CORBA
Common Object Request Broker Architetture

Gabriele Di Stefano

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica
Università degli Studi dell'Aquila, Italy
gabriele.distefano@univaq.it

Corso per Thales - Chieti

Introduzione a CORBA

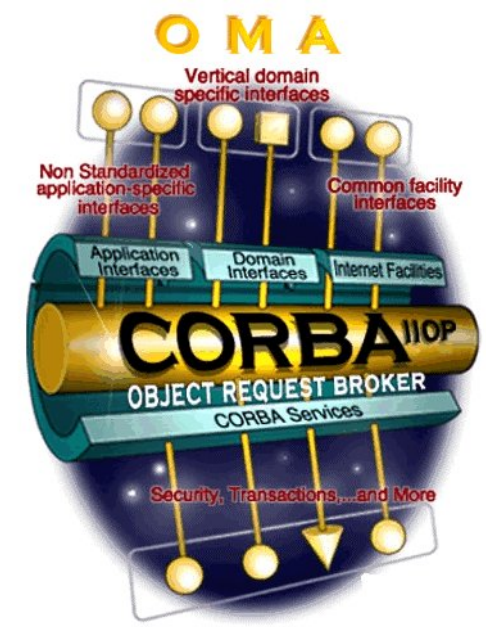
Cosa vedremo:

- Overview
  - L'OMG
  - Il Middleware
  - CORBA: panoramica
- I componenti di CORBA
  - L'Object Management Architecture (OMA)
  - Interface Definition Language (IDL)
  - ORB core
  - Object Adapters /Portable Object Adapter (POA)
  - CORBA Interoperability
- CORBA services
  - Naming Service
  - Implementation Repository
  - Event Service

Queste slides possono essere scaricate da:
http://gs.ing.univaq.it/CORBA.

Common Object Request Broker Architecture (CORBA)

Cos'è CORBA?



CORBA è uno standard definito dall'Object Management Group (OMG)

- specificare il funzionamento dell'ORB: meccanismo base mediante il

Object Management Group (OMG)

L'OMG:

- Organizzazione non-profit fondata nel 1989 da undici compagnie (IBM, HP, Apple, SUN, ...)
- Attualmente: *"Our members include hundreds of organizations including software end-users in over two dozen vertical markets (from finance to healthcare and automotive to insurance) and virtually every large organization in the technology industry."*
- Finalità: *"OMG Task Forces develop enterprise integration standards for a wide range of technologies and an even wider range of industries. OMG's modeling standards, including the Unified Modeling Language (UML) and Model Driven Architecture (MDA), enable powerful visual design, execution and maintenance of software and other processes."*
- Sede: 109 Highland Ave, Needham, MA 02494 USA.

Il Middleware

Definition (MIDDLEWARE)

Insieme degli strumenti che permettono l'integrazione di diverse applicazioni e servizi da utilizzare in ambienti aperti.
I middleware comprendono strumenti per lo sviluppo fino al livello di applicazione

Definition (MIDDLEWARE)

Middleware is software that enables interprocess communication. It provides an API that isolates the application code from the underlying network communication formats and protocols.

Tipi di Middleware

- RPC Middleware
- Message Oriented Middleware (MOM)
- Distributed Transaction Processing (TP) Monitor
- Database Middleware
- Distributed Object Computing (DOC) Middleware
- Special purpose: Mobile & Qos Multimedia Middleware, Agent-based Middleware...

Remote Procedure Call (RPC) Middleware

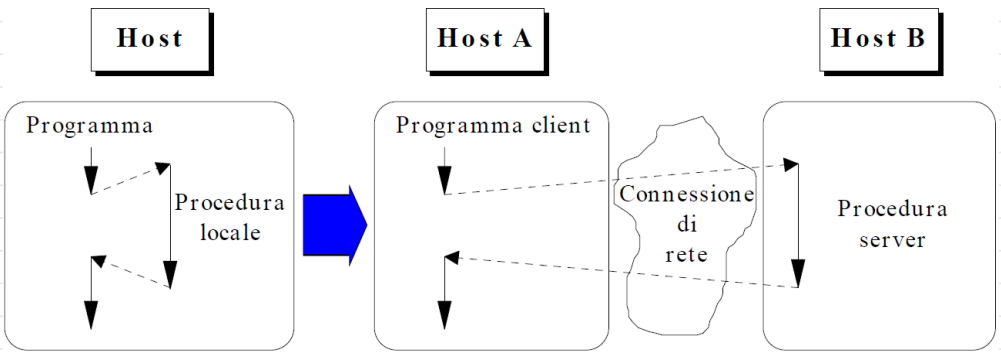
L'RPC è uno strumento per implementare applicazioni distribuite di tipo client/server.
Caratteristiche:

- Interface Definition Language (IDL) spesso usato per definire l'accordo
- Sincronicità: il client bloccato in modo sincrono in attesa della risposta dal server
- Gestione eterogeneità dei dati
- Uso di Stub per la trasparenza
- Binding spesso statico (o poco dinamico)

Modello piuttosto rigido, poco scalabile e poco replicabile.
Il server deve essere presente e prevedere i processi necessari in modo esplicito.
Non si tiene conto di eventuali ottimizzazioni nell'uso delle risorse

Nascita ed evoluzione della RPC

Estensione del normale meccanismo di chiamata a procedura.

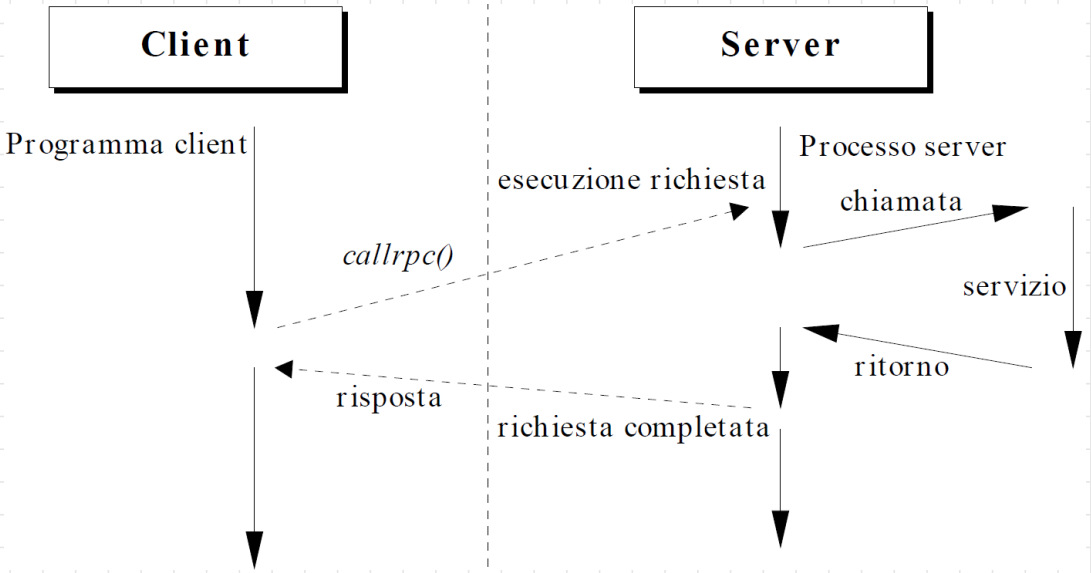


Il client invia la richiesta ed attende fino alla risposta del servitore stesso.
A differenza della chiamata a procedura locale:

- i processi non condividono lo spazio di indirizzamento
- i processi hanno vita separata
- possono accadere malfunzionamenti, sia ai nodi, sia alla interconnessione

Nascita ed evoluzione della RPC

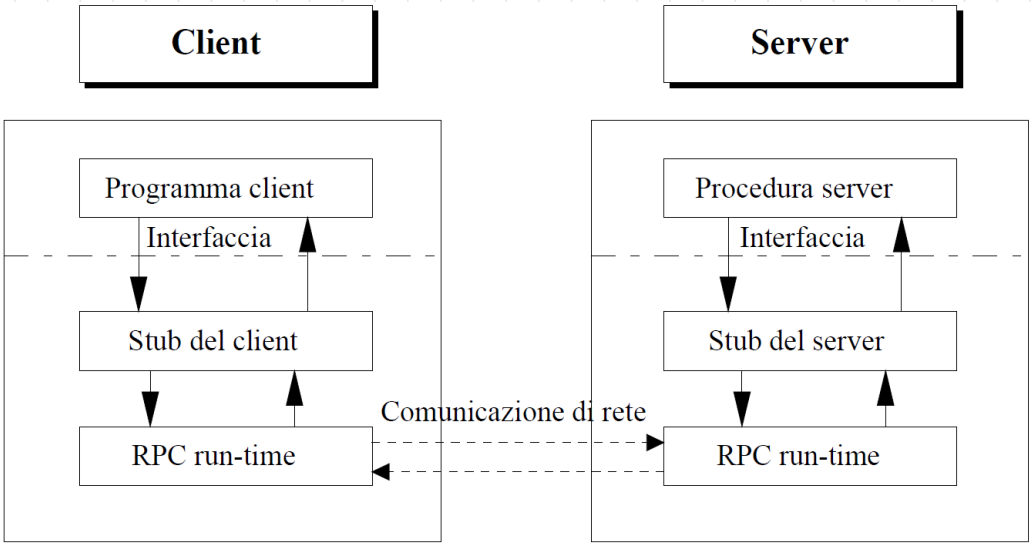
Schema non trasparente:



Il servizio è esplicitamente realizzato da un unico processo sequenziale o affidato a processi indipendenti generati ad ogni richiesta.

Nascita ed evoluzione della RPC

Uso di interfacce per la trasparenza.



Uso di stub: componenti che servono a superare i problemi di trasparenza. Lo sviluppo con stub prevede trasparenza: gli stub sono in genere prodotti "automaticamente": l'utente progetta solo le parti applicative

RPC: Passaggio parametri

Passaggio per valore o per riferimento.

Trattamento dei parametri per valore:
Serializzazione dei parametri (marshalling), trasferimento deserializzazione (unmarshalling). Poi il valore viene perso.

Trattamento dei parametri per riferimento:
uso di oggetti che rimangono su un nodo e devono essere identificati in modo unico nell'intero sistema.

RPC: Affidabilità (Reliability)

Malfunzionamenti:

- perdita di messaggio di richiesta o di risposta
- crash del nodo del cliente
- crash del nodo del servitore

In caso di crash del servitore, prima di fornire la risposta il cliente può:

- aspettare per sempre;
- time-out e riportare una eccezione al cliente;
- time-out e ritrasmettere (uso identificatori unici);

Semantica di funzionamento di una RPC:

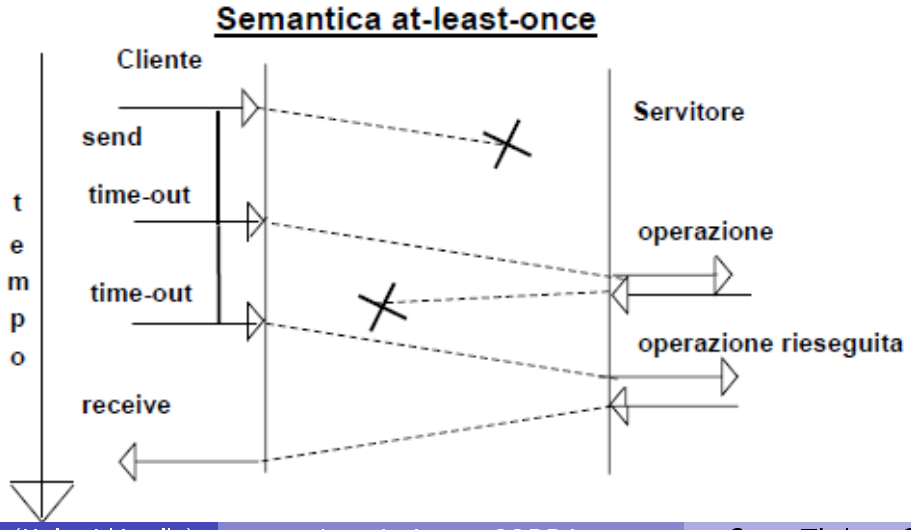
may-be: time-out per il cliente

at-least-once: time-out e ritrasmissioni

at-most-once: tabelle delle azioni effettuate lato server

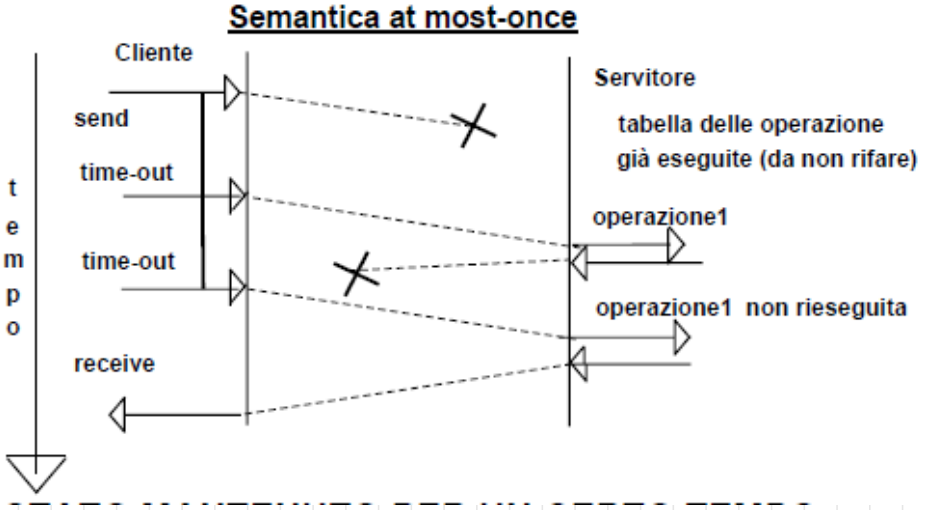
At least once

Strategia: più invii ad intervalli
Il messaggio può arrivare più volte a causa delle ritrasmissioni $\Longrightarrow$ semantica adatta per azioni idempotenti.
Implementazione:
il cliente fa ritrasmissioni (quante?, ogni quanto? ...) il server non se ne accorge



At most once

Strategia: più invii ad intervalli
Il messaggio può arrivare più volte a causa delle ritrasmissioni $\Longrightarrow$ il messaggio, viene considerato al più una volta $\Longrightarrow$ il server deve mantenere uno stato.
Implementazione:
il cliente fa ritrasmissioni (quante?, ogni quanto? ...) il server se ne accorge



Remote Procedure Call: funzionamento

Sequenza di eventi durante una RPC:

1. Il client chiama il client stub. La chiamata è locale e i parametri sono messi sullo stack.
2. Il client stub impacchetta i parametri (marshalling) in un messaggio ed esegue una system call per spedire il messaggio.
3. Il sistema operativo locale al client manda il messaggio alla macchina server.
4. Il sistema operativo locale al server passa il pacchetto al server stub.
5. Il server stub spacchetta i parametri (unmarshalling).
6. Il server stub chiama la procedura "remota".
7. I parametri di ritorno sono mandati al client con la stessa procedura in direzione opposta.

Message Oriented Middleware (MOM)

La distribuzione dei dati e codice avviene attraverso lo scambio di messaggi. Scambio messaggi tipizzati e non tipizzati sia sincrono sia asincrono con strumenti ad-hoc.

Caratteristiche:

- ampia autonomia tra i componenti
- asincronicità e persistenza delle azioni
- gestori (broker) con politiche diverse e QoS diverso
- facilità nel multicast, broadcast, publish / subscribe

Distributed Object Computing (DOC) Middleware

La distribuzione dei dati e codice avviene attraverso richieste di operazioni tra clienti e servitori remoti.

Uso di oggetti come framework e di un broker come intermediario nella gestione delle operazioni.

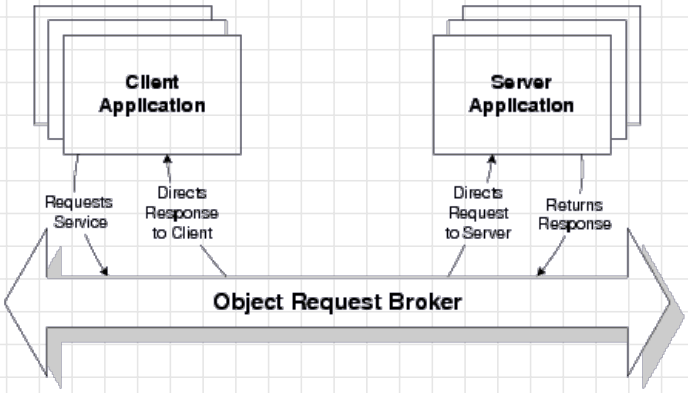
Caratteristiche:

- il modello ad oggetti semplifica il progetto
- il broker fornisce i servizi base e molti servizi addizionali
- alcune operazioni si possono fare in modo automatico
- la integrazione di sistemi è facilitata e incentivata
- la tecnologia open source è favorita

A middleware: CORBA

CORBA è un acronimo per Common ORB Architecture.
Common Architecture sta per standard, quindi CORBA è uno standard per qualcosa chiamato ORB.

