interface ServerRequest {// Pseudo IDL
 readonly attribute Identifier operation_name;
 readonly attribute OperationDef operation_definition;
 void parameters(inout NVList params);
 Context ctx();
 void set_result(in Any val);
 void set_exception(in Any val);
};
```

The pseudo-object **`ServerRequest`** must be registered to the **POA** and the ORB does know that a new implementation of a specific interface exists (from the object itself)

ORB and POA play a fundamental role in DSI

DYNAMIC BINDING in CORBA (DSI)

The server object must implement an **`invoke`** to be called by the POA (to execute methods) as a **`callback`**

the ORB requests to the POA to use that **`invoke`** to obtain the generic execution from servant; ORB and POA pass the request to the object, that has responsibility to execute the implmented method (using the content of the **`ServerRequest`**)

Obviously, in DSI, the usual checks of the static case are not done by the skeleton

the **`invoke`** method must **acquire the name method,** the **parameters, check them,** execute the **logic**, and produce **results** (to check the type)

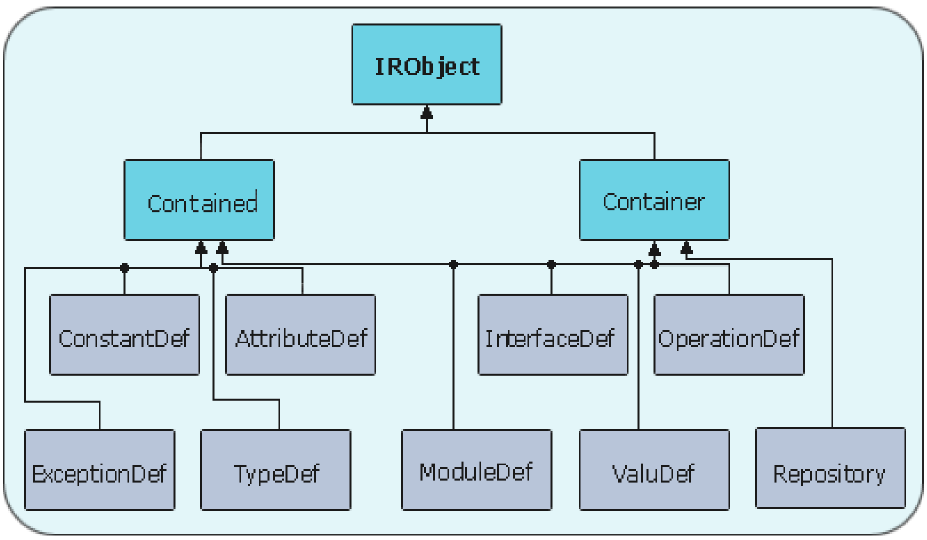
In case of problems, necessary exceptions must be passed

The client receive results without identifying the static / dynamic mode

INTERFACE REPOSITORY

The Interface Repository handles **registrations of** every **interface** and manage **storing** and **discovery (no track of the specific objects that implements them)**

The repository behave as **content container**



INTERFACE REPOSITORY in CORBA

Interface Repository is not a name system, but allows to explore all available **interfaces**

It allow remote access

direct or through **proprietary utilities**

Every entity is also labeled with a *RepositoryID*

Some different standard formats are recognized

IDL	IDL:/Go/Services/Interface:1.0
RMI hashed	RMI: name … /hashcode
DCE format	DCE: UUID
Local format	LOCAL: free

Access operations are standardized

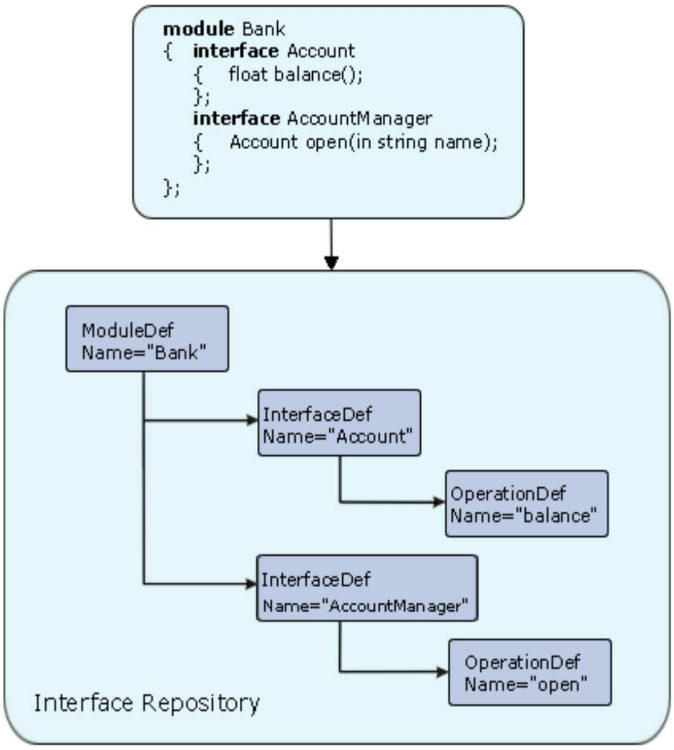
```
Contained lookup_id (in RepositoryID searchid);
InterfaceDef get_interface();
```


INTERFACE REPOSITORY in CORBA

From every defined and compiled interface

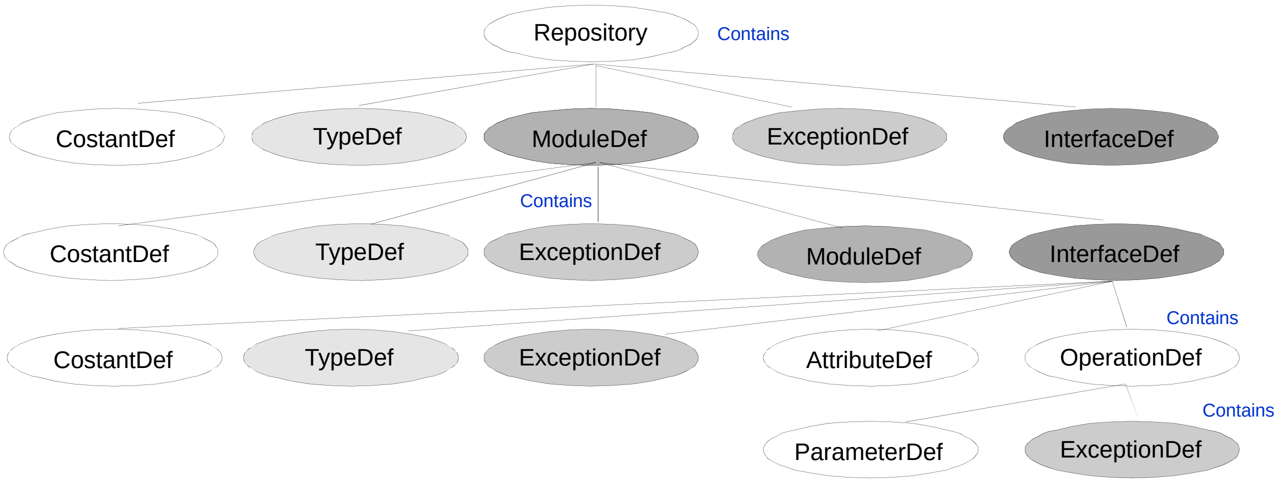
contents for the IR are generated

based on the types that can be defined and acknowledged



INTERFACE REPOSITORY in CORBA

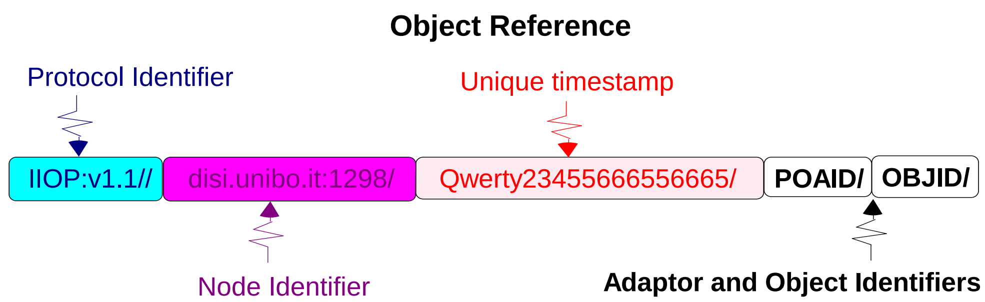
More complete structure of IR types



REFERENCES TO OBJECTS in CORBA

References in CORBA are **opaque** and **allow to reach a POA and to find any servant** without granting a **specific servant implementation**

Typically, they can contain the whle **information** to access to the servants: **address, POA creation name, object ID (*various data*)**



There are problems of different information, also partly visible to users

REFERENCES TO OBJECTS in CORBA

Available **identifiers** at a **CORBA client** are **valid only for the environment in use and opaque to user**

They are completely different from the way the ORB keeps control of the objects themselves and pass them from an environment to another: there exist ***names valid only in some localities,*** while others with the *possibility to* ***identify specific objects*** (*servant*)

A user name (*as in most language environments*) before passing to the receiver, **it is converted** for the execution environment

In a receiver environment, the reference could be also a different object with the same interface

In case of state information on the server, problems ☻

If we want precise and focused information?

REFERENCES TO OBJECTS in CORBA

CORBA acknowledge the need of **identifiers that can pass among different environments keeping servant identity**

CORBA 1.2 **does not provide unique names**

Object Reference (OR) are (not unique) **names associated to a specific service** and **not to a specific servant**

ObjectRefs passing from client to server site are converted by a name system in a proxy for the receiving environment
ObjectRefs must be passed from an environment to another and do not refer necessarily to the same object

Normally,

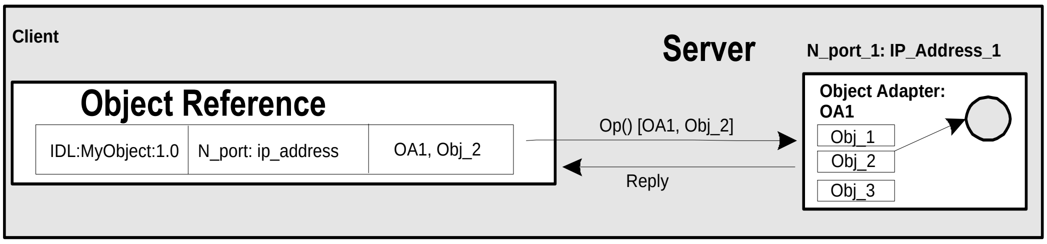
the identifiers inside one ORB are always connected to the specific object (servant)

So, ORs were not portable enough

IOR and UNIQUE NAMES in CORBA

Interoperable Object Reference (IOR) are **unique names associated to a servant** that can be transferred between different ORBs (*passing also through strings*)

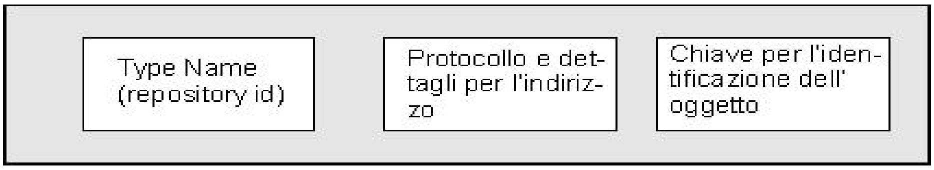
In general, before passing from an ORB to another OR ⇨ **IOR**



CORBA 2 **support for unique names**

Identifiers unique and tied to a specific target

IOR



Interoperable Object Reference or IOR

Standard over the representations of different ORBs for uniquely identify objects

Interoperable Object Reference or **IOR as standard**

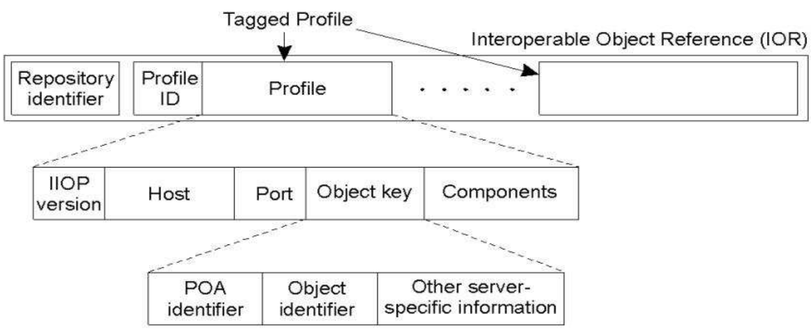
IOR as ProfileID (identifier) and **tagged Profile** (to identify completely)

More than one **profile** for different access rules

Tagged Profile

complete information for object discovery

This information is used to decide what to pass to the client in an operation request (then a local proxy for the object itself is provided)



IOR in CORBA…

We can have **two possible forms of IOR** with different support to **reach the servant via POA** and support of the **Implementation Repository (IMR),** that intervenes to provide by need also the re-generation of registered servants