ORB è l'acronimo di Object Request Broker, cioè un meccanismo intermediario per richiamare metodi di oggetti di un processo remoto, in esecuzione sullo stesso o un altro computer. A livello di programmazione, questa chiamata sarà simile ad una chiamata locale.



Caratteristiche di CORBA

Caratteristica principale di CORBA è di essere un *middleware distribuito orientato agli oggetti*. In particolare, permette alle applicazioni di comunicare anche se sono:

- su differenti computer;
- su differenti S.O.;
- su differenti tipi di CPU (indipendenza da codifica big-endian o little-endian, 32-bit e 64-bit);
- implementate con differenti linguaggi di programmazione come C, C++, Java, Smalltalk, Ada, COBOL, PL/I, LISP, Python...

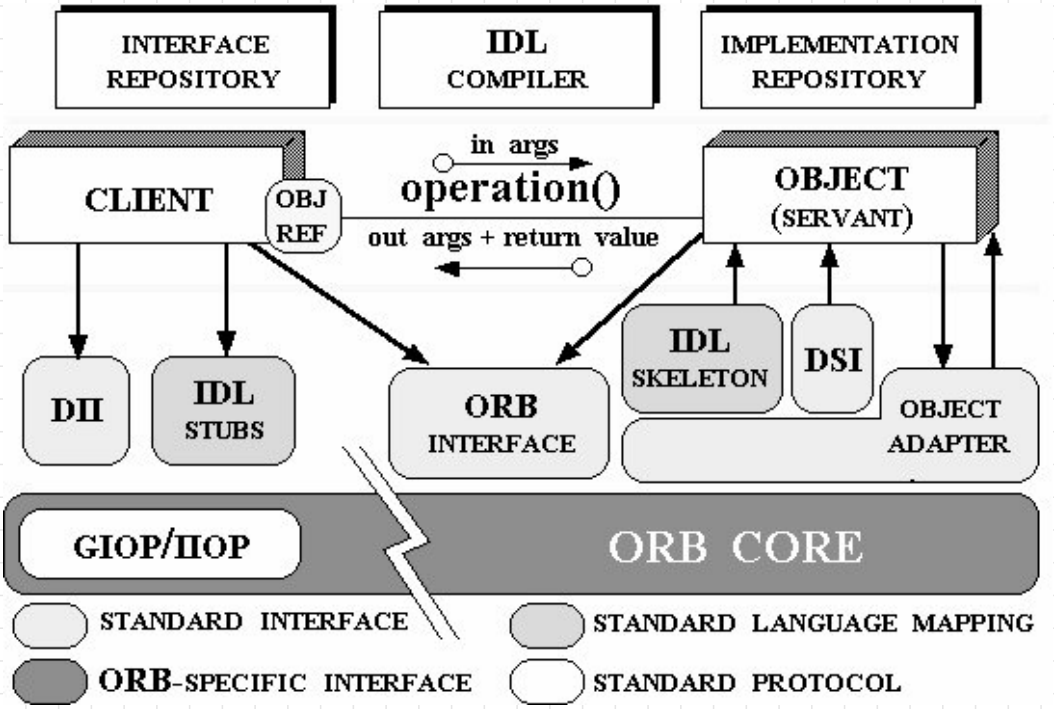
Caratteristiche di CORBA

Essendo CORBA orientato agli oggetti, un client non fa chiamate ad un processo sever. Un CORBA client chiama i metodi di un oggetto (che vive in un processo server).

Per cui lo standard CORBA fornisce:

- definizione di una interfaccia per oggetto;
- definizione della interazione tra oggetti;
- bus per integrazione di oggetti progettati in linguaggi diversi;
- definizione della interazione anche tra sistemi diversi con diversi gestori.

Vista generale



L'Object Management Architecture (OMA)

La descrizione dell'architettura CORBA è derivata dall'*abstract Core Object Model* definita dall'OMG nel documento *Object Management Architecture (OMA) Guide*.

L'Object Model di CORBA fornisce una presentazione organizzata dei concetti e della terminologia legata agli oggetti.

L'Object Model definisce un modello parziale di computazione che incorpora le caratteristiche chiave degli oggetti realizzate dalle sottostanti tecnologie.

Esso è un esempio di object model classico, dove un client manda un messaggio ad un oggetto.

Concettualmente, l'oggetto interpreta il messaggio e decide cosa fare. Come nei modelli classici, un messaggio identifica un oggetto e zero o più parametri attuali. È richiesto un parametro speciale che identifica l'operazione da eseguire.

Concetti estranei all'OMA

Alcuni concetti sono al di fuori dello scopo del modello. Ad esempio:

- gli oggetti composti;
- la realizzazione di collegamenti tra oggetti;
- la copia degli oggetti.

Sono anche al di fuori i dettagli della stuttura di controllo:

- nessuna specifica su client o server single-threaded o multi-threaded;
- né su creazione o distruzione di threads;
- e neanche sulla programmazione dell'*event loop*.

Concetti in OMA

I principali costituenti dell'Object Management Architecture sono:

Cliente: qualsiasi entità capace di richiedere un servizio.

Oggetto: entità identificabile e incapsulata che fornisce uno o più servizi.

Creazione o Distruzione di oggetti: Gli oggetti possono essere creati e distrutti. Dal punto di vista del cliente, non c'è un meccanismo per creare o distruggere oggetti. Gli oggetti sono creati e distrutti in conseguenza di emanazioni di richieste. La creazione di un oggetto è resa palese al cliente nella forma di un riferimento ad oggetto che denota il nuovo oggetto.

Richiesta: meccanismo per manifestare l'esigenza di un servizio. Implica una sequenza di eventi causalmente correlati tra un client e un server.

Concetti in OMA (segue)

...

Valore: qualsiasi entità che può fungere da parametro attuale in una richiesta. È un'istanza di un tipo di dato IDL. Ci sono valori che non sono oggetti, così come valori che si riferiscono ad oggetti.

Riferimento ad oggetto: valore che denota un particolare oggetto. Un oggetto può essere denotato da distinti riferimenti.

Operazione: servizio con nome che può essere richiesto ad un oggetto

Parametro: entità per passare dati con oggetti destinatari di una richiesta. Possono essere di input, output o input-output.

Valore di ritorno: risultato (singolo se presente) dell'esecuzione di una richiesta.

Eccezione: entità di ritorno al client in caso di occorrenza di una condizione anomala nell'esecuzione di una richiesta. L'eccezione può portare parametri di ritorno.

Concetti in OMA (segue)

...

Tipo: entità identificabile con associato un predicato. Un'entità per cui il predicato è vero è un *membro del tipo*. L'*estensione di un tipo* è l'insieme di entità per cui il predicato è vero. Un tipo di oggetto è un tipo i cui membri sono riferimenti ad oggetti.

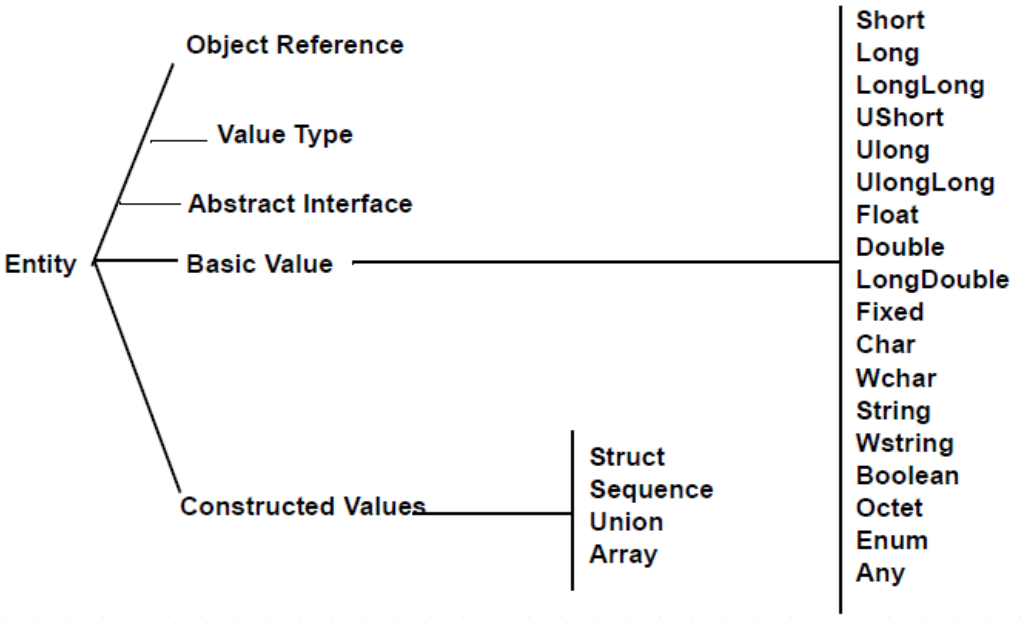
Interfaccia: descrizione delle operazioni possibili per un insieme di oggetti.

Tipi base: Es.: `Short`, `Long`, `Float`, `Boolean`, `String`,..., ma anche `Any` per indicare un tipo qualsiasi.

Strutture dati: Es.: `Struct`, `Array`, `Union`, `Sequence`

Tipo valore (*value type*): specifica lo stato e l'insieme di operazioni di un'istanza di un tipo. È utilizzato principalmente per passare oggetti per valore.

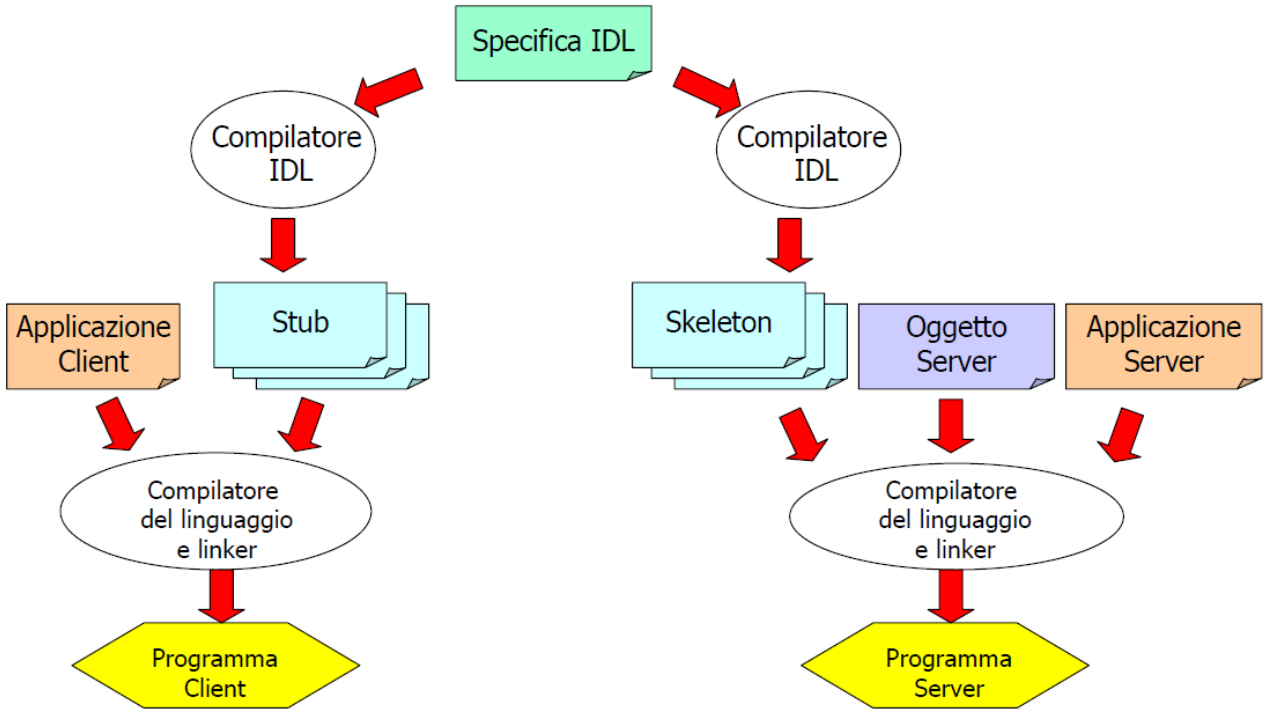
Entità definite nell'Object Model di CORBA



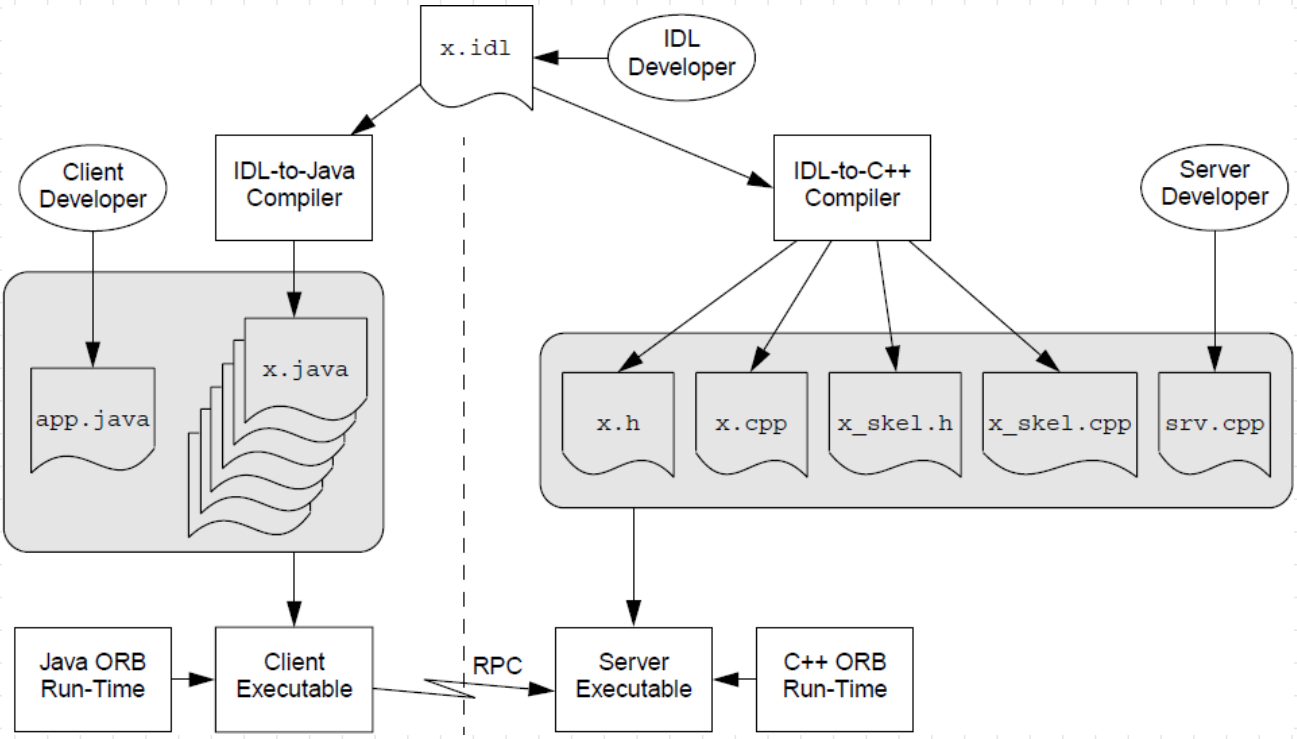
Interface Definition Language (IDL)

- La motivazione principale per IDL è quella di risolvere l'eterogeneità nelle architetture distribuite.
- L'eterogeneità tra i linguaggi di programmazione possono sorgere perché, per esempio, i loro costrutti e caratteristiche differenti, o loro rappresentazioni codice macchina differiscono.
- Lo standard CORBA supporta un modello di oggetto comune e un Interface Definition Language (IDL), che rappresenta una chiave per la risolvere l'eterogeneità dei linguaggi di programmazione.
- CORBA definisce vincoli di programmazione per i linguaggi di programmazione disponibili verso le specifiche dell'IDL:
  - I tipi di Oggetto sono mappati su stub e skeleton.
  - I riferimenti agli oggetti dei server sono incapsulati negli stub dei client.
  - Le operazioni sono mappate in operazioni o metodi dei linguaggi di programmazione.