Indirect link (indirect binding) if **IOR** refers the **repository IMR** and only **indirectly** to the final object

The indirect binding is the one durable and persistent

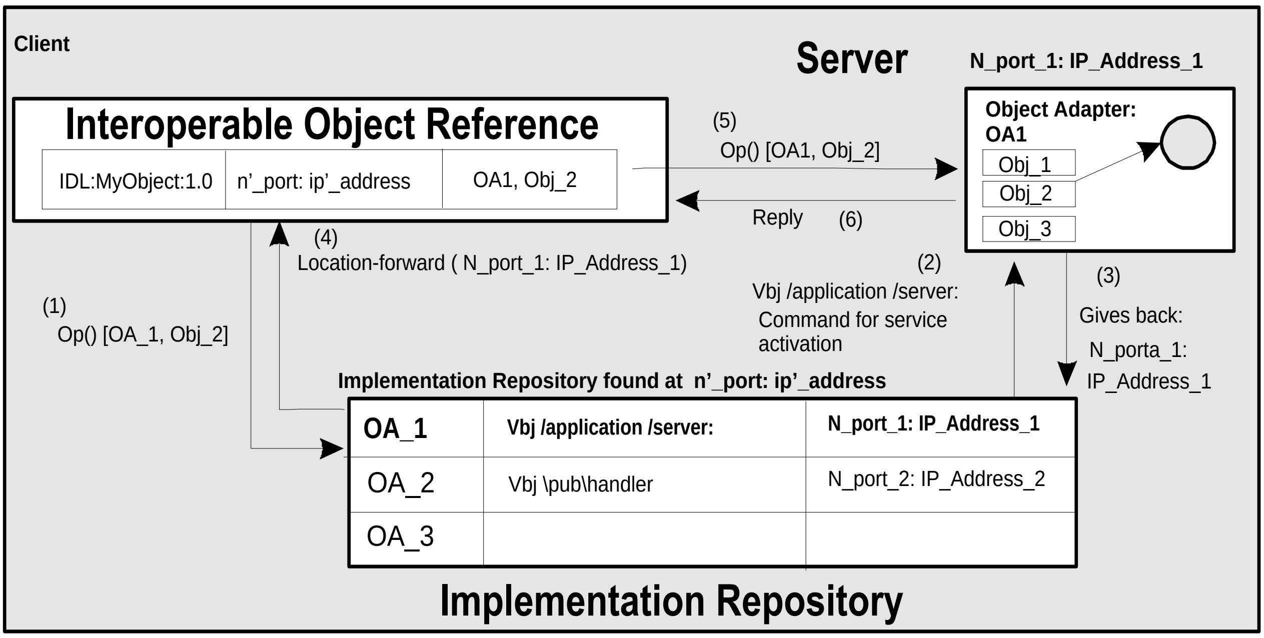
with the insertion of an **Implementation Repository** (registered the first time), called **Repository Identifier**

Direct link (direct binding) if **IOR** refers **directly** the related object (via POA)

The direct binding is for transient objects

IOR (indirect binding via POA)

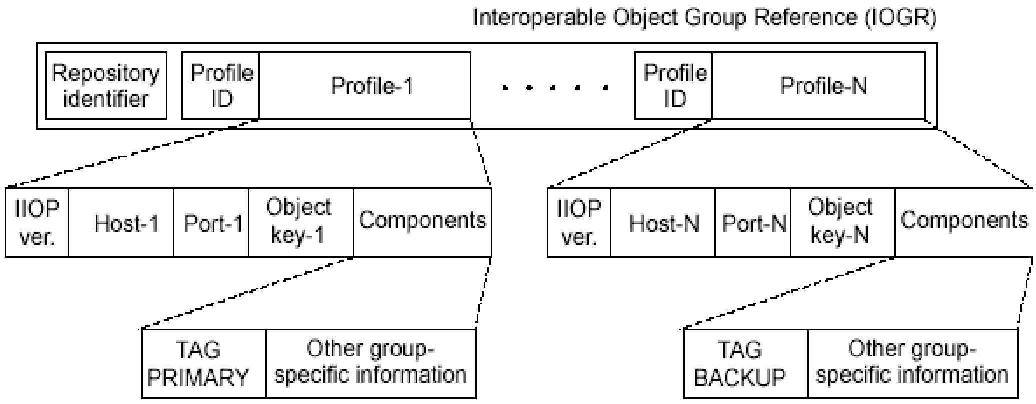
Indirect link (indirect binding)



EXSTENSIONS: Groups of objects

CORBA3 also defines the possibility of associating multiple copies to a service
with form of replication **completely transparent** to the client

Interoperable **O**bject **G**roup **R**eference (**IOGR**) provides copies and the ORB is responsible to find the **available copies** for all the requested functions
(also manage disconnection / reconnection / consistency)



OBJECT ADAPTER in CORBA

Object Adapter as **mediator agents** to overcome **heterogeneity** problems in different environments

BOA (Basic Object Adapter) as the early basic entity
BOA allowed server activation with only some simple policies and other more complex policies were totally implementation-dependent

Shared server	a **unique job** for a set of objects (unique shared activity)
Unshared server	a **job** for **every servant**
Persistent server	a **unique job** started at initialization or explicitly
Per Method server	a **job** for every invocation

PORTABLE OBJECT ADAPTER - POA

POA are **interoperable portable agent** that allows to pass from an **object reference** of a client to **real code** of the **servant** that must serve the same request

*A POA can manage **many different objects** and select which one to **direct the operations to***

In **different environments**, the **POA is different** (class, variables, methods, jobs) but *must realize basic policies necessaries to the variety of possible interactions*

In a **specific language environment**, there exist *a base class* from which **every POA inherits** that contains the mechanisms for request and servants management

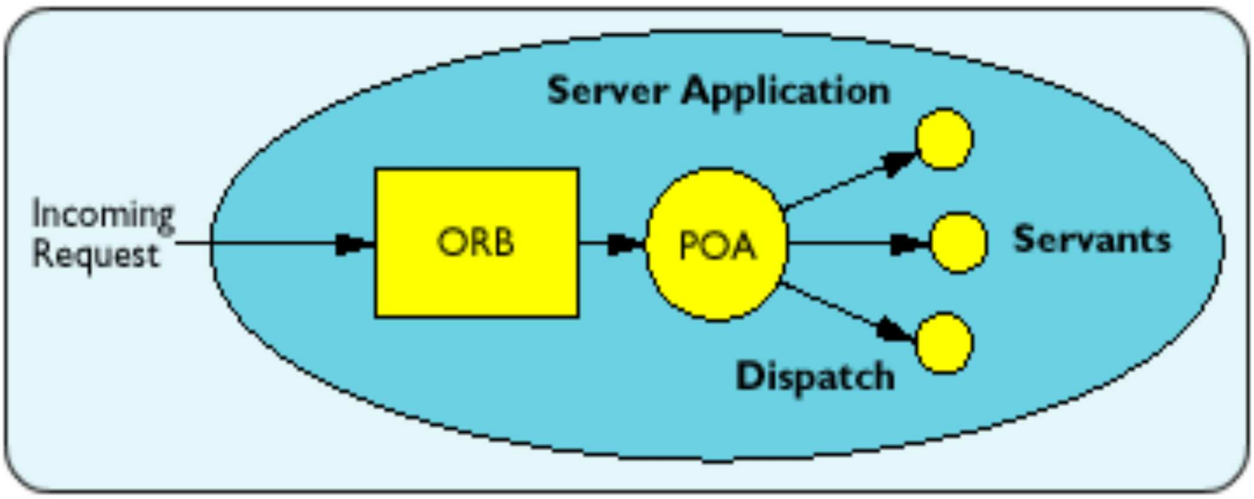
The *POA does not inherit the policies* defined for every case

PORTABLE OBJECT ADAPTER - POA

POA as portable agent for interoperability

It inherit from **other POAs** (by default) without inheriting policies that must be ad-hoc configured

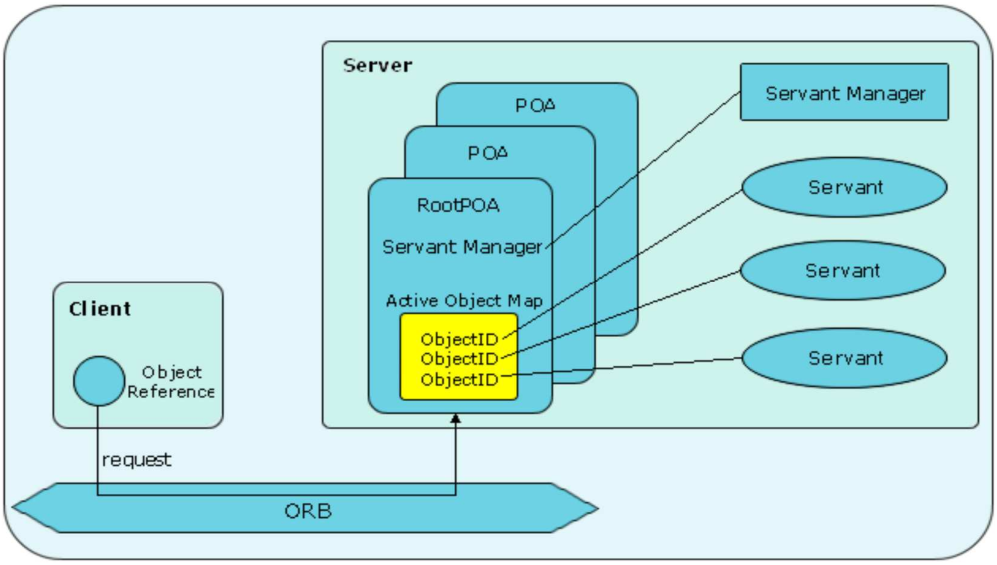
Policies also different and specialized



PORTABLE OBJECT ADAPTER

The POA has an its own **internal organization** (**AOM**)

POA uses an internal table, **Active Object Map**, to map servants (even the same several times)



PORTABLE OBJECT ADAPTER MANAGER

Any **POA** is managed by a **POA Manager** to implement suitable **management policies** (***mapping servants*** *and* ***object references***)

The **POA Manager** provide operations to manage **different policies** and also **change them.** The POA Manager allows to:

- **activate a POA** (to start the job)
- **deactivate a POA** (to close the work of a POA)
- **block the requests** to POAs (the job is stopped and no operation is started)
- **discard requests** to POAs (every incoming requests and the queued are discarded: no operations)

Only on a **deactivated POA, policies** can be **changed**

OBJECT ADAPTER in CORBA

A POA capable of managing its objects and servants: typically a **POA manage with the same policy any responsibility interface** (often **more than one**)

RESPONSIBILITY of

Object Reference Creation

ObjectID Identification (unique servant identifiers)

Manage related servants

Transient CORBA objects

that do not survive the application that has generated them

Persistent CORBA objects

that survive the application that has generated them and remain available also for subsequent applications

ADAPTER functions

The **POAs** have some **methods** visible to clients to register servants

`ObjectID` **`activate_object`** `(in` **`Servant p`**`);`

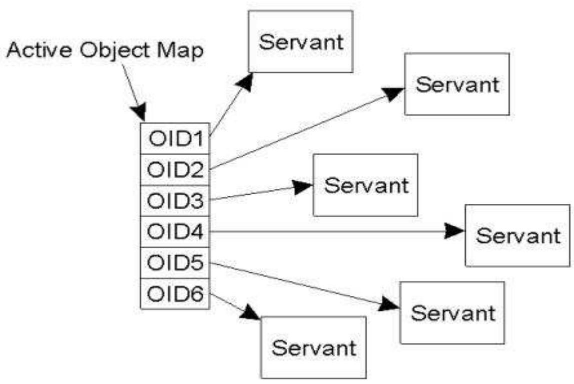
returns an object identifier and receive a pointer to a servant

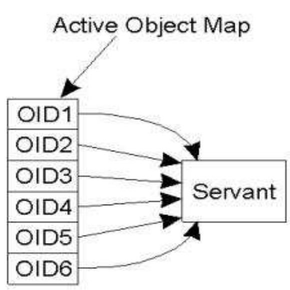
`void` **`activate_object_with_ID`**`(in` **`ObjectID oid`**`, in` **`Servant p`**`);`

associate a pointer to a servant to an entry inside the **Active Object Map**

The methods allows an explicit choice for servants inside the **AOM**

ObjectIDs allow servant choice inside POA





OBJECT ADAPTER in CORBA

POA is a portable agent for interoperability

the **policies** are ruled by **properties specified** ad-hoc with **standardized attributes**:

- **Thread** (ORB_CTRL_MODEL, SINGLE_THREAD_MODEL)
- **Lifespan** (TRANSIENT, PERSISTENT)
- **Object ID Uniqueness** (UNIQUE_ID, MULTIPLE_ID)
- **ID Assignment** (USER_ID, SYSTEM_ID)
- **Servant Retention** (RETAIN, NON_RETAIN)
- **Requests** (USE_ACTIVE_OBJECT_MAP_ONLY, USE_DEFAULT_SERVANT, USE_SERVANT_MANAGER)
- **Implicit Activation** (IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION)

ATTRIBUTES for OA in CORBA

POA expect **different values of attributes** that combined produce many very differentiated policies (default in red):

- **Thread** (ORB_CTRL_MODEL, SINGLE_THREAD_MODEL)
- **Lifespan** (TRANSIENT, PERSISTENT)
- **Object ID Uniqueness** (UNIQUE_ID, MULTIPLE_ID)
- **ID Assignment** (USER_ID, SYSTEM_ID)
- **Servant Retention** (RETAIN, NON_RETAIN)
- **Requests** (USE_ACTIVE_OBJECT_MAP_ONLY, USE_DEFAULT_SERVANT, USE_SERVANT_MANAGER)
- **Implicit Activation** (IMPLICIT_ACTIVATION, NO_IMPLICIT_ACTIVATION)