Compilazione di un file IDL



Compilazione IDL (Mixed Languages)



Esempi di interfaccia

Esempi di interfacce IDL:

```
module HelloApp
{

  interface Hello
  {
    string sayHello();         // This line is an operation statement.

    oneway void shutdown();    // This line is another
  };

};
```

```
module Calculator
{

  interface Functions
  {
    float square_root (in float number);

    float power (in float base, in float exponent);
  };
};
```


Compilazione in Java

IDL:

```
module HelloApp
{

  interface Hello
  {
    string sayHello();         // This line is an operation statement.

    oneway void shutdown();    // This line is another
  };

};
```

Java implemetazione (automatica):

```
package HelloApp; //Un modulo IDL diventa un package Java

// Generated by the IDL-to-Java compiler (portable), version "3.2"

public interface HelloOperations
{
  String sayHello (); // This line is an operation statement.

  void shutdown ();
} // interface HelloOperations
```

Costrutti base dell'IDL: Moduli

I moduli servono per:

- Creare spazi di nomi (name spaces) per le definizioni IDL
- Definire un ambito comune per interfacce IDL.
- Contenere la definizione di una o più interfacce.
- Contenere eventuali altri moduli annidati.

Esempio:

```
module Assembly {

    ......   //definizione di Interfacce

};
```


Costrutti base dell'IDL: Interfacce

Le interfacce sono il costrutto principale dell'IDL.

- Sono un confine software tra implementazione di un servizio e client.
- Possono ereditare da altre interfacce
- Possono includere attributi, operazioni e definizioni di tipi.

Esempio

```
module Assembly {

   interface Account {

     ... //Account definitions

   };

   interface Savings: Account {

     ... //Inherits all Account definitions
     ... //then adds Savings definitions

   };

};
```

Costrutti base dell'IDL: Attributi

Le interfacce possono definire attributi (!)

- Gli attributi definiscono caratteristiche generali per una interfaccia.
- Il compilatore IDL definisce metodi *get* e *set* per ogni attributo.
- Gli attributi possono essere read-only o read-write.

Esempio

```
...
   interface Account {
     attribute string balance;    // Il compilatore genera due metodi
     readonly attribute long account_number;  //Il compilatore genera
                                              //un solo metodo get
     ......
   };
```


Costrutti base dell'IDL: Attributi

Compilazione in Java (usando `idlj`)

```
 public interface AccountOperations
{
  String balance ();
  void balance (String newBalance);
  int account_number ();
}
```

Compilazione in C++ (usando `omniidl`)

```
class _objref_Account :...
{
public:
  char* balance();
  void balance(const char* _v);
  ::CORBA::Long account_number();
  ...
}
```

Costrutti base dell'IDL: Tipi base

L'IDL è fortemente tipizzato, permette di ridenominare tipi e crearne tramite: enumeration, structures, arrays, sequences, unions.
Tipi Base: l'IDL supporta la maggior parte dei tipi dei linguaggi come il C, il C++ e Java.

IDL Basic Types	Notes
short, unsigned short	Integer-16 bits
long, unsigned long	Integer-32 bits
long long, unsigned long long	Integer-64 bits
float	IEEE single-precision floating point
double	IEEE double-precision floating point
long double	IEEE double-extended floating point
char	8-bit quantity
wchar	Encode wide characters from any character set (size undefined)
boolean	True or false
Octet	Byte- is guaranteed not to undergo any conversion when it passes over the wire.
string, wstring	Are basically array of chars and wchar

Tipi IDL: Ridenominazione, costanti, enumeration, struct

La ridenominazione con la keyword typedef.

Le costanti IDL sono espresse tramite la keyword const.

Sono ammessi tipi enumerativi e strutture.

```
...
    typedef string LastName;

    const LastName my_lastname = "Smith";

    enum ChargeCard {MasterCard, Visa, Diners};

    struct GuestRecord {
      GuestName name;
      Address address;
      PhoneNumber number;
    };
...
```

Tipi IDL: Array, Sequenze, Any

Array: sequenza limitata di elementi di tipi IDL.

Sequence: array non necessariamente limitati.

Any: Consentono la definizione di valori di dati liberamente tipizzati. Utile per definire interfacce riutilizzabili.

```
typedef EmployeeList Employees[100];
typedef sequence <GuestRecord> record; //unbounded sequence
typedef sequence <GuestList,10> list; //bounded sequence
typedef any DynamicallyTypedValue;

struct RunTimeValue {
  string description;
  any run_time_value;
 };
```


Tipi IDL: Union

Union: incrocio tra il tipo union e l'istruzione switch del C.

```
enum PersonKind {A_GUEST, AN_EMPLOYEE, OTHER}
struct GuestRecord { ... }
struct EmployeeRecord { ... }
union Person switch (PersonKind) {
  case A_GUEST:  GuestRecord guest_record;
  case AN_EMPLOYEE:  EmployeeRecord employee_record;
default: string description;
};
```

Il tipo del discriminante deve essere `integer`, `char`, `boolean` o `enum`.

Il valore di una variabile è dato dal discriminante e dal valore del campo attivo oppure dal valore del campo di default.

L'accesso al discriminante e al relativo campo è "language-mapping dependent".

Eccezioni IDL

Definiscono valori passati dall'interfaccia in caso di malfunzionamenti.

Possono contenere dati accessibili come attibuti pubblici.

Le eccezioni sono dichiarate in modo simile alle strutture IDL.

```
...
  exception CardExpired {string expiration_data;};

  exception CardReportedStolen {
    string reporting_instructions;
    unsigned long hotline_phone_number;
  };
```

Ci sono due tipi di eccezioni: le ecccezioni definite dall'utente e quelle definite da CORBA, chiamate anche Standard Exceptions.

Operazioni in IDL

```
interface Thermometer {
   string get_location();
   void set_location(in string loc);
};
```

Invocare un'operazione su un'istanza di interfaccia, manda una RPC al server che implementa l'istanza. Di default le operazioni sono sincrone di tipo *at-most-once*.

Ogni definizione di operazione ha:

- un nome dell'operazione
- un tipo di ritorno (`void` se non c'è)
- zero o più parametri

Eventualmente una operazione può avere:

- una dichirazione `raises`, per le eccezioni
- una dichiarazione `oneway`
- una clausola di `context`.

Non si può fare l'overload delle operazioni.

Passaggio parametri

Esempio di interfaccia per illustrare il passaggio parametri:

```
interface NumberCruncher {

  double square_root(in double operand);

  void square_root2(in double operand, out double result);

  void square_root3(inout double op_res);
};
```

I parametri sono qualificati con le keywords: `in`, `out`, o `inout`.

- **in:** Il parametro è mandato dal client al server.
- **out:** Il parametro è mandato dal server al client.
- **inout:** Il parametro è mandato dal client al server, forse modificato dal server e restituito al client.

Passaggio parametri: implementazione Java

Molti linguaggi, come Java, non prevedono parametri in output. Si risolve il problema mediante classi "Holder" i cui oggetti mantengono al loro interno un attributo `value` di tipo uguale a quello del parametro.

Esempio:

```
public interface NumberCruncherOperations
{
  double square_root (double operand);

  void square_root2 (double operand, org.omg.CORBA.DoubleHolder result);

  void square_root3 (org.omg.CORBA.DoubleHolder op_res);
}
```

Quindi il parametro `result` dovrà essere un oggetto istanza della classe DoubleHolder. Al server verrà passato un riferimento all'oggetto. Il server mediante questo riferimento modificherà l'attributo `value` che potrà essere letto dal client al termine dell'esecuzione della procedura.

Dichiarazione di possibili eccezioni

Una clausola `raises` indica che può essere sollevata una eccezione:

```
exception Failed {};

exception RangeError {
   unsigned long min_val;
   unsigned long max_val;
};

interface Unreliable {
  void can_fail() raises(Failed);
  void can_also_fail(in long l) raises(Failed, RangeError);
};
```

- Le eccezioni sono come strutture, ma possono anche non avere membri.
- Le eccezioni non possono essere annidate o ereditate.
- Le eccezioni non possono essere membri di altre strutture dati.

Operazioni oneway

L'IDL permette alle operazioni di essere dichiarate `oneway`:

```
interface Events {
  oneway void send(in EventData data);
};
```

Le seguenti regole si applicano alle operazioni `oneway`:

- Il tipo di ritorno deve essere `void`.
- Non devono avere parametri `out` o `inout`.
- Non devono avere una clausola `raises`.

Le operazione oneway hanno una semantica "best effort" send-and-forget, sono di tipo *may-be*.

Clausola context

Le operazioni possono avere una clausola `context`. Per esempio:

```
interface Poor {
  void doit() context("USER", "GROUP", "X*");
};
```

In questo caso il client manda tutti i valori delle variabili USER e GROUP del contesto CORBA, nonché tutti i valori delle variabili che iniziano per X.

I contesti sono concettualmente simili alle variabili UNIX d'ambiente.

Da evitare! Introducono falle nel controllo dei tipi e molti ORB non lo supportano.

Value Types

Ci sono molte occasioni in cui è desiderabile passare un oggetto per valore piuttosto che per riferimento.

Questo è utile quando il fine principale di un oggetto è passare dati incapsulati, o quando si vuole fare una copia di un oggetto.

La semantica CORBA per passare un oggetto per valore è simile a quella di liguaggi standard: il ricevente riceve una descrizione dello stato dell'oggetto, ne fa una nuova istanza con identità separata dal primo ma stesso stato.

I Value types forniscono una semantica tra le structs e le interfacce di CORBA.

Value Types

Dalle *specifiche di CORBA v. 3.3, Part 1* sui Vale Types:

- They support description of complex state (i.e., arbitrary graphs, with recursion and cycles).
- Their instances are always local to the context in which they are used (because they are always copied when passed as a parameter to a remote call.
- They support both public and private (to the implementation) data members.
- They can be used to specify the state of an object implementation.
- They support single inheritance (of valuetype) and can support an interface.
- They may be also be abstract.

Value Types

Esempio di uso di valuetype:

```
valuetype Date {

  private short  year;     //Il compilarore IDL genera un attributo non
                           //una coppia di metodi
  private short  month;

  private short  day ;

  void    next_day();

  void    previous_day();

};
```

Mapping tra tipi IDL e Java

IDL	Java
float	float
double	double
long, unsigned long	int
long long, unsigned long long	long
short, unsigned short	short
char, wchar	char
boolean	boolean
octet	byte
string, wstring	java.lang.String
any	class Org.omg.CORBA.Any.java
enum, struct, union	class

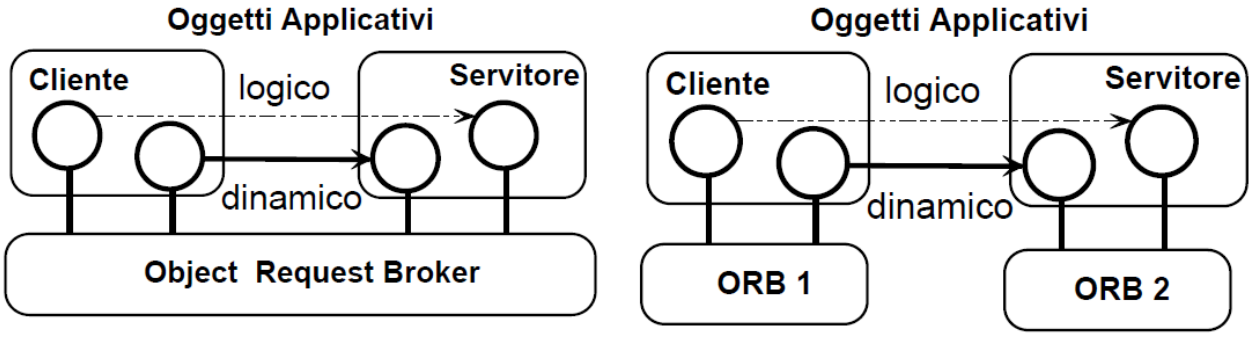
Esercizi su IDL

- Compilare `Hello.idl`
- Il compilatore genera in identificativo univoco per ogni interfaccia. Qual è l'identificativo dell'interfaccia `Hello`?
- Compilare un modulo più complesso come `CourseRegistration.idl`
- Analizzare i file generati per entrambi gli esempi
- Compilare `ValueTypes.idl` e `Union.idl` e analizzare i file generati.

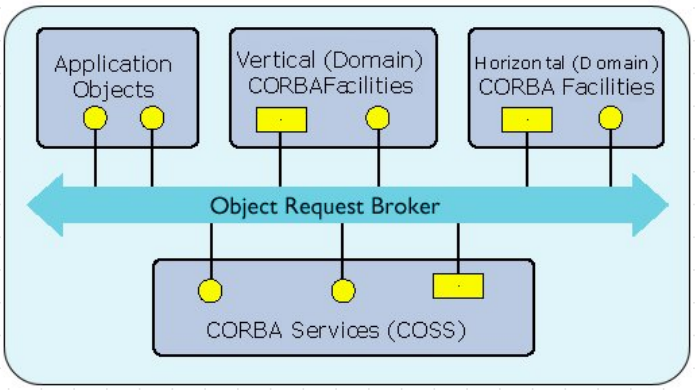
Il Broker

L'Object Request Broker (ORB) che consente i collegamenti in modo statico e dinamico tra le entità

- gestione dei nomi
- controllo allocazione e visibilità di oggetti
- controllo dei metodi e della comunicazione
- controllo di servizi accessori disponibili automaticamente
- gestione facilitata dei servizi



L'ORB come bus



CORBA services: forniscono funzionalità di base come system call di S.O.. Es.: Naming service

Horizontal CORBA facilities: funzionalità potenzialmente utili per ogni tipo di applicazione. Es.: Printing, Time, Internationalization

Domain CORBA facilities: forniscono interfacce per oggetti standard definite tramite IDL condivisi da compagnie o industrie

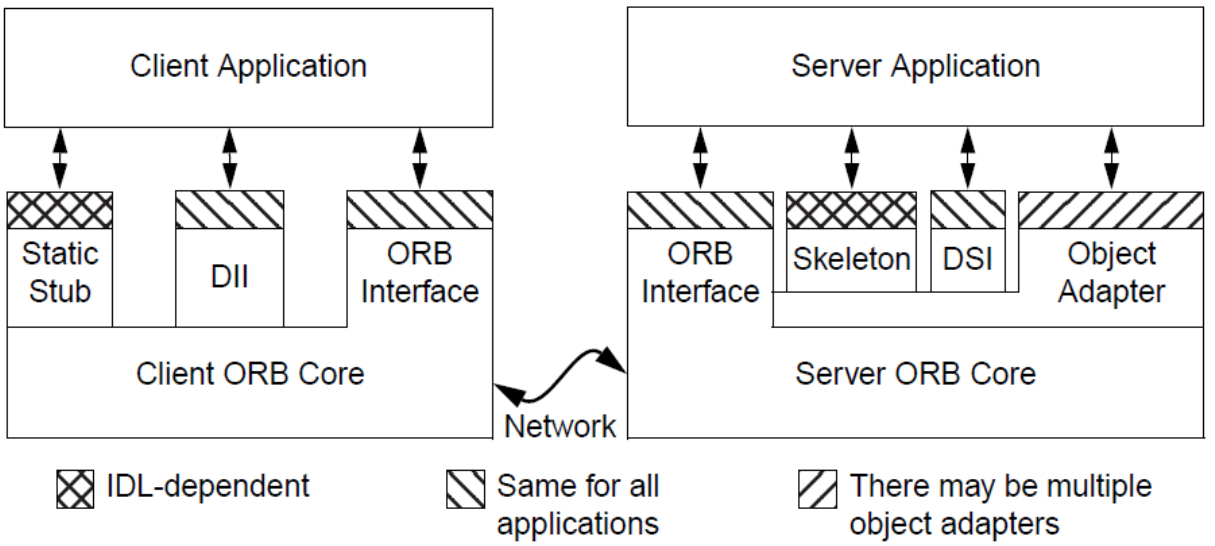
Application Objects: oggetti non standardizzati

ORB core

L'Object Request Broker (ORB) deve coordinare l'invocazione di servizi locali e remoti. Deve:

- individuare l'implementazione di un oggetto come servitore ad una richiesta (object location)
- preparare il servitore a ricevere la richiesta (object creation, activation & management)
- trasferire la richiesta dal cliente al servitore
- restituire la risposta al cliente

Componenti dell'ORB



Creazione di un oggetto ORB

Nel modulo CORBA è dichiarata un'operazione per la creazione di un ORB:

```
module CORBA {
  typedef sequence <string> arg_list;
  ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

L'operazione ORB_init serve per creare un ORB il cui identificatore (la stringa orb_identifier) è specificato come parametro. Non ci sono particolari specifiche sulla strutture della stringa, che può essere nulla visto che generalmente una applicazione istanzia un solo ORB.

La lista dei parametri arg_list, che può essere nulla anch'essa, serve per passare dei valori di inizializzazione dell'ORB. Ogni parametro deve avere la forma:
-ORB<suffix><optional white space><value>

Interfaccia dell'ORB

Nel seguito si riporta un estratto dell'interfaccia *ORB*. Sono omesse alcune parti relative ad argomenti non trattati, come le invocazioni dinamiche.