Retention and Request Processing Policy

- Retention policy: expect either the use or not of the AOM
  – **RETAIN**: memorization of every Object Id inside AOM
  – **NON_RETAIN**: **NOT** can be used **AOM** → use of Default Servant, or Servant Manager
- Request Processing policy: indicate the locating modality of servant objects to elaborate requests
  – **USE_ACTIVE_OBJECT_MAP_ONLY**: the dispatching happen for the servant objects registered to AOM
  – **USE_DEFAULT_SERVANT**: (if it is set a policy **NON_RETAIN**, or the servant object is not inside the AOM) the requests for **servant objects** not available inside the POA are delegated to a unique **servant**, called **Default Servant**
  – **USE_SERVANT_MANAGER**: policies of activation/deactivation of servant objects are in charge of a Servant Manager, specified and managed **directly by final user**

POA POLICIES

POA can contains more differentiated policies for servant objects management

Default POA policies:

Single Servant (for all objects)

Just one servant for every request (even for objects of different type)

Explicit Object Activation

Every specific servant is connected to an ObjectID, with servant control for service execution

On-Demand activation (only for a single method) stateless

On-Demand activation (for infinite duration)

the servant is activated by request and kept on for every subsequent request

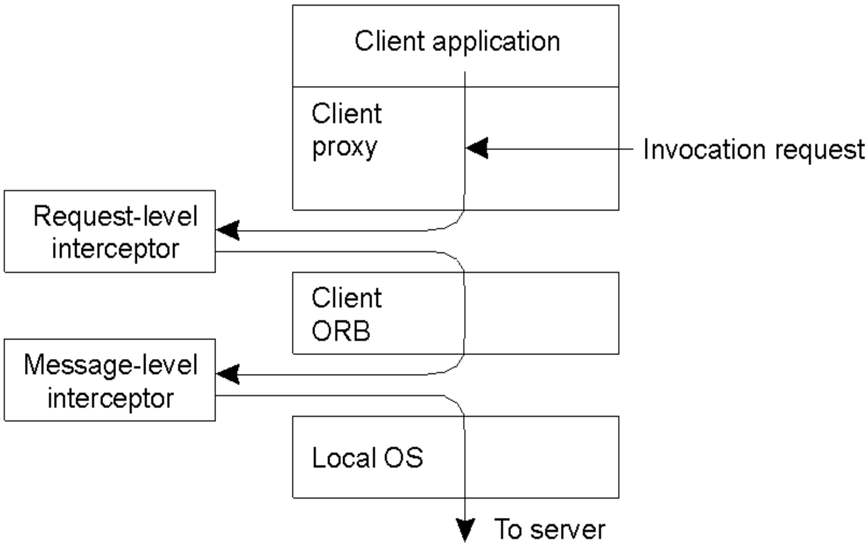
It is also possible any combination of policies

INTERCEPTOR in CORBA

To add services or functions transparently, interceptors are introduced without changing neither the server nor the client

They are used at different system levels

- application
- transport
  - security
  - transactions
  - …



CORBA 3

CORBA 3 introduces some significant areas of extension /completion

Internet

names as URL, firewall proxy for GIOP, …

QoS

new ways of invocations with more QoS control Asynchronous calls (**AMI**) & Time-independent (**TII**)

CORBA Real-time, CORBA reduced, CORBA fault-tolerant

Components

more abstract level to work in transparent way

INVOCATION SEMANTICS

In CORBA invocations are synchronous

The client must wait the operation completion from infrastructure

Static operations always synchronous (at-most-once)

Dynamic operations also less synchronous

one-way without result (best-effort)

no server response expected

deferred-synchronous deferred results (at-most-once)

the client can not wait for the answer
that the server make available afterwards
and the client can get successively

ASYNCHRONOUS INVOCATION - AMI

CORBA invocations are **not persistent and much coupled**

CORBAMessaging introduces an invocation strategy not available in standard CORBA

It is intended to **decouple:**

- **servant operations (with normal and synchronous result) from client invocation modalities**
- **lifetime** of the two environments

with **Callback** and **Polling** modality

The client interface is modified and it is possible to move requests and have different **interaction** from the synchronous one …but
the **client must define new additional operations**

POLL ASYNCHRONOUS INVOCATION

Asynchronous polling: the client decides **when and if to ask** for a completion verify method of the remote operation (obtaining the results). The **support** creates the poll object

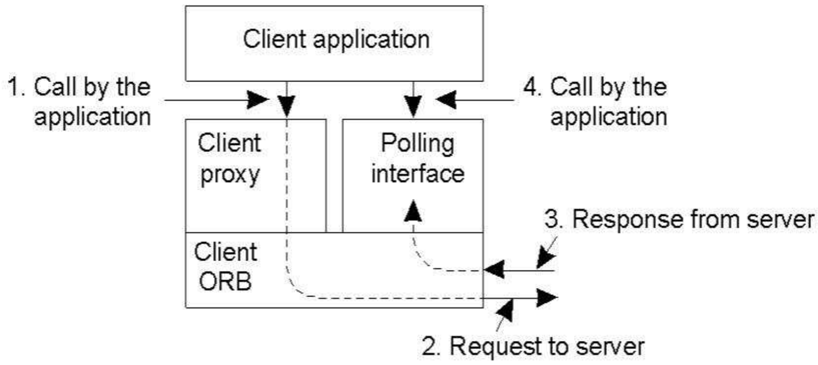
Rather than: `int sum (in int i, in int j, out int sum)`

`void sendpoll_sum (in int i, in int j, pollobj)`

`void pollsum (out int success, out int sum)`

For dealing with **polling**

the client invokes **sendpollsum** and when wants to recover the result invokes the **pollsum** operation, automatically generated by CORBA support



ASYNCHRONOUS INVOCATION - AMI

Callback: the client provides a **callback method** the support calls at completion via an **(automatic)** asynchronous operation

The static interface is modified:

`int sum (in int i, in int j, out int sum)`

`void sendcallback_sum (in int i, in int j, callbackobj)`

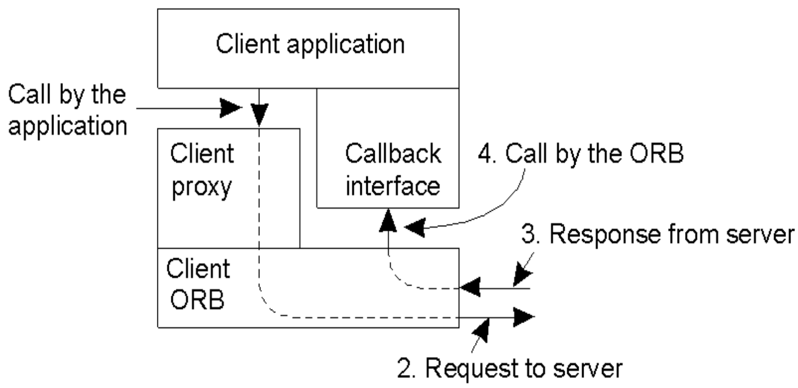
`void callback_sum (in int success, in int sum)`

We use two methods only changing the ***client implementation*** *and not the service part*

Client calls the **sendcallback_sum**

ORB invoke **callback_sum**

specified by user



MESSAGING

Possibility to define message QoS

Interface **RebindPolicy** to reestablish connection if broken
`TRANSPARENT, NO_REBIND, NO_RECONNECT`

Interface **SyncScopePolicy** to establish synchronization warranty

`SYNC_NONE, SYNC_WITH_TRANSPORT, SYNC_WITH_SERVER, SYNC_WITH_TARGET`

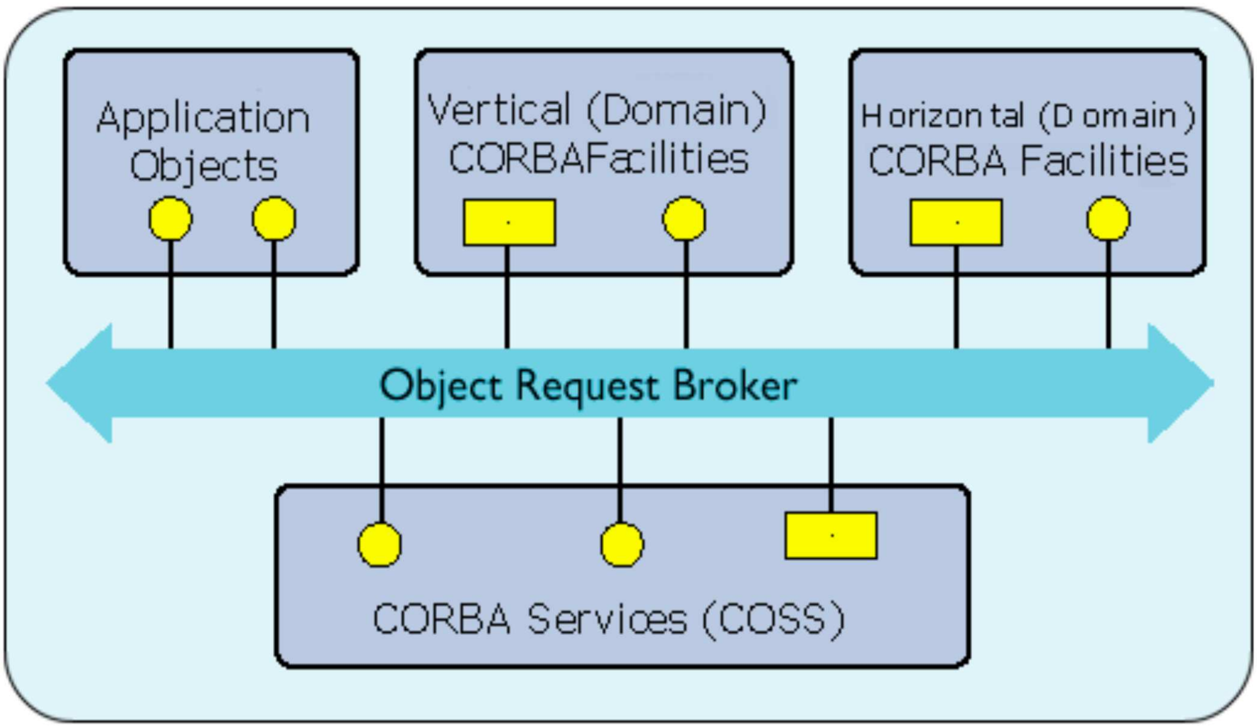
Interfaces **RequestPriorityPolicy** and **ReplyPriorityPolicy** for determine priority between the two sides of invocation, if necessary

Interface **QueueOrderPolicy** for manage priority in order requests

`ORDER_ANY, ORDER_TEMPORAL, ORDER_PRIORITY, ORDER_DEADLINE`

Other possibilities…

CORBA ARCHTECTURE



CORBA SERVICES

CORBA requests also other parts

The **CORBA Services** allow to provide support functions to obtain *more or less essentials* services

Collection service for group objects

Query service for query to interrogate objects

Concurrency (control) service for available lock services

Event service for using asynchronous events

Notification service events advanced management

The presence of these services qualify CORBA as a mature component integration environment

CORBA SERVICES

Service	**Description**
Collection	Facilities for grouping objects into lists, queue, sets, etc.