```
module CORBA {
 typedef string ORBid;
 interface ORB {
   typedef string ObjectId;
   exception InvalidName {};
   ORBid id();
   string object_to_string ( in Object obj );
   Object string_to_object ( in string str );
   // Dynamic Invocation related operations
   ....
```

Operazioni dell'ORB

```
ORBid id();
```

Restituisce la stringa identificatrice dell'ORB passata come parametro ad `ORB_init`.

```
string object_to_string ( in Object obj );
```

```
Object string_to_object ( in string str );
```

I riferimenti ad oggetti sono opachi e possono differire da ORB ad ORB. Per favorire la loro memorizzazione e comunicazione il riferimento può essere trasformato in stringa dall'operazione `object_to_string` e di nuovo in riferimento da `string_to_object`.

Il richiamo di `string_to_object(object_to_string(obj))` deve restituire un riferimento allo stesso oggetto `obj`.

Interfaccia dell'ORB

Operazioni per la risoluzione di riferimenti iniziali e per condividere il main thread con l'ORB.

```
module CORBA {
 ...

 interface ORB {
   ....

   // Initial reference operation

   Object resolve_initial_references ( in ObjectId identifier)
      raises (InvalidName);

   ....

   // Thread related operations

   boolean work_pending( );

   void perform_work();

   ....
```

Operazioni dell'ORB

```
Object resolve_initial_references ( in ObjectId identifier)
raises (InvalidName);
```

Restituice l'oggetto identificato dal parametro `identifier`. Alcuni valori riservati sono `NameService`, `RootPOA`, `POACurrent`, `ORBPolicyManager`.

```
boolean work_pending();
```

Se la `work_pending()` restituisce `TRUE` vuol dire che l'ORB ha del lavoro da fare che richiede il thread principale.

```
void perform_work();
```

Se chiamata nel thread principale, la `perform_work()` permette all'ORB di svolgere un'unità di lavoro (definita dall'implementazione), altrimenti non fa nulla.

Uso di work_pending() e perform_work()

Le operazioni `work_pending()` e `perform_work()` possono essere usate per scrivere un semplice ciclo di polling che divide il thread principale tra l'ORB e altre attività. Un tale ciclo è utile per i server con thread singolo.

```
// C++
   for (;;) {

       if (orb->work_pending()) {

          orb->perform_work();
       };

   // do other things
   // or sleep
   };
```

Interfaccia dell'ORB

Operazioni per l'avvio e la chiusura dell'ORB.

```
module CORBA {
 ...

 interface ORB {
   ....

   void run();

   void shutdown(
     in boolean wait_for_completion
   );

   void destroy();

   ...

   };
 };
```


Operazioni dell'ORB

`void run();`

`void shutdown( in boolean wait_for_completion );`

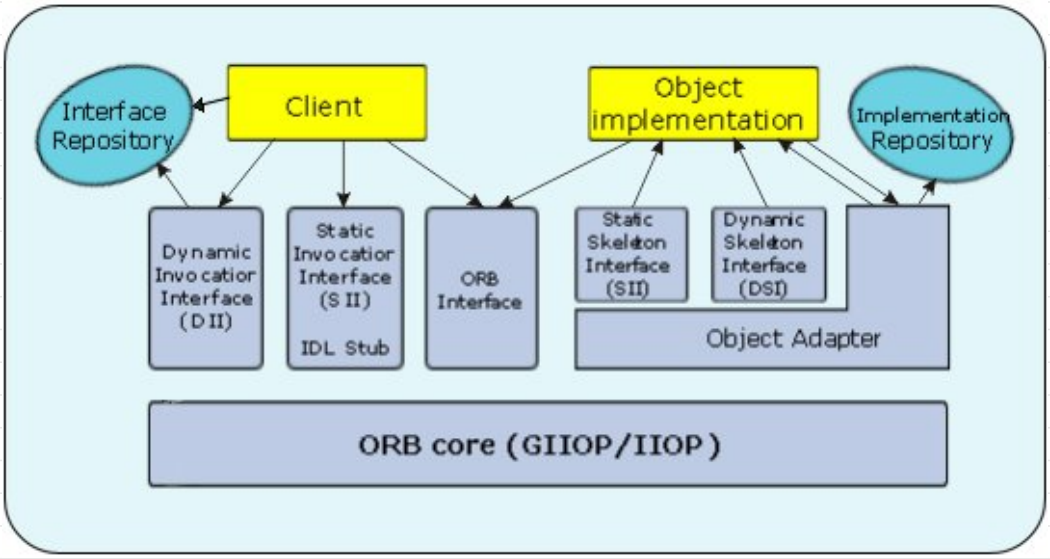
L'operazione `run` permette all'ORB di eseguire le sue funzioni interne. Un'applicazione dovrebbe richiamare o `run` oppure `perform_work` nel thread principale.

L'operazione `run` è bloccante e termina quando un qualche thread richiama `shutdown`, che stoppa l'attività dell'ORB. `shutdown` può uscire senza che le attività di shutdown siano completate se il parametro è `FALSE`.

`void destroy();`

Questa operazione distrugge l'oggetto ORB. Una volta che l'ORB è distrutto, un'altra chiamata ad `ORB_init` restituirà un riferimento ad un nuovo ORB. Se `destroy` è chiamato senza aver fatto lo shutdown, questa operazione richiama `shutdown(TRUE)`.

Object Adapter



Un Object Adapter (OA) è parte de CORBA runtime system che "adapt" il concetto di CORBA object alla realtà dei linguaggi di programmazione dei server.

Object Adapter

In pratica un OA tratta problemi come:

- Leggere e deserializzare (unmarshaling) i parametri delle richieste ed invocare la relativa operazione sul servant.
- Fornire le API che permettono ad un object reference di essere di essere mappato sul corrispondente servant e viceversa.
- Creare al volo dei servants e salvare lo stato di unservant su file o database.

Le prime versioni di CORBA specificavano un Basic Object Adapter (BOA) molto semplice (sotto-specificato).
I fornitori di CORBA dovettero aggiungere miglioramenti e il risultato fu che ogni fornitore aveva la propria versione di BOA, e questo diminuiva la portabilità delle applicazioni tra differenti prodotti CORBA.

L'OMG rimpiazzò quindi il BOA con il Portable Object Adapter (POA).

Portable Object Adapter: Specifiche

(Direttamente dalle specifiche di CORBA 3.3)
The POA is designed to meet the following goals:

- Allow programmers to construct object implementations that are portable between different ORB products.
- Provide support for objects with persistent identities. More precisely, the POA is designed to allow programmers to build object implementations that can provide consistent service for objects whose lifetimes (from the perspective of a client holding a reference for such an object) span multiple server lifetimes.
- Provide support for transparent activation of objects.
- Allow a single servant to support multiple object identities simultaneously.
- Allow multiple distinct instances of the POA to exist in a server.

(Segue....)

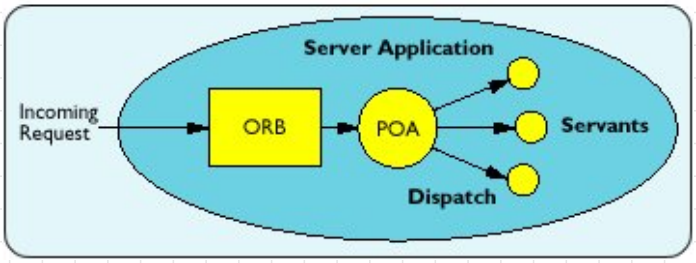
Portable Object Adapter: Specifiche

...

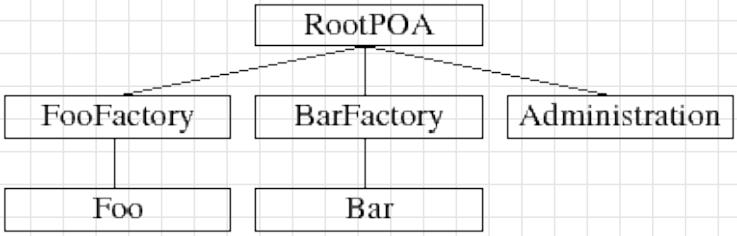
- Provide support for transient objects with minimal programming effort and overhead.
- Provide support for implicit activation of servants with POA-allocated Object Ids.
- Allow object implementations to be maximally responsible for an object's behavior.
- Avoid requiring the ORB to maintain persistent state describing individual objects, their identities, where their state is stored, whether certain identity values have been previously used or not, whether an object has ceased to exist or not, and so on.
- Provide an extensible mechanism for associating policy information with objects implemented in the POA.
- Allow programmers to construct object implementations that inherit from static skeleton classes, generated by IDL compilers, or a DSI implementations

Root POA e gerarchie di POA

Un POA può essere visto come un insieme di *servants*.

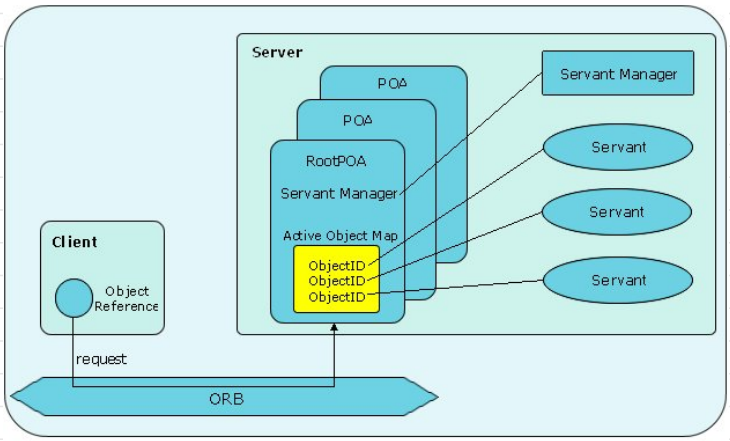


All'interno di una applicazione server possono essere attivi più POA.
I POA sono organizzati in modo gerarchico e all'apice della gerarchia c'è il RootPOA.



Molteplicità dei POA

Un server può avere più di un POA perché così può associare differenti politiche a ciascun gruppo di servants associato ad un singolo POA.
Per esempio, se alcuni servants sono implementati in thread-safe, possono essere raggruppati in un multi-threaded POA. Al contrario se alcuni servants non sono thread-safe si possono raggrupare in un single-threaded POA.



Tipi di POA

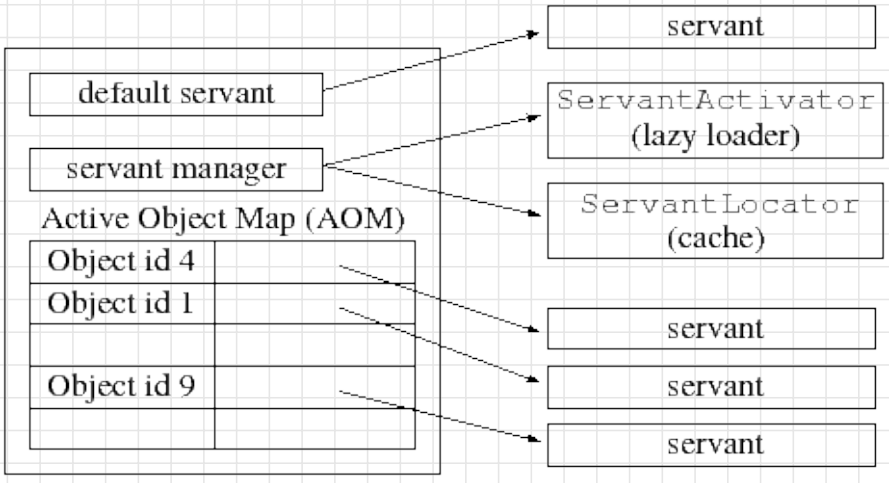
Esistono differenti tipi di POA per poter rispondere adeguatamente a situazioni diverse.
Per esempio:

- Un servant deve essere creato prima dell'arrivo della richiesta? O può essere creato al bisogno?
- Una volta che il servant è stato creato può essere distrutto e rimpiazzato da un altro servant? Se sì, si può pensare ad una cache di servants che contiene i servants per gli oggetti CORBA usati più recentemente.
- C'è una relazione uno-ad-uno tra servants e oggetti CORBA che essi rappresentano? Oppure un servant può essere utilizzato per più oggetti CORBA?

Per rispondere alle diverse esigenze dei programmatori, CORBA definiscce varie soluzioni e tecniche per migliorare le performace dei server e la loro scalabilità.

POA Generico

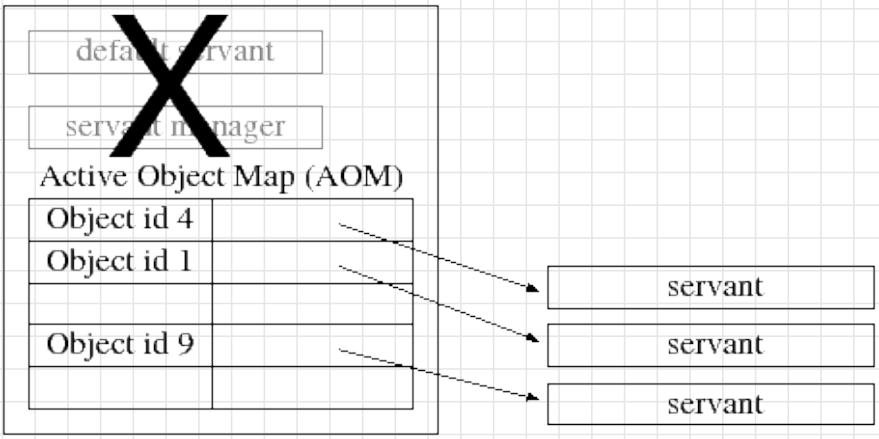
Al fine di soddisfare tutte le possibilità, il POA deve essere un tipo di contenitore flessibile.
La struttura logica del POA:



Un POApuò avere: un default servant, uno di due tipi di servant manager e una Active Object Map (AOM).

Un semplice POA

Il caso più semplice e frequente di POA:



Uno IOR contiene i dettagli per riferirsi ad un oggetto CORBA, come l'host e la porta del processo server e anche l'Object Key, che identifica il servant.
L'Object key varia da un prodotto CORBA all'altro ma in genere contiene:

- il nome gerarchico del POA che contiene il servant per l'oggetto CORBA.
- un Object Id che univocamente identifica un servant nel POA.

Un semplice POA

Quando una richiesta arriva al server, l'header della richiesta contine l'object key.
Il CORBA runtime system nel server estrae il nome del POA e l'Object Id dall'object key.
Il CORBA runtime system usa l'object id per eseguire un lookup nella AOM del POA specificato.
Se esiste un servant associato all'object id allora la richiesta è inviata al servant. Altrimenti un'eccezione OBJECT_NOT_EXIST è inviata al client.
Si può utilizzare questo tipo di POA se:

- C'è abbastanza memoria per mantenere tutti i servants.
- Si possono creare i servants prima di aver ricevuta la prima richiesta per il relativo oggetto CORBA.

Un semplice POA: attività del server

Un server deve inizializzare l'ORB a run time prima che possa accettere le richieste. Per questo i seguenti passi sono necessari:

1. Chiamare un metodo come `ORB.init()` per inizializzare l'ORB.
2. Ottenere un riferimento al Root POA.
3. Istanziare uno o più servants.
4. Attivare ogni servant.
5. Rendere i riferimenti agli oggetti disponibili per i clients.
6. Attivare il Root POA Manager (vedremo).
7. Iniziare un dispatch loop.

Un semplice POA: esempio di codice JAVA lato server

```
...
ORB orb = ORB.init(args, null);  // create and initialize the ORB

// get reference to rootpoa and activate the POAManager
POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