Query	Facilities for querying collections of objects in a declarative manner
Concurrency	Facilities to allow concurrent access to shared objects
Transaction	Flat and nested transactions on method calls over multiple objects
Event	**Facilities for asynchronous communication through events**
Notification	**Advanced facilities for event-based asynchronous communication**
Externalization	Facilities for marshaling and unmarshaling of objects
Life cycle	Facilities for creation, deletion, copying, and moving of objects
Licensing	Facilities for attaching a license to an object
Naming	**Facilities for systemwide name of objects**
Property	Facilities for associating (attribute, value) pairs with objects
Trading	**Facilities to publish and find the services on object has to offer**
Persistence	Facilities for persistently storing objects
Relationship	Facilities for expressing relationships between objects
Security	Mechanisms for secure channels, authorization, and auditing
Time	Provides the current time within specified error margins

CORBA SERVICES

OMG has standardized other components to simplify programming and support: **it is** (in practice) **necessary** the

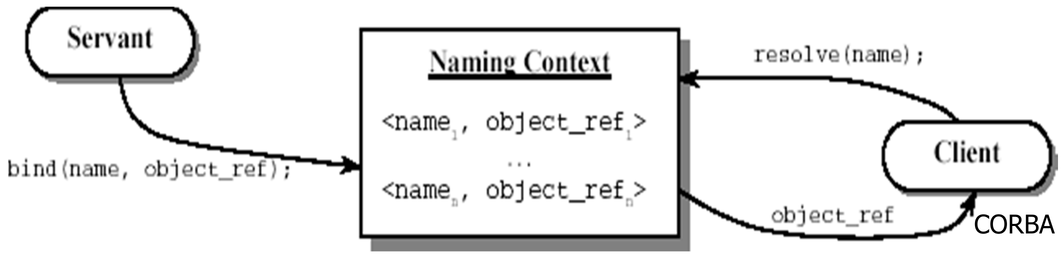
NAMING SERVICE

Mechanisms and strategies to support **persistent names** of CORBA, to classify and find **ObjectReference through logical names,** and to realize usable name systems

Name binding is an association between **object** and **name**

Name context as a binding set in which every name (of pairs) is unique

Bindings are specified, by definition, relative to a specific context

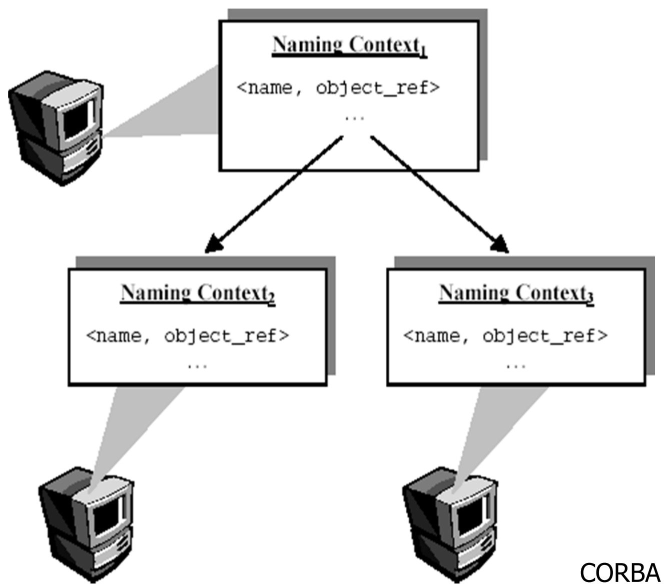


NAMING SERVICE

A **name** is structured as a sequence of **name components to identify the corresponding ObjectRef**

Different names can refer to ***different objects*** or to the ***same object*** finding it with a process of **resolution of different context even distributed**

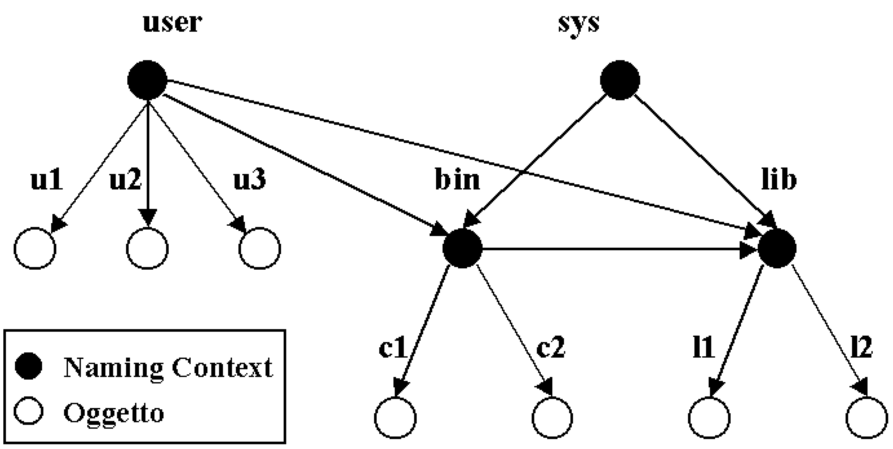
Names can also refer to federated contexts with **federated servers** (and different as context to manage) and **coordinated among them**



NAMING SERVICE

Context graphs are identified and containment relationships can create context to maintain references to objects (via ObjectReferences)

Contexts are dealt with as CORBA objects



NAMING SERVICE

A name **simple** or **composed** is a **sequence** of name components

Every **component** is constituted by two parts or attributes

[Identifier , Kind]

Identifier as Object Reference of CORBA Object type

Kind of descriptive type, for example *executable, postscript*

```
struct NameComponent {string id; string kind;};
typedef sequence <NameComponent> Name;
```

Only base mechanisms are provided over whom it is possible to build different user policies

The idea is that this service provide only mechanisms and do not impose policies of any kind

NAMING CONTEXT

The operations on a naming context derive from the NamingContext interface that specify the typical operations of a name system

```
interface NamingContext{
void bind(in Name n; in Object obj) raises …;
void rebind(in Name n; in Object obj) raises …;
void unbind(in Name n) …;
void bind_new_context(in Name n)…;
object resolve(in Name n)…;
void list(in unsigned long how_many,
   out BindingList bl, out BindingIterator bi);
}
```


TRADING SERVICE

The **TRADING Service** has the objective to ease the **search of services that implement a certain interface through specified attributes** (*feature similar to* ***yellow pages or …***)

The Trader is an object that **allows to keep** the **knowledge** of **services that can be requested (as logical names)**

The trader allows to ***expose services***

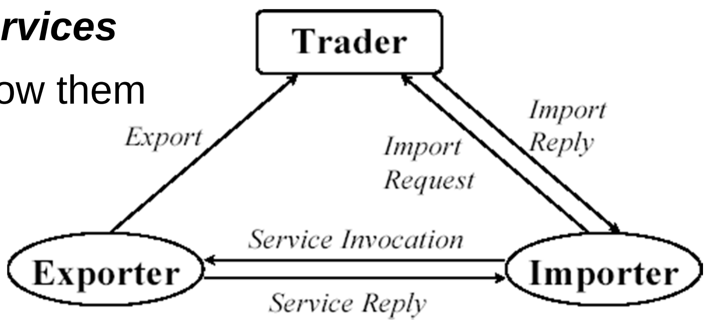
export from the provider

The trader allows to ***import services***

import from who want to know them

Obviously we can have

Federated Trader



TRADING SERVICE

A **search on a trader** allows to obtain an **unknown interface (the name is obtained)** through request of **characterizing features** (the result **obtains also more names**)

CORBA does specify nothing on **TRADING Service** implementation**: it is possible to realize it with a database or in-memory tables**