ServantImpl servantImpl = new ServantImpl();  // create servant

// store the servant in the AOM and get its object reference
org.omg.CORBA.Object ref = rootpoa.servant_to_reference(servantImpl);
Servant href = ServantHelper.narrow(ref);

rootpoa.the_POAManager().activate(); // POA Manager activation

orb.run();  //  dispatch loop
...
```

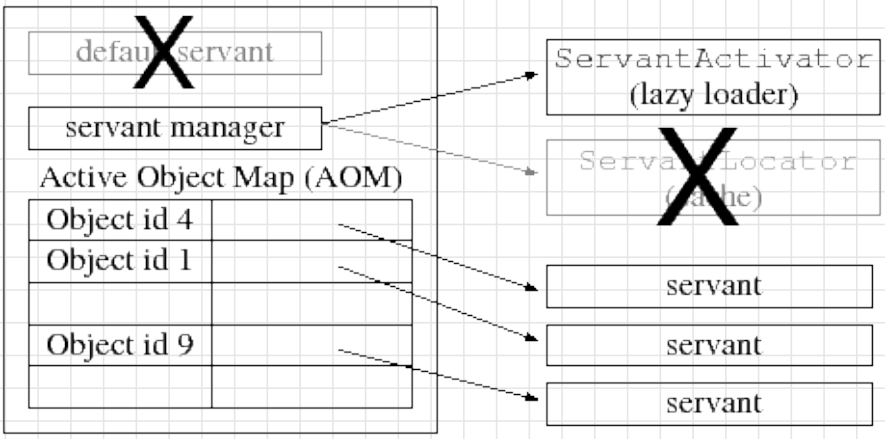
Un semplice POA: esempio di codice C++ lato server

```
#include <iostream.h>
#include <OB/CORBA.h>
#include "Servant_skel.h"

// Servant class definition here...

int
main(int argc, char * argv[])
{
   // Initialize ORB

   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);


   // Get reference to Root POA

   CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
   PortableServer::POA_var poa = PortableServer::POA::_narrow(obj);


   // Create a servant

   Servant_impl servant = new Servant_impl();

   ...
```


Un semplice POA: esempio di codice C++ lato server

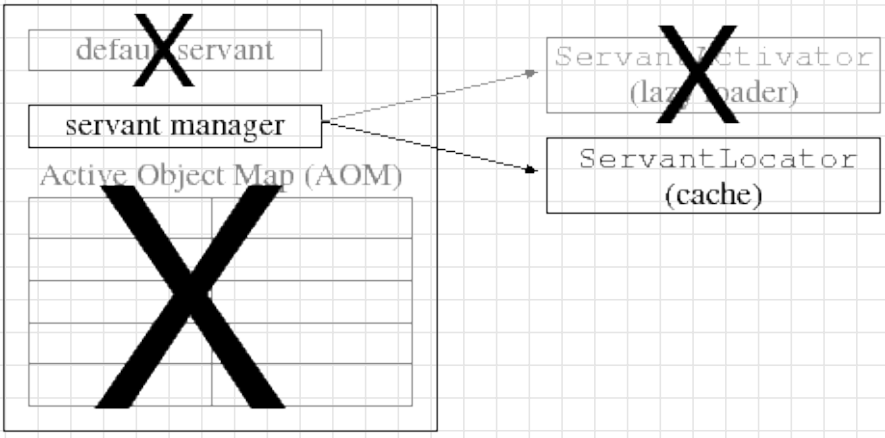
```
  // Get a CORBA reference with the POA through the servant

   CORBA::Object_var sv = myPOA->servant_to_reference(servant);


   // Write its stringified reference to stdout

   CORBA::String_var str = orb->object_to_string(sv);
   cout << str << endl;


   // Activate POA manager

   PortableServer::POAManager_var mgr = poa->the_POAManager();
   mgr->activate();


   // Accept requests
   orb->run();

}
```

POA Loader



Quando una richiesta arriva al server, se il CORBA runtime system non trova l'object nella AOM dello specificato POA, chiede al POA di invocare l'operazione `incarnate()` del servant manager per ricreare il servant desiderato. Il POA aggiunge il servant all'AOM. Se il servant manger non può ricreare il servant allora genera un'eccezione OBJECT_NOT_EXIST. Questo tipo di POA si usa quando si ha abbastanza memoria per tutti i servant, ma non si vuole utilizzarla se non arrivano richieste.

POA Cache



Quando una richiesta arriva al server il CORBA runtime system passa il controllo al POA specificato nell'object key. Il POA poi esegue il seguente (pseudocode) algoritmo per inviare la richiesta:

```
sv = cache.preinvoke(object_id,...);
sv.operation(...);
cache.postinvoke(..., sv, ...);
```

POA Cache

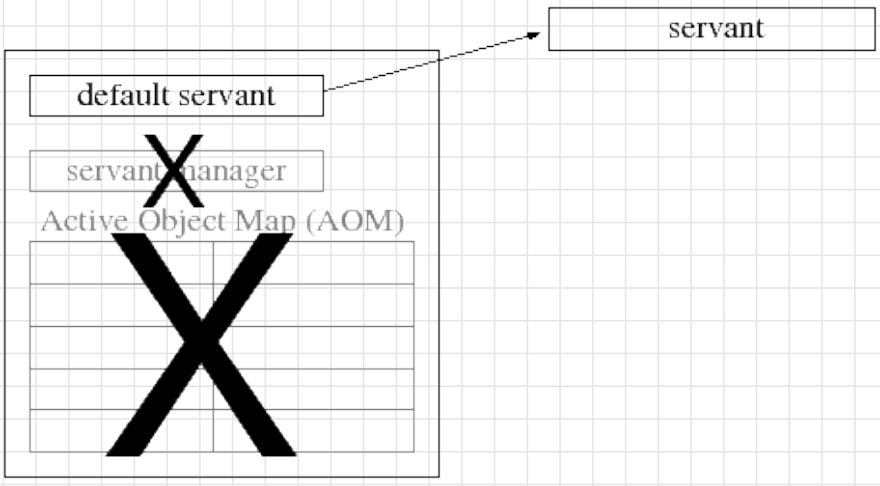
Il POA non mantiene una AOM. Questo perché assume che il servant manager implementi la sua AOM-like struttura dati.

Si usa un POA Cache quando si ha abbastanza memoria per mantenere *alcuni ma non tutti* i servants in memoria allo stesso tempo.

Nota: è compito del programmatore implementare le interfacce `ServantActivator`, nel caso del POA Loader, e `ServantLocator`, nel caso del POA Cache.

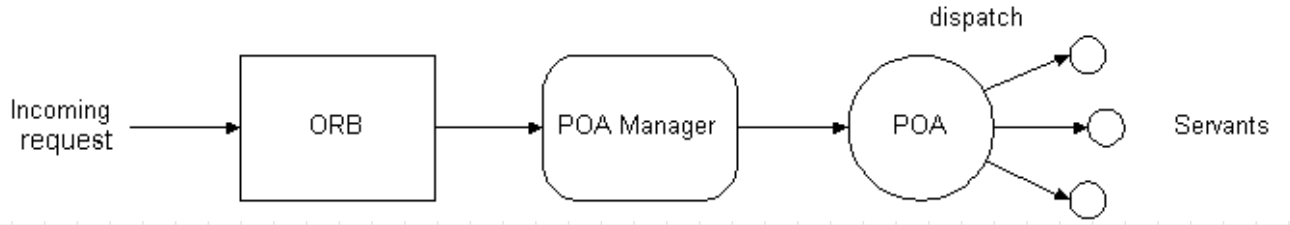
Gli oggetti che implementano queste interfacce saranno passate al POA una volta creato.

POA Default Servant



Il POA ha un unico *default servant*. Una richiesta per un qualsiasi CORBA object, è inoltrata dal POA al servant, che chiama la `get_object_id()` del POA per conoscere a quale oggetto CORBA si riferisce la richiesta.
L'object identifier è usato per accedere ad un database di servants.
Il modello POA "default servant" minimizza il consumo di memoria al prezzo di un overhead nei tempi di esecuzione.

POA Manager

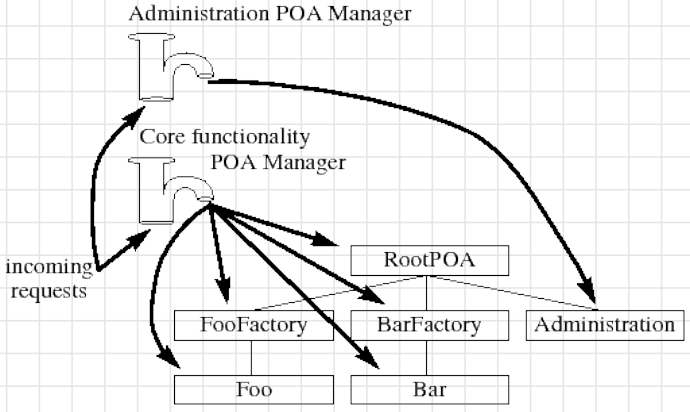


Associato ad ogni POA c'è un POA Manager che funziona come un rubinetto:
Concettualmente ha due stati: "on" permette il flusso delle richieste, e "off" lo nega.

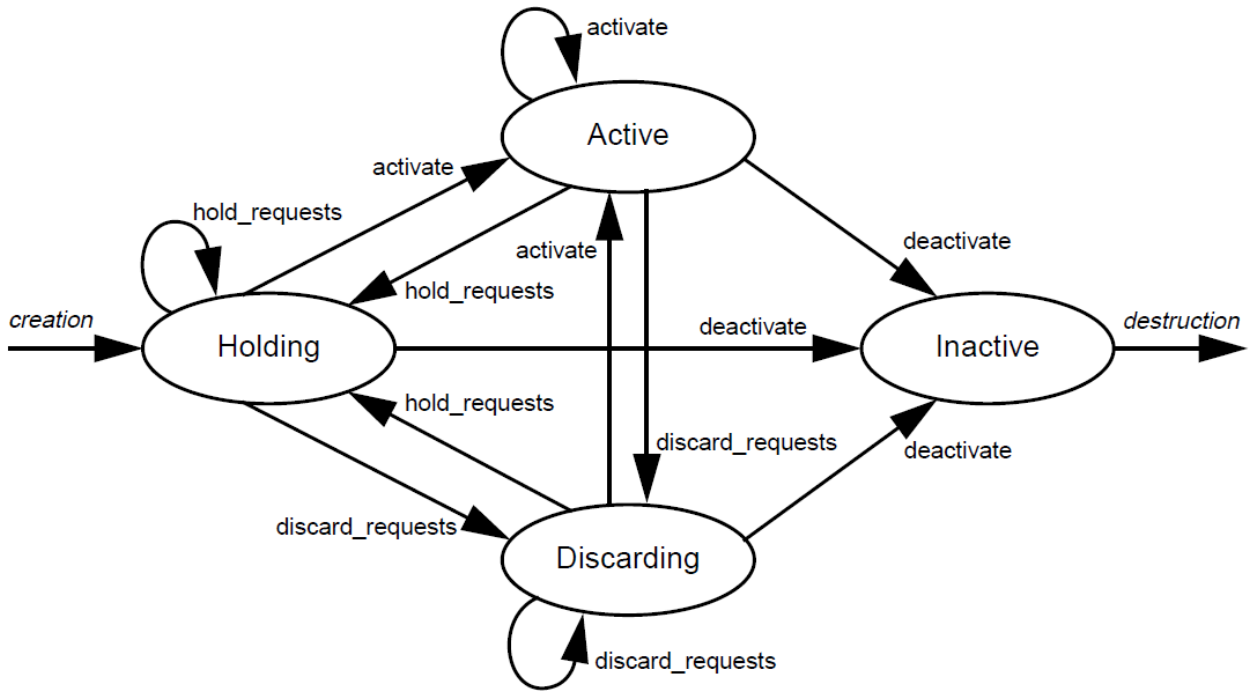
POA Manager

Un POA Manager può trovarsi in quattro stati:

- HOLDING: è uno stato "off". Le richieste sono accodate.
- DISCARDING: è uno stato "off". Le richieste scartate. Un'eccezione CORBA::TRANSIENT è lanciata al client.
- ACTIVE: è uno stato "on". Le richieste sono inviate ai servants.
- INACTIVE: è uno stato "off". Le richieste rigettate. Si entra in questo stato quando il server sta eseguendo lo shut down.



POA Manager: State transition



POA Policies

I POA hanno delle politiche associate.
Le politiche controllano le caratteristiche implementative dei riferimenti agli oggetti e dei servants.
Esse sono:

- LifespanPolicy
- IdAssignmentPolicy
- IdUniquenessPolicy
- ImplicitActivationPolicy
- RequestProcessingPolicy
- ServantRetentionPolicy
- ThreadPolicy

POA policy: l'iterfaccia CORBA

Il modulo CORBA fornisce una interfaccia per definire le politiche:

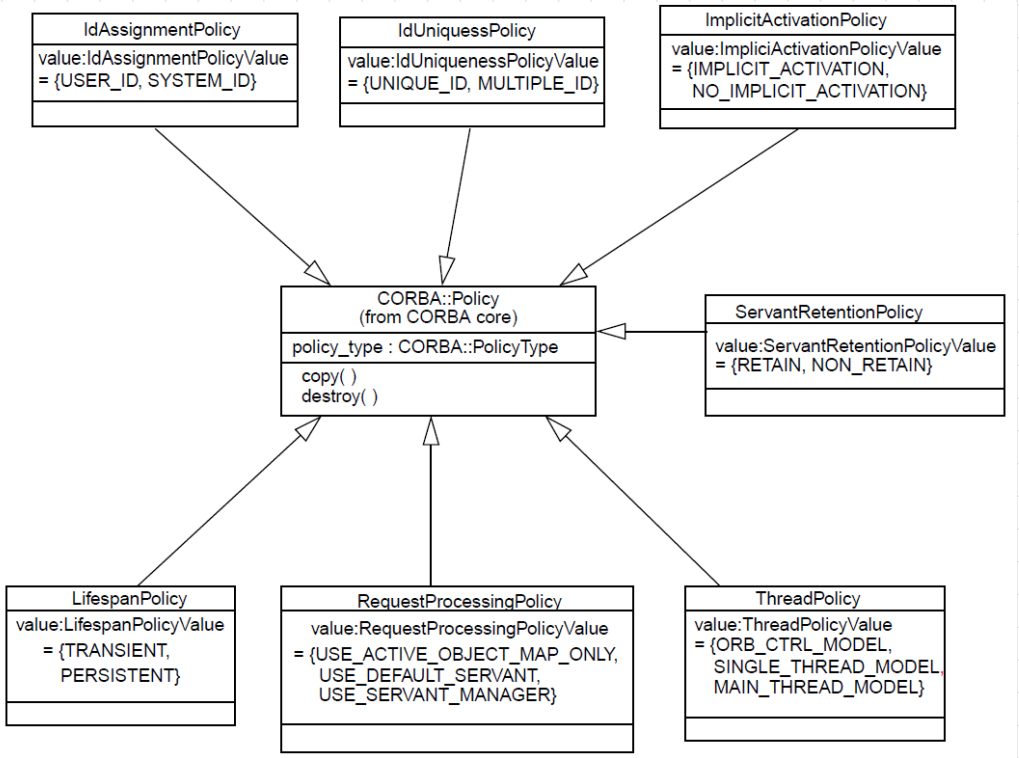
```
module CORBA {

   typedef unsigned long PolicyType;

    interface Policy {

          readonly attribute PolicyType policy_type;

          Policy copy();

          void destroy();

          };

   typedef sequence<Policy> PolicyList;

...
};
```


Creazione di un POA con politiche

Per associare delle politiche ad un gruppo di servants bisogna creare un POA specifico secondo la seguente interfaccia CORBA:

```
module PortableServer {

      interface POAManager;

      exception AdapterAlreadyExists {};
      exception InvalidPolicy { unsigned short index; };

      interface POA {

          POA create_POA(
                          in string adapter_name,
                          in POAManager manager,
                          in CORBA::PolicyList policies;)
                raises(AdapterAlreadyExists, InvalidPolicy);

      readonly attribute POAManager the_POAManager;
      ...
      };

 ...
};
```

Valori per le politiche POA



Valori per le politiche del POA

Life Span Policy: `TRANSIENT` (il servant termina la sua vita con il server) / `PERSISTENT` (il servant persiste oltre il server)

ID Assignment Policy: `SYSTEM_ID` (l'Object Id è assegnato dal sistema) / `USER_ID` (l'Object Id è assegnato dal programmatore)

ID Uniqueness Policy: `UNIQUE_ID` (nell'AOM ogni servant ha un unico Object Id associato) / `MULTIPLE_ID` (più Object Id sono associati ad un unico servant)

Servant Retention Policy: `RETAIN` (Il POA ha l'AOM) / `NON_RETAIN`

Request Processing Policy: `USE_ACTIVE_OBJECT_MAP_ONLY` / `USE_DEFAULT_SERVANT` / `USE_SERVANT_MANAGER`

Implicit Activation Policy: `IMPLICIT_ACTIVATION` (Servants aggiunti a AOM richiedendo l'Obj. Id) / `NO_IMPLICIT_ACTIVATION`

Thread Policy: `ORB_CTRL_MODEL` (L'ORB può usare più threads nell'invio delle richieste) / `SINGLE_THREAD_MODEL` (Le richieste per il POA sono serializzate) / `MAIN_THREAD_MODEL` (Le richieste per tutti i POA con questa politica sono serializzate)

Esempio per politiche POA

```
...
 POAManager poa_manager = root_poa.the_POAManager();

 POA myPOA = null;

 Policy[] myPolicies = new Policy[2];

 myPolicies[0] =
  root_poa.create_id_assignment_policy(IdAssignmentPolicyValue.USER_ID);

 myPolicies[1] =
  root_poa.create_lifespan_policy(LifespanPolicyValue.PERSISTENT);

 String myPoasName = "myPOA";

 myPOA = root_poa.create_POA(myPoasName, poa_manager, myPolicies);

....
```


Politiche del Root POA

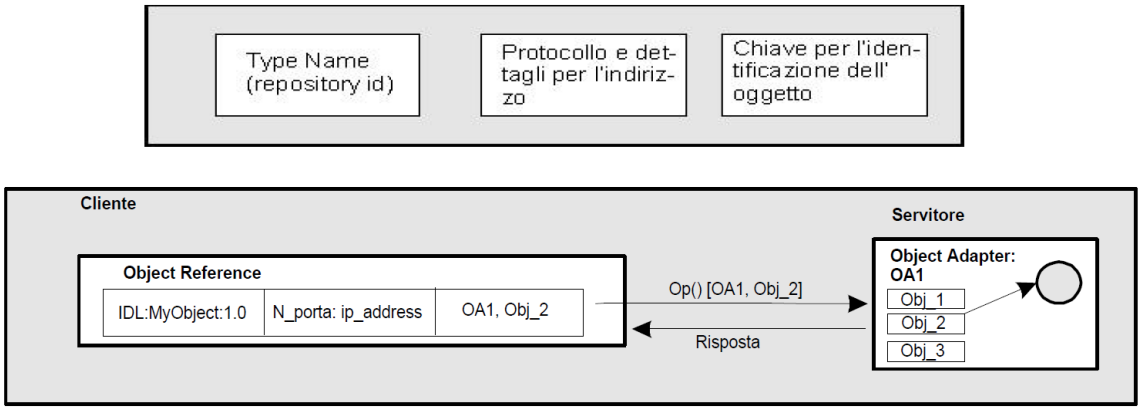
Le politiche del Root POA sono fisse e hanno i seguenti valori:

- Life Span Policy: `TRANSIENT`
- ID Assignment Policy: `SYSTEM_ID`
- ID Uniqueness Policy: `UNIQUE_ID`
- Servant Retention Policy: `RETAIN`
- Request Processing Policy: `USE_ACTIVE_OBJECT_MAP_ONLY`
- Implicit Activation Policy: `IMPLICIT_ACTIVATION`
- Thread Policy: `ORB_CTRL_MODEL`

Il Root POA è utile solo per oggetti non persistenti.
Se si vogliono creare oggetti persistenti o usare altre politiche si devono creare nuovi POA.

CORBA interoperability: IOR

Le implementazioni dell'ORB sono differenti, ma è necessario che gli oggetti CORBA vengano identificati in modo univoco all'interno di ambiente CORBA.
Un server può creare un oggetto CORBA ed ottenere dall'ORB un Iteroperable Object Reference (IOR) che lo identifica univocamente.
Uno IOR è un riferimento ad un'oggetto CORBA che può essere utilizzato dai client per accedere alle operazioni di quell'oggetto.



IOR

Uno IOR non è in un formato leggibile, ma può essere trasformato in una stringa e viceversa in un riferimento ad un oggetto.
Esempio:

IOR:000000000000001749444c3a48656c6c6f4170702f48656c6c6f3a312e3000000000000100000000000000820001020000000000a3132372e302e312e310095d400000031afabcb0000000020e7217df3000000010000000000000100000008526f6f74504f41000000000800000001000000001400000000000002000000010000002000000000000100010000000205010001000100200001010900000001000101000000026000000020002

IOR

Un server può stampare uno IOR "stringified" e un client può utilizzarlo per l'accesso all'oggetto relativo.
Le operazioni sono `orb.object_to_string( ... )` e `orb.string_to_object( ... )`.
Esempio di pubblicazione di uno IOR da parte di un server:

```
...

 // creazione di un servant
 HelloImpl helloImpl = new HelloImpl();

 //  richiesta dell'object reference relativo al servant
 org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
 Hello href = HelloHelper.narrow(ref);

 // pubblicazione di uno IOR
 System.out.println(orb.object_to_string(href));

...
```


Struttura di un Interoperable Object Identifier (IOR)

Uno IOR è concepito come un biglietto da visita:

Nel biglietto c'è l'indicazione della persona a cui si riferisce e una serie di contatti (indirizzi) per poterla reperire.

Struttura di un Interoperable Object Identifier (IOR)

Formalmente uno IOR è una istanza della seguente interfaccia:

```
module IOP {

    typedef unsigned long ProfileId;

    const ProfileId TAG_INTERNET_IOP = 0;

    struct TaggedProfile {
        ProfileId        tag;
        sequence<octet>  profile_data;
    };

    struct IOR {
        string                    type_id;
        sequence<TaggedProfile>   profiles;
    };
    ...
```

I riferimenti all'oggetto sono dati dalla sequenza di "Tagged Profile": i possibili contatti.

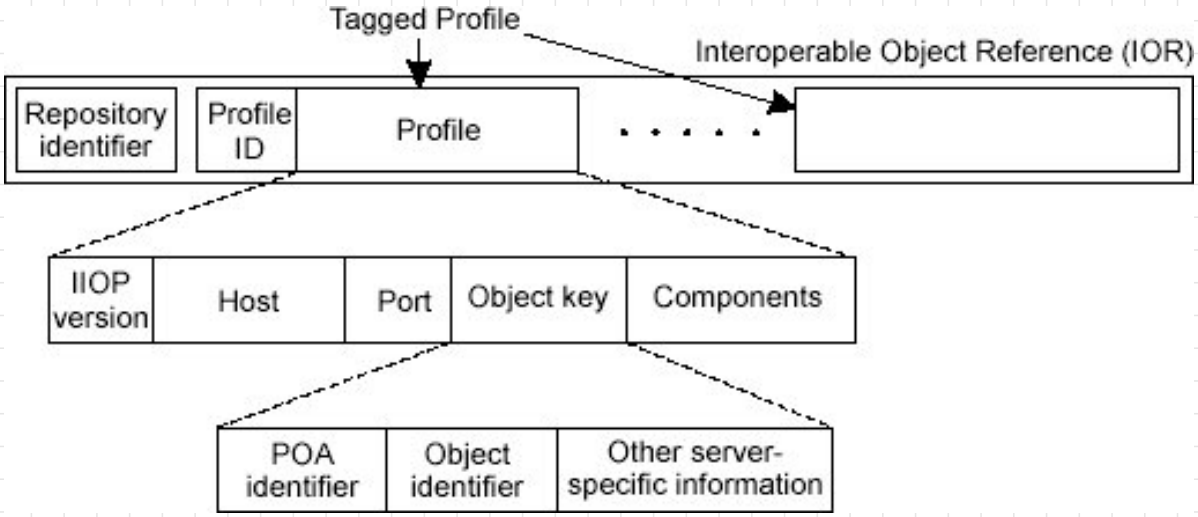
Struttura di un Interoperable Object Identifier (IOR)

Formalmente uno IOR è una istanza della seguente interfaccia:

```
module IOP {

   ...

    struct IOR {
        string                    type_id;
        sequence<TaggedProfile>   profiles;
    };

// Standard way of representing multicomponent profiles.
// This would be encapsulated in a TaggedProfile.

    typedef unsigned long ComponentId;

    struct TaggedComponent {
        ComponentId tag;
        sequence <octet> component_data;
    };

    typedef sequence<TaggedComponent> TaggedComponentSeq;

};
```

Struttura di un Interoperable Object Identifier (IOR)

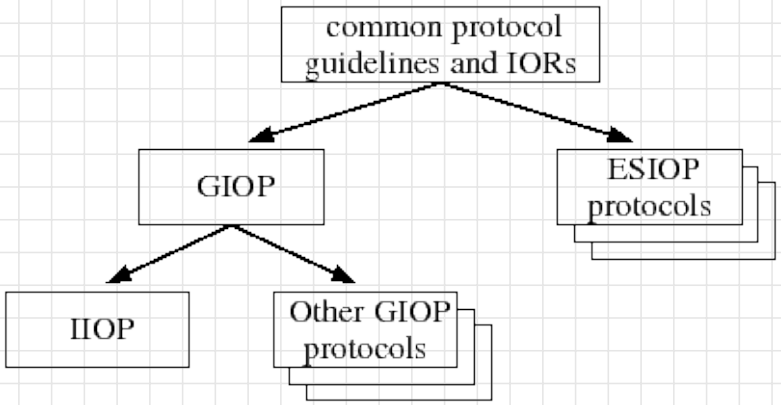
Nel caso un tagged profile fosse un contatto attraverso un protocollo su TCP/IP ed in particolare il protocollo IIOP, la struttura di uno IOR è concettualmente come segue:



I "Components" sono informazioni aggiuntive come host e port alternativi, codifica dei caratteri (char/wchar), ...

Protocolli CORBA

L'OMG fornisce un meccanismo di comunicazione tra applicazioni mediante uno stack di protocolli.



Tutti i protocolli sono definiti seguendo delle linee guida comuni (che includono ad esempio la definizione dello IOR).

L'*Environment-Specific Inter-ORB Protocol (ESIOP)* è un portocollo ottimizzato per ambienti specifici o per compatibilità verso vecchi protocolli (es. DCE RPC).

GIOP/IIOP

Il *General Inter-ORB Protocol (GIOP)* specifica una sintassi di rappresentazione dei dati a basso livello e un insieme di formati per messaggi tra ORBs.

Il GIOP è pensato per specificamente per le interazioni ORB-to-ORB

Il GIOP è progettato per lavorare su un qualsiasi protocollo di trasporto connection oriented che rispetti un insieme minimo di assunzioni.

L' *Internet Inter-ORB Protocol (IIOP)* specifica come i messaggi GIOP sono scambiati utilizzando connessioni TCP/IP.

GIOP e IIOP sono obbligatori per ogni implementazione di CORBA.

Common Data Representation (CDR)

Come visto, l'atto di inserire dati di un certo tipo in un buffer per la trasmissione è chiamato marshalling, mentre l'atto di estrarre dati di un certo tipo da un buffer è chiamato unmarshalling.

Le regole per il marshalling sono chiamate Common Data Representation (CDR).

Una regola è che i dati sono codificati usando il formato big-endian o little-endian nativo del computer che spedisce. Un flag nell'header dei messaggi GIOP specifca quale dei due formati è usato.

Il ricevente effettuarà la trasformazione dei dati se il suo formato di rappresentazione è diverso da quello del trasmettitore.

Common Data Representation (CDR)

Le regole CDR per i tipi base stabiliscono quanti byte di memoria occupano ciascuno e il loro allineamento in memoria:

Tipo	lung.	allineam.
`char, octet, boolean`	1 byte	
`short, unsigned short`	2 bytes	2 byte
`long, unsigned long, float`	4 bytes	4 bytes
`long long, unsigned long long, double`	8 bytes	8 bytes
`long double`	16 bytes	8 bytes

Common Data Representation (CDR)

Marshalling di alcuni altri tipi:

`enum` : `unsigned long`.

`string` : lunghezza della stringa (`unsigned long`) seguita dai caratteri della stringa e dal carattere `null`.

`struct` : marshalling di ogni campo.

`exception` : repository id dell'eccezione, che è una stringa, seguita dal marshalling di ogni campo.

`union` : discriminante seguita dal marshalling del campo attivo.

`sequence` : numero degli elementi che contiene (`unsigned long`) seguito da ogni elemento.

`array` : marshalling di ogni elemento nell'array.

`object reference` : marshalling dello IOR.

GIOP: tipi di messaggi

GIOP definisce otto differenti tipi di messaggi.

I tipi di messaggio GIOP sono: *Request, Reply, Fragment, CancelRequest, CloseConnection, MessageError, LocateRequest, LocateReply*.

Request, Reply : usati per inviare richieste da client ad server o per ricevere risposte da server a client.

LocateRequest, LocateReply : come un messaggio "ping" per chiedere: "l'oggetto è lì?". La risposta è mandata per mezzo di un messaggio *LocateReply*.

Fragment : Se un messaggio è molto lungo, l'ORB può decidere di inviarlo a pezzi. In questo caso ogni pezzo è inviato come un normale messaggio *Request, Reply*, ma con un flag nell'header che indica la presenza di altri pezzi che seguono.

GIOP: tipi di messaggi

CancelRequest : Se il cliente non riceve una risposta entro un timeout, può mandare un messaggio *CancelRequest* al server per indicare che il client ignorerà il messaggio *Reply* di risposta.

CloseConnection : CORBA permette che connessioni inattive siano chiuse. Se un server manda un tale messaggio al client, immediatamente dopo chiude la connessione. Il client può comunque riaprirla in modo trasparente.

MessageError : se viene ricevuto un messaggio non in formato GIOP, un *MessageError* viene inviato al mittente.

GIOP redirection

Quando un client manda un messaggio *Request* al server, riceve un messaggio *Reply* che può essere un "normal reply", un "exception reply", oppure un "redirection reply", detto `LOCATION_FORWARD` in CORBA.

Questo messaggio dice al client che l'oggetto target non vive nel processo server, ma è altrove.

Il corpo di questo messaggio contiene uno IOR che ridireziona il client verso il server dove l'oggetto risiede.

Il CORBA runtime system del client rispedisce il messaggio in modo trasparente utilizzando il nuovo IOR.

La riderezione avviene solo la prima volta: ulteriori richieste andranno direttamente all'oggetto target.

Questo meccanismo viene utilizzato per realizzare l'Implementation Repository (IMR).

GIOP header

Header di un messaggio GIOP contenente un messaggio dei tipi descritti.

```
module GIOP { // IDL extended for version 1.1, 1.2, and 1.3

    struct Version {
      octet major;
      octet minor;
    };

...

typedef char Magicn[4];