Every trader is characterized by

- **an interface that defines the features** exposed by the service
- **some properties** to represent behavioral aspects and non-functionals not expressed by the service interface

Every property is identified by an attribute **PropertyMode**

PropertyMode associated to a triple <**`name, type, mode`**>

```
enum PropertyMode {PROP_NORMAL, PROP_READONLY,
 PROP_MANDATORY, PROP_MANDATORY_READONLY}
```


TRADING SERVICE

Interface of a **Service associated to properties**

Every property is composed by <**`name, value`**>

With inheritance between services (and on considered interfaces)

```
service <ServiceTypeName>
  [:<BaseServiceTypeName> [,<BaseServiceTypeName>]*]
{interface <InterfaceTypeName>;
 [[mandatory] [readonly] property <IDLType>
           <PropertyName>;]*
};
```

The **publication** occurs by providing the **service name**, no one or more properties, **and** an implemented **interface(s) name**

A request we can obtain also more names (to be checked againdt the name service for `ObjectReferences`

EVENT & NOTIFICATION SERVICE

CORBA provides **synchronous one-to-one communication**
EVENT SERVICE makes it **more asynchronous and flexible**

Events may **have** content: **value** or references to an **object**

The information can be **generic** o **typed**

Considering basic the usual **mutual knowledge of interface**, we can consider events **supplier** and **consumer**, with different communication modalities

- **direct communication** or
- **indirect communication mediated by channels**

and **2 communication models**

- **Push** modality suppliers send to consumers
- **Pull** modality consumers send to suppliers (on need)

EVENT & NOTIFICATION SERVICE

CORBA considers an event management either direct or through **event channels,** as mediators enabler

If a consumer is not registered to the channel and executes registration at a certain time, ***every previous event is lost***

Every registered consumer *receives* ***every event*** *that occurs*

The events are **not persistent**

not reliable

without filtering capabilities

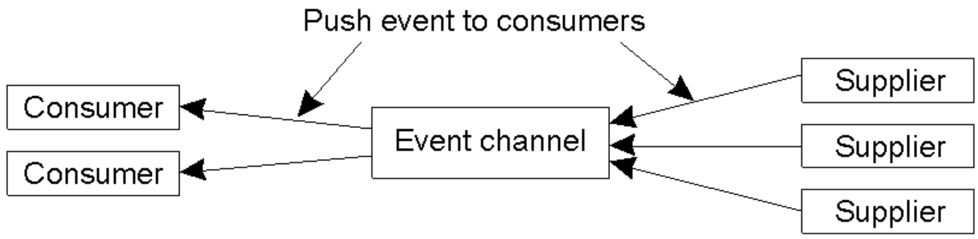
But introduce a **communication model change**
There are proposals for introducing reliability and filtering notifications

INTERFACCE EVENT: Push modality

In **push** modality, consumers and suppliers know each others directly or indirectly via channels and interfaces are defined

```
interface PushConsumer                {
  void push (in any data) raises(Disconnected);
  void disconnect_push_consumer();   };
interface PushSupplier                {
  void disconnect_push_supplier();   };
```

Disconnects can also terminate and block the communication

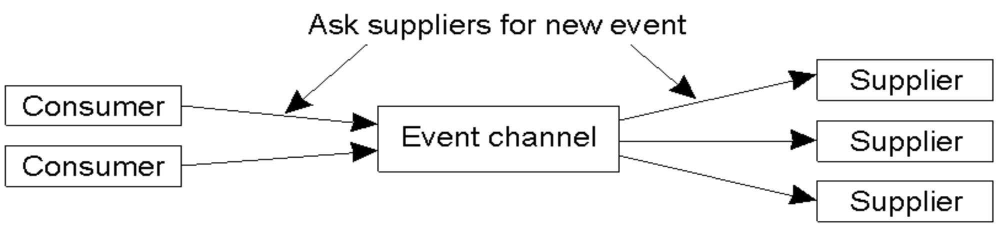


INTERFACCE EVENT: Pull modality

In **pull** modality, via standard interfaces consumers and suppliers know each others directly (or via channels)

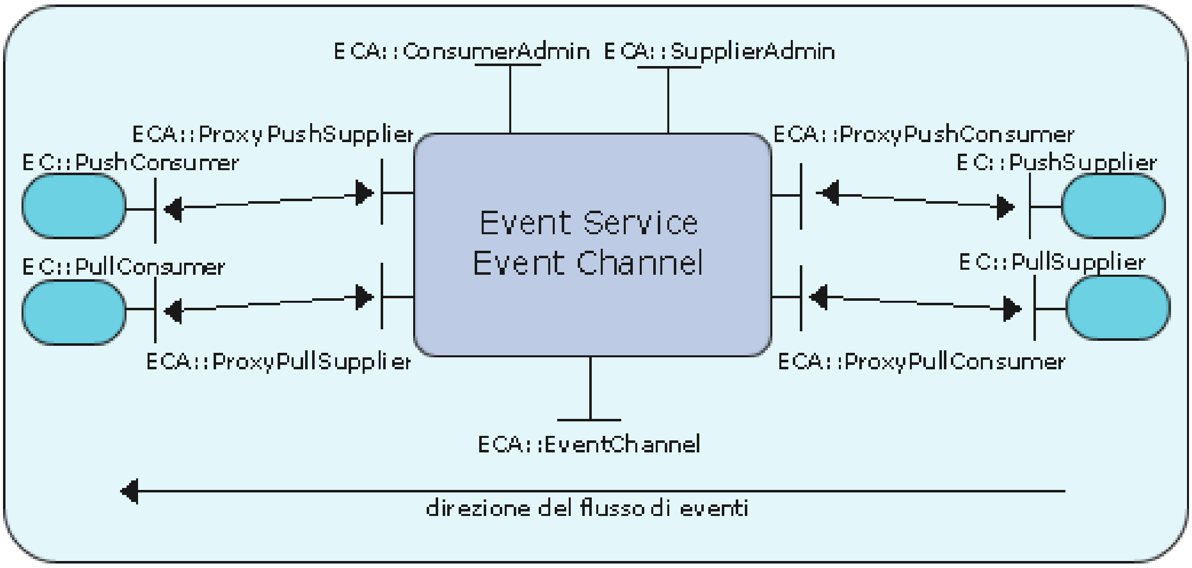
```
interface PullSupplier                {
  void pull () raises(Disconnected);
  void try_pull (out boolean event)
                  raises(Disconnected);
  void disconnect_pull_supplier();   };
interface PullConsumer              {
void disconnect_pull_consumer();   };
```

Disconnect operations can stop operations



EVENT & NOTIFICATION SERVICE

Event Channel object for many-to-many communication



EVENT SERVICE: LIMITS

Event Channel allows and enable
many-to-many communication

The **Channel** has the ability to coordinate also *more possible supplier* before trigger *multiple events* on different consumers, but

- **does not introduce filtering** on receivers
- does not provide **quality of service** of the communication (not durably maintained, permanently, …, depend on specific implementation)

The **Notification service** extends event service with these new significant features

event description and information, *filters and filters repository*

NOTIFICATION SERVICE

Events can be denoted by properties that allow new attributes (**Header** and **Body** on which it is possible to filter)

- **Reliability** (best-effort, persistent), **Priority**, **StartTime**, **Stoptime**, **Timeout**

It is available also an **Event type repository** for their description