// GIOP 1.3
    struct MessageHeader {
      Magicn magic;
      Version GIOP_version;
      octet flags;
      octet message_type;
      unsigned long message_size;
};
```

GIOP header

`magic` : identifica un messaggio GIOP. Il valore è sempre la stringa "GIOP".

`GIOP_version` : identifica la versione. Es. per la 1.3, major vale 1 e minor 3.

`flags` : il bit meno significativo indica la codifica: 0 big-endian, 1 little-endian. Il secondo meno significativo indica, se posto a 1, la presenza di ulteriori frammenti. Gli altri bit hanno valore 0.

`message_type` : indica il tipo di messaggio tra gli otto ammessi.

`message_size` : numero di ottetti del messaggio che seguono l'header.

CORBA services

CORBA fornisce tutte una serie di servizi. Tra questi:

- Naming Service
- Trading Service
- Event Service
- Log Service
- Time service

Naming service: Motivazioni

Un Server può stampare, (inviare, scrivere su file ...) uno "stringified" IOR e un client può utilizzarlo per l'accesso all'oggetto relativo mediante le operazioni `orb.object_to_string( ... )` (lato server) e `orb.string_to_object( ... )` (lato client).

Questo meccanismo funziona fintanto che un client e un server condividono un file system o un altro meccanismo per passare lo IOR di un oggetto.

È inverosimile che in una wide area network un file system sia condiviso.

Inoltre, copiare riferimenti stringified da un server a tutti i client è una soluzione goffa e non è scalabile.

Per questa ragione CORBA ha proposto diversi meccanismi con cui un server può notificare la presenza di un oggetto.

Naming Service: Motivazioni

Il Naming Service fornisce un modo ai Server per notificare i riferimenti ad oggetti tramite nomi, e per i Client un modo per richiamarli. I vantaggi sono:

- I Client e i Server possono usare nomi significativi invece di dover manipolare stringified IORs.
- Cambiando un riferimento ad un servizio senza cambiare il nome, si possono indirizzare i client in modo trasparente verso nuovi oggetti.
- Il Naming Service risolve il problema del bootstrap perché fornisce un punto fisso per i Client e i Server dove incontrarsi.

Il Naming Service è come il servizio telefonico di Pagine Bianche: dato un nome restituisce un riferimento.

Naming service: Meccanismo lato server

Le caratteristiche dell'interazione dei Server con un Naming Service sono le seguenti:

- Il Naming Server gira su una ben definita porta di un host di una rete di computer.
- I processi Server registrano nel Naming Server gli oggetti da loro gestiti a cui danno un nome.
- Il Naming Server memorizza, per ogni oggetto, il nome e lo IOR.
- I nomi degli oggetti sono come path names di files.

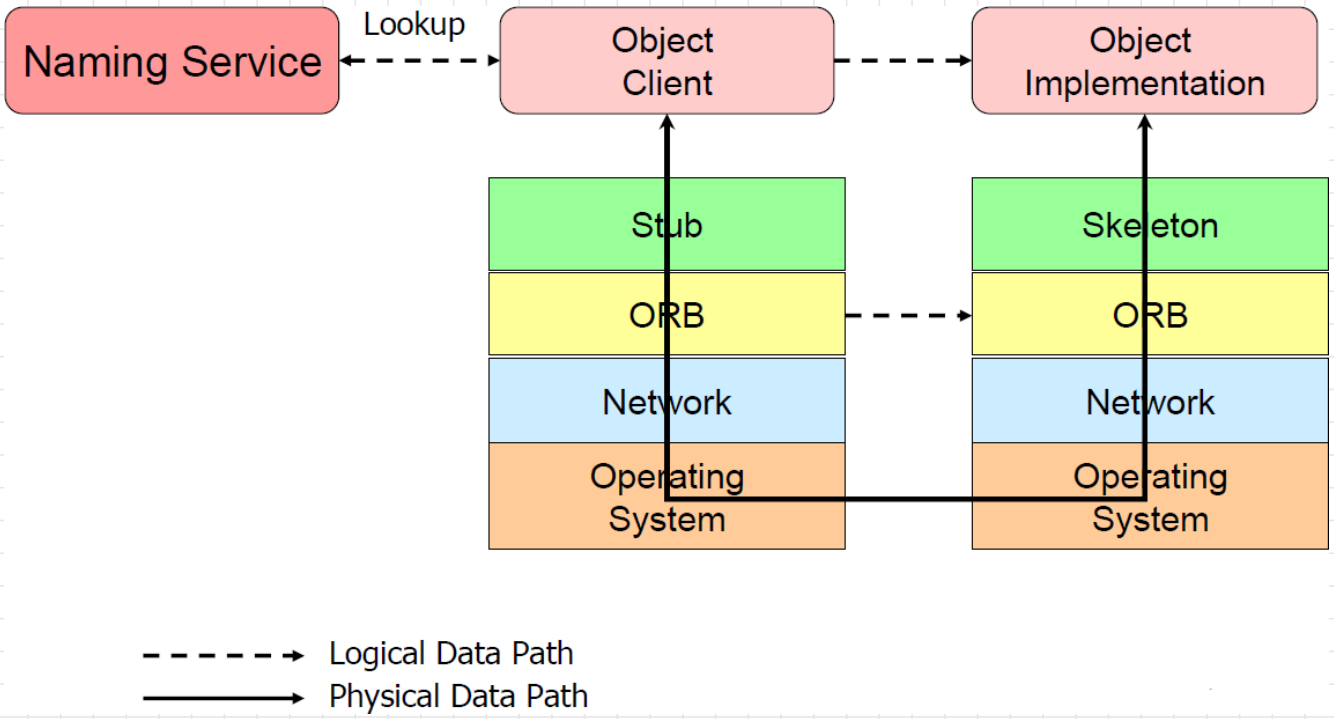
Naming service: Meccanismo lato client

Le caratteristiche dell'interazione dei client con un Naming Service sono le seguenti:

- Il processo Client contatta il Naming Server specificando il nome dell'oggetto desiderato.
- Il Naming Server restituisce lo IOR dell'oggetto al Client.
- Il Client costruisce uno stub appropriato e accede al servizio del Server.

Iterazioni tra Naming Service, Client e Server

Il Client chiede lo IOR al Naming Server

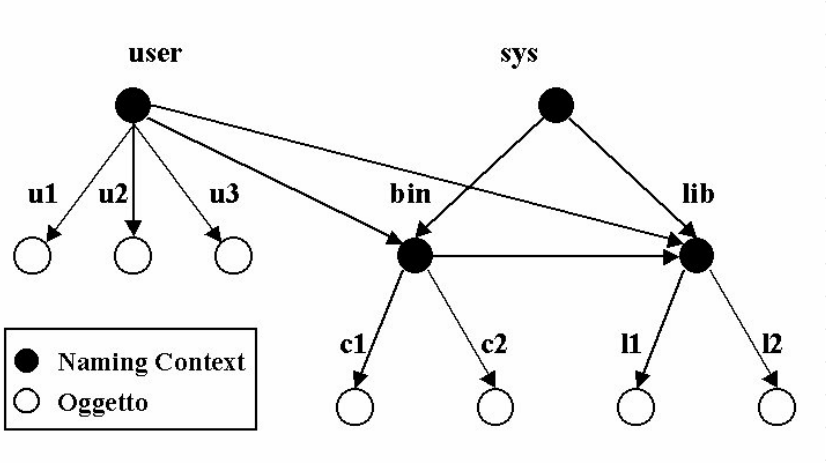


Terminologia

- Una associazione nome-IOR è chiamata name binding.
- Ogni binding identifica esattamente un riferimento ad un oggetto, ma un riferimento può essere associato a più di un nome (può avere più nomi).
- Un naming context è un oggetto che contiene name bindings. I nomi all'interno di un contesto devono essere unici.
- Binding a name to a context vuol dire aggiungere una associazione nome–IOR al contesto. context.
- I naming contexts possono contenere bindings ad altri naming contexts, così da formare un grafo.
- Resolving a name significa cercare un nome in un contesto e ottenere lo IOR associato a quel nome.

Gerarchie di nomi

Il Name Service gestisce una gerarchie di nomi:

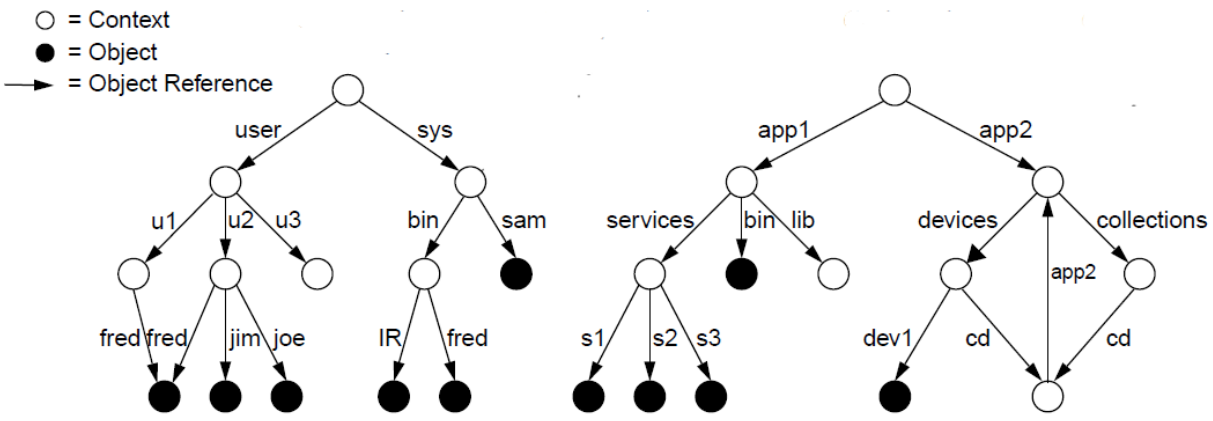


Ad un oggetto CORBA viene quindi associato un nome che appartiene ad un Name Context.

Il nome qualificato di un oggetto è dato da una sequenza di Name contexts seguito dal nome proprio dell'oggetto. Es., un nome qualificato dell'oggetto "c1" è "user/bin/c1".

Gerarchie di nomi: altro esempio

Altro esempio di grafo dei nomi.



Notare i cicli nel grafo e gli oggetti che possono avere diversi nomi qualificati.

Interfaccia IDL

L'interfaccia degli oggetti name contexts è definita nel modulo CosNaming. Vediamo la definizione dei nomi (NameComponent) e dei nomi qualificati (Name)

```
module CosNaming {

   typedef string Istring;

   struct NameComponent {
       Istring id;
       Istring kind;
   };

   typedef sequence<NameComponent> Name;

   ...
```

L'id contiene il nome, mentre kind dovrebbe contenere il tipo di componente, come le estensioni dei file (es., ".txt", ".exe", ...), ma in pratica non è usato.

Interfaccia IDL

L'interfaccia NamingContext per i contesti di nomi è:

```
module CosNaming {

   ...
   interface NamingContext {

     // Definizione di eccezioni, enumerazioni usate, etc.

   enum NotFoundReason {
         missing_node, not_context, not_object
   };


   exception NotFound {
         NotFoundReason why;
         Name rest_of_name;
   };

   ...
```

Operazioni principali

Le operazioni nel NamingContext per il bindig di oggetti.

```
interface NamingContext {

...
  // Definizione delle operazioni di bind

void bind(in Name n, in Object obj)
   raises(NotFound, CannotProceed,
            InvalidName, AlreadyBound );


 void rebind(in Name n, in Object obj)
    raises(NotFound, CannotProceed,
             InvalidName)

...
```


Operazioni principali

Le operazioni nel NamingContext per il bindig di contesti.

```
interface NamingContext {
...

void bind_context(in Name n, in NamingContext nc)
   raises( NotFound, CannotProceed,
             InvalidName, AlreadyBound );

void rebind_context(in Name n, in NamingContext nc)
   raises(NotFound, CannotProceed, InvalidName);

NamingContext new_context();

NamingContext bind_new_context(in Name n)
   raises( NotFound, AlreadyBound,
             CannotProceed, InvalidName);

...
```

Operazioni principali

Le operazioni di unbind() e destry().

```
    interface NamingContext {
    ...

   void unbind(in Name n)
          raises(NotFound, CannotProceed, InvalidName);


   void destroy() raises(NotEmpty);

    ...
```


Operazioni principali

Le operazioni dal lato del Client.

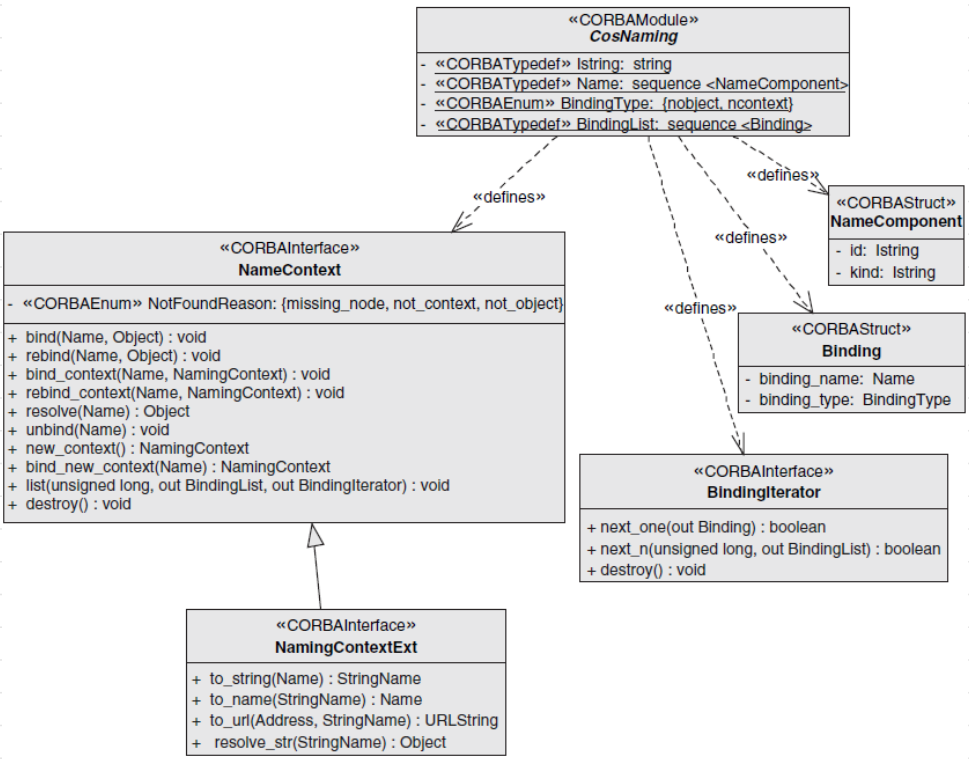
```
    interface NamingContext {
    ...

    // Client lookup operations - resolve a name to
    // an object reference or obtain an iterator
    // that returns a list of names in a context.

    Object resolve (in Name n)
       raises(NotFound, CannotProceed, InvalidName);

    void list( in unsigned long how_many,
               out BindingList bl, out BindingIterator bi );

    }; // end of NamingContext

// Helper iterator class used when getting lists of names
   interface BindingIterator { ... };

    ...
```

Estensione al NamingContext

Per gestire i nomi qualificati come stringhe e utilizzare anche URL come nomi è definita l'interfaccia NamingContextExt.

```
interface NamingContextExt: NamingContext {

   typedef string StringName;
   typedef string Address;
   typedef string URLString;

   StringName to_string(in Name n) raises(InvalidName);

   Name to_name(in StringName sn) raises(InvalidName);

   URLString to_url(in Address addr, in StringName sn)
      raises(InvalidAddress, InvalidName);

   Object resolve_str(in StringName n)
      raises( NotFound, CannotProceed,
              InvalidName, AlreadyBound );
};
```


CosNamimg: UML



Naming Service: esempio lato Client

```
public class AphorismClient {
 public static void main(String args[])
  {
   try{
        ORB orb = ORB.init(args, null);
        org.omg.CORBA.Object objRef =
           orb.resolve_initial_references("NameService");

        NamingContext initctx = NamingContextHelper.narrow(objRef);
        NameComponent nc0 = new NameComponent("examples", "");
        NameComponent nc1 = new NameComponent("aphorism", "");
        NameComponent nc2 = new NameComponent("Guru", "");

        NameComponent path[] = { nc0, nc1, nc2 };

        Guru myGuru = GuruHelper.narrow(initctx.resolve(path));

        String wiseSaying = myGuru.enlightenMe();
        System.out.println(wiseSaying);

    } catch (Exception e) {
         System.out.println("Failed because : " + e) ;
        }
   }
}
```


Naming Service: esempio lato Server

Il programma Server:

- Il `Main`
  - Inizializzare l'Orb
  - Creare un Guru servant
  - Registrare il servant nel POA di default
  - Utilizzare la funzione di supporto per registrare l'oggetto server nel Naming Service
  - Chiedere al POAManager di aprire il flusso di richieste
  - Lanciare l'Orb
- Funzione di supporto `registerObjWithNameService`.
  - Seguire il path di nomi e creare i contesti specificati (se necessario)
  - Esegurie il Rebind dell'oggetto nel contesto appropriato

Naming Service: esempio lato Server

```
public static void main(String args[])
 {
  try{
      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object poaobj =
         orb.resolve_initial_references ("RootPOA");

      POA root_poa = POAHelper.narrow (poaobj);
      POAManager poa_manager = root_poa.the_POAManager();

      Guru theGuru = new Guru();

      byte[] oid = root_poa.activate_object(theGuru);
      org.omg.CORBA.Object ref =
              root_poa.id_to_reference(oid);

    ...
```


Naming Service: esempio lato Server

```
public static void main(String args[])
{
 try{
     ...

     org.omg.CORBA.Object objRef =
        orb.resolve_initial_references("NameService");

     NamingContext initCtx = NamingContextHelper.narrow(objRef);

     NameComponent nc0 = new NameComponent("examples", "");
     NameComponent nc1 = new NameComponent("aphorism", "");
     NameComponent nc2 = new NameComponent("Guru", "");

     NameComponent path[] = { nc0, nc1, nc2 };

     registerObjWithNameService(initCtx, path, ref);

     poa_manager.activate ();
     orb.run();
    }
  ...
}
```

Naming Service: esempio lato Server

La procedura registerObjWithNameService.

```
private static void registerObjWithNameService(NamingContext root,
  NameComponent[] serverName, org.omg.CORBA.Object obj) throws
      InvalidName, AlreadyBound, CannotProceed, NotFound
  {
   NamingContext currentContext = root;
   NameComponent singleElement;

   //Si segue il cammino dei contesti
   for(int i=0;i<serverName.length-1;i++) {

     singleElement = serverName[i];
     try {
          currentContext = NamingContextHelper.narrow(
                    currentContext.resolve(singleElement));
     }

     //Si crea un contesto se non esiste
     catch (NotFound nf) {
        currentContext = currentContext.bind_new_context(singleElement);
     }
   }
   singleElement = serverName[serverName.length-1];
   currentContext.rebind(singleElement, obj);
  }
```


Implementation Repository (IMR)

Le specifiche CORBA descrivono brevemente il concetto di Implementation Repository (IMR).
Nella terminologia CORBA Implementation sta per server application, e Repository per una area persistente come un database o un file.
Quindi l'IMR è un servizio che mantiene un database di applicazioni server.
Tipicamente un IMR mantiene le seguenti informazioni riguardo ad un server:

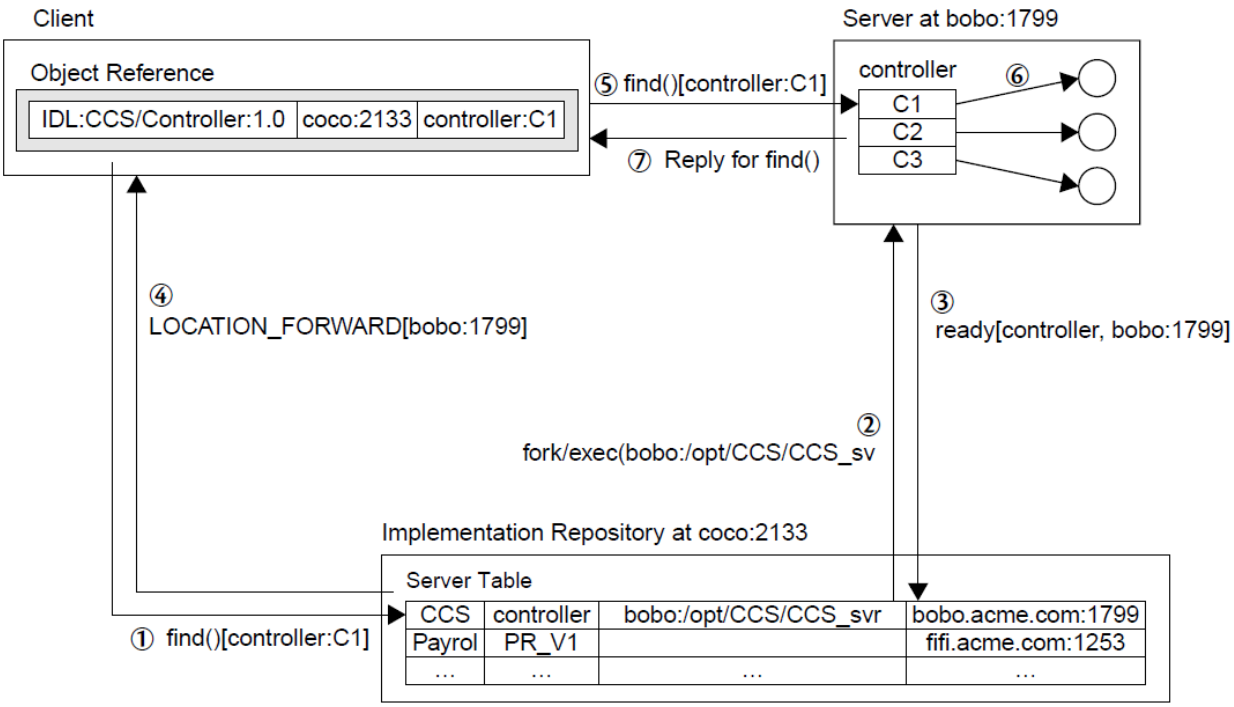
- Un nome che univocamenente identifica il server
- Un comando che l'IMR può utilizzare per far (ri)partire il server
- Lo stato del processo server (running oppure not-running)
- Se il processo è in esecuzione l'IMR memorizza l'host e la porta su cui il processo è in ascolto

Implementation Repository

Le specifiche CORBA forniscono solo una parziale definizione di un IMR riguardante le funzionalità di alto livello.
Nulla è specificato su come l'IMR deve essere amministrato.
Il motivo è che un IMR deve essere implementato ed amministrato in un modo che dipende dalla piattaforma.

- Differenti sistemi operativi hanno modi differenti per lanciare processi
- La presenza o meno di database nel sistema (in assenza si possono usare file testuali)
- Implementazioni di un IMR su un device embedded possono non aver accesso ad un file sistem: i dettagli sui server vanno mantenuti in RAM.

Indirizzamento indiretto



Indirizzamento indiretto

1. Il client ottiene uno IOR in cui però è indicato l'IMR (trasparente al client). Con esso chiede il servizio.
2. L'IMR manda in esecuzione il server
3. Il server indica la porta su cui ricevere le richieste (trasparente al programmatore: è nelle operazioni `ORB_init()` o `create_POA()`).
4. L'IMR indica al client quale porta contattare per la richiesta
5. Il client contatta il server
6. Il server attiva il servant
7. Il client riceve il servizio

Esempio IMR

Nell'esempio, un CLient chiama un Server ogni 6 secondi, da cui ottiene una stinga e poi ne fa lo shutdown.
Inizializzazione preliminare:

- lanciare l'orbd.
- lanciare il servertool.
- registrare il server con il comando:
  `register -server PersistentServer -applicationName s1 -classpath .`
  In questo modo il server viene registrato nell'IMR e attivato (cioè lanciato e poi ucciso per permettere la registrazione nel Name Service del servant). Il Naming Service nello IOR associato al servant registra però l'indirizzo del IMR.
- lanciare il PersistenClient .

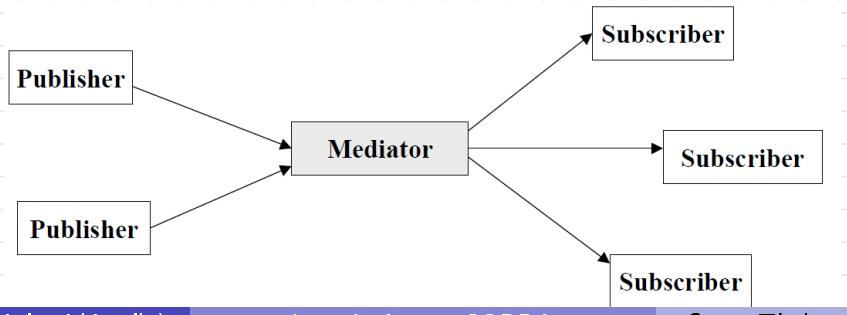
Event Service

L'invocazione di metodi CORBA standard risulta in una esecuzione sincrona di un'operazione fornita da un oggetto.

Per molte applicazioni è richiesto un modello di comunicazione tra oggetti più disaccoppiato (ad esempio, comunicazioni asincrone con più trasmettitori e ricevitori).

CORBA definisce un insieme di interfacce che abilitano una comunicazione asincrona e disaccoppiata tra oggetti chiamata Event Service.

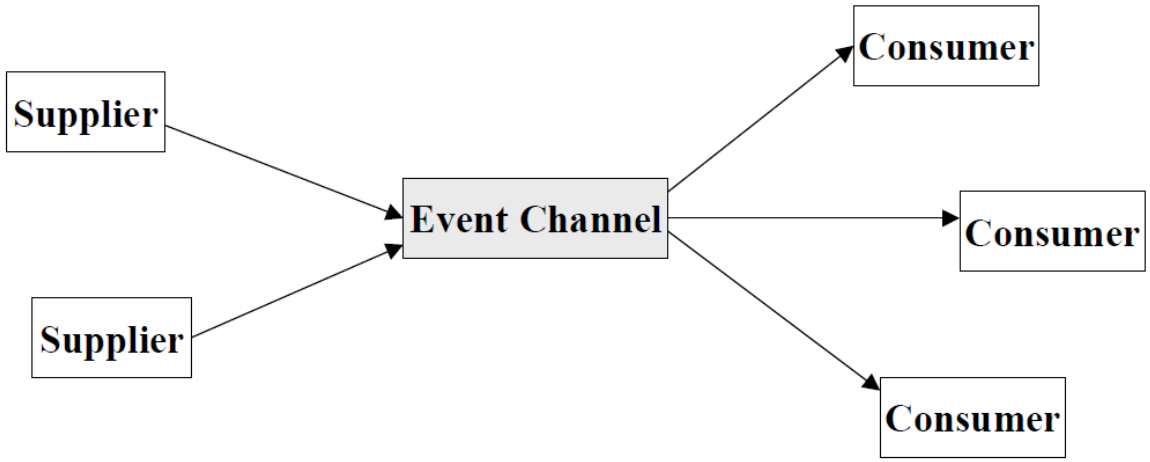
Il CORBA Event model è basata sull'observer pattern anche noto come "publish/subscribe" pattern.



Event Service

L'Event Service definisce tre ruoli:

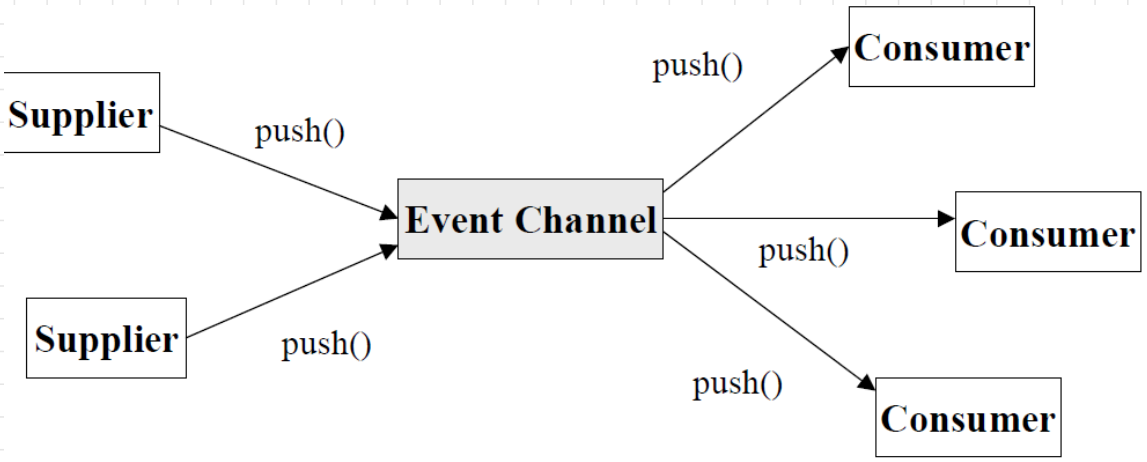
- il Supplier: che genera gli eventi
- il Consumer: che processa gli eventi
- l'Event Channel: che funge da mediatore e incapsula la semantica dell'accodamento e propagazione.



Push Communication

L'Event service supporta sia modelli di comunicazione *Push* che *Pull*.

Modalità Push:

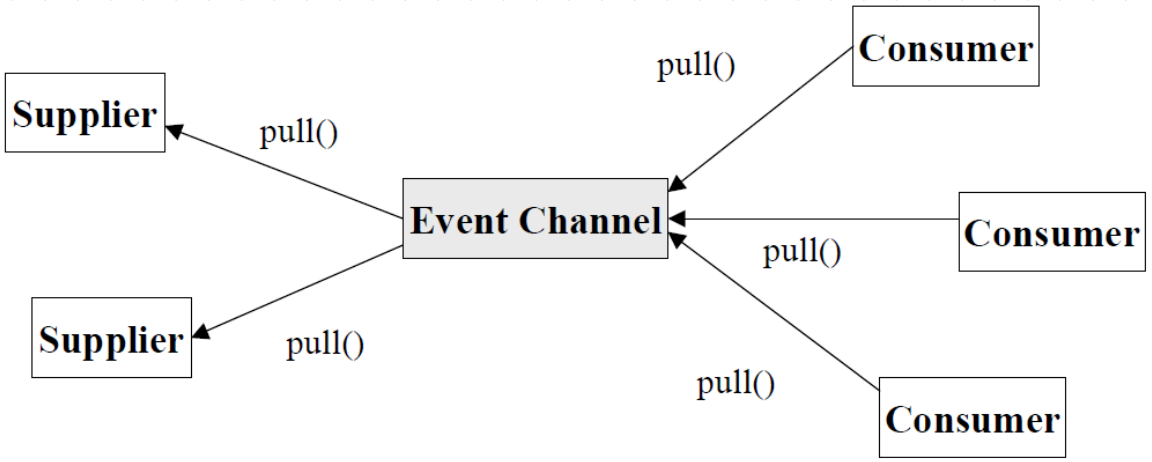


Il supplier spinge (pushes) i dati dell'evento nell'event channel

L'event channel, a sua volta, spinge i dati dell'evento a tutti i consumers.

Pull Communication

Modalità Pull:

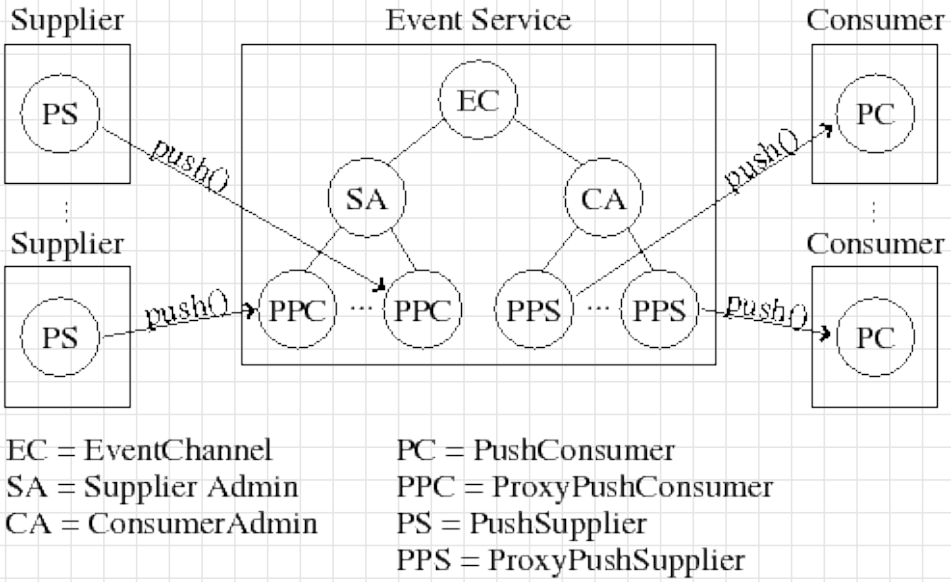


Il consumer richiede (pulls) i data dell'evento dall'event channel.

L'event channel, a sua volta, richiede i dati ai suppliers.

Push Communication

Illustriamo la modalità di comunicazione Push.



Interfacce IDL per modalità Push

```
module CosEventComm {
  interface PushConsumer {
    void push(in any data);
    void disconnect_push_consumer();
  };

  interface PushSupplier {
    void disconnect_push_supplier();
  };
};
```

```
module CosEventChannelAdmin {

  interface ProxyPushConsumer : CosEventComm::PushConsumer
  {
    void connect_push_supplier(
        in CosEventComm::PushSupplier push_supplier);
  };

  interface ProxyPushSupplier : CosEventComm::PushSupplier
  {
    void connect_push_consumer(
        in CosEventComm::PushConsumer push_consumer);
  };

  interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier();
    ProxyPullSupplier obtain_pull_supplier();
  };

  interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
    ProxyPullConsumer obtain_pull_consumer();
  };

  interface EventChannel {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
    void destroy();
  };
};
```


Funzionamento della modalità Push

Un Consumer deve implementare l'interfaccia *PushConsumer*. Questa ha un'operazione `push()` che è invocata per passare dati arbitrari relativi ad un evnto. Il metodo `disconnect_push_consumer()` è chiamato quando l'Event Service si vuole disconnetere dal consumer (Es., l'Event Channel sta per essere distrutto).

Un Supplier deve implementare l'interfaccia *PushSupplier*. L'Event Service invocherà `disconnect_push_supplier()` per disconnettere il Supplier.

L'*EventChannel* è il punto di contatto iniziale per un Event Service. Le sue funzionalità sono realizzate dai due oggetti `SupplierAdmin` e `ConsumerAdmin`, forniti da due operazioni.

SupplierAdmin è una factory interface che con il suo metodo `obtain_push_supplier()` crea un *ProxyPushConsumer* su cui un Supplier può richiamere il metodo `push()`. Questo invocherà direttamente o meno i metodi `push()` su ogni oggetto *PushConsumer* delle applicazioni consumer.

Funzionamento della modalità Push

Una applicazione Supplier svolgerà i seguenti passi:

- connettersi all'EventChannel.
- invocare `for_suppliers()` per ottenere il *SupplierAdmin*
- invocare `obtain_push_consumer()` per creare un oggetto `ProxyPushConsumer`
- invocare `connect_push_supplier()` per registrare il suo oggetto *PushSupplier*
- richiamare il metodo `push()` del `ProxyPushConsumer`

Funzionamento della modalità Push

Analogamente un'applicazione Consumer svolgerà i seguenti passi:

- connettersi all'EventChannel.
- invocare `for_consumers()` per ottenere il *ConsumerAdmin*
- invocare `obtain_push_supplier()` per creare un oggetto `ProxyPushSupplier`
- invocare `connect_push_consumer()` per registrare il suo oggetto *PushConsumer*
- il metodo `push()` sarà richiamato quando occorrerà un evento