CORBA

COMMON OBJECT REQUEST BROKER ARCHITECTURE

OVERVIEW OF CORBA, OMG'S OBJECT TECHNOLOGY FOR DISTRIBUTED APPLICATIONS

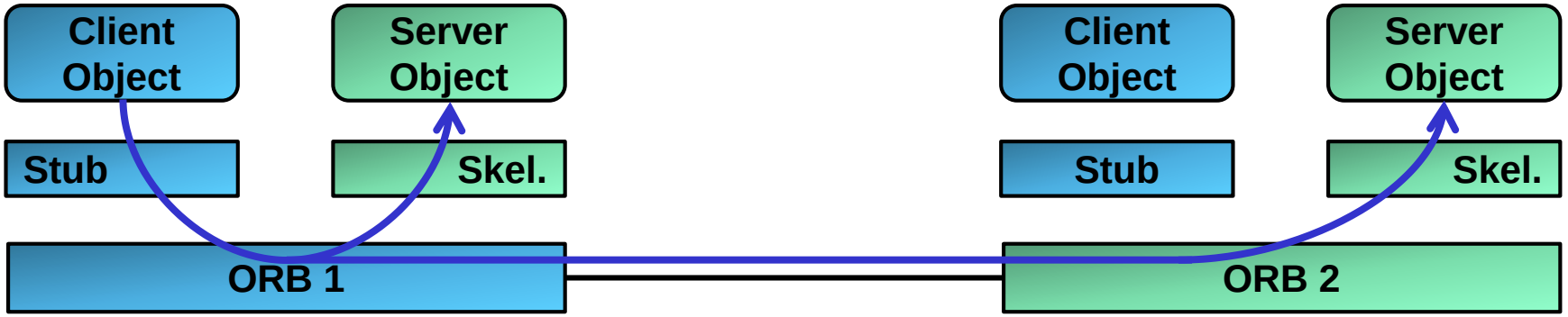
Peter R. Egli
peteregli.net

Contents

1. What is CORBA?

CORBA: Common Object Request Broker Architecture.

CORBA is a distributed object technology.
→ Objects call operations (methods) on other objects, either locally or on a remote objects.
→ CORBA provides interoperability between vendors and languages (e.g. objects developed in one programming language like C++ may call operations on objects developed in another language like Java).



Inter-ORB communication uses the GIOP (General Inter-ORB Protocol)

2. CORBA Elements

ORB:

The Object Request Broker dispatches operation calls to the right server object.

Stub:

The stub is a component that connects the client object to the ORB.

Skeleton:

The skeleton is a server-side component that, like the client stub, connects the server object to the ORB.

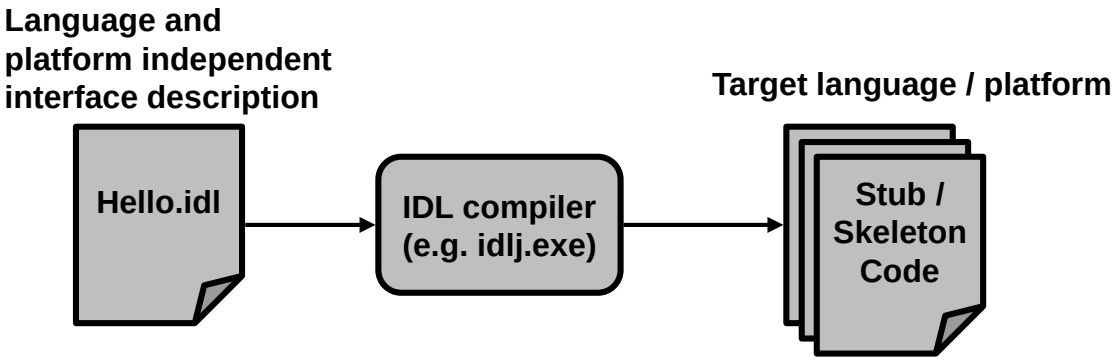
GIOP / IIOP:

GIOP: General Inter ORB Protocol.
IIOP: Internet Inter ORB Protocol.
Communication between ORBs uses a standard protocol. GIOP is a general interoperability protocol while IIOP, a variant of GIOP, is used in TCP/IP based networks.

3. The CORBA IDL (1/11)

IDL (Interface Description Language) defines an object interface in an abstract and implementation neutral way.
The IDL of an object represents an interface contract. This separates interface definition from the interface implementation.

The IDL compiler generates stub / skeleton code for specific languages (=language mapping or language binding).
Language mappings exist for a range of languages (Java, C++, Python, Ruby, COBOL etc.).



E.g. Java:

```
idlj.exe -fall -td ../HelloServer/src ../../Hello.idl
```

E.g. omniORB:

```
omniidl.exe -C. -bcxx ..\..\Hello.idl
```

3. The CORBA IDL (2/11)

The syntax of IDL inherited much from the C/C++ programming language.

Namespace:

Keyword `module` defines a namespace or scope (scoping for avoiding naming conflicts).
`module` can be omitted (the interface then is in the global IDL namespace).

Example:

```
//IDL
module HelloApp
{
  interface Hello
  {
    string sayHello();
    oneway void shutdown();
  };
};
```

The fully qualified name of the interface is `HelloApp::Hello` (note the scoping operator ‚::‘).

Reopening modules:

Modules can appear multiple times in IDL-files.

```
//IDL
module HelloApp
{
  interface Hello {...};
...
  interface Hello {...};
};
```

3. The CORBA IDL (3/11)

Main elements of an interface:

Operations:

→ Similar to operations (methods) in classes.

→ Arguments have classifiers (in, inout, out) for specifying the direction of the argument passing.

Attributes:

→ Similar to attributes in classes.

→ The IDL compiler generates setter and getter functions for each attribute except when it is declared readonly.

Example:

```
//IDL
module BankSimple {
  typedef float CashAmount; //type definition
  interface Account; //forward declaration
  interface bank {...};
  interface Account {
    readonly attribute string name;
    readonly attribute CashAmount balance;
    void deposit (in CashAmount amount);
    void withdraw (in CashAmount amount);
  };
};
```

3. The CORBA IDL (4/11)

IDL exceptions:

There are 2 types of exceptions:

a. System exceptions defined by CORBA (may be thrown at any time).

b. User defined exceptions (operations may throw multiple exceptions).

Example:

```
//IDL
module BankSimple {
  typedef float CashAmount; //type definition required so it can be used as operation argument
  interface Account {
    exception InsufficientFunds {
      string reason;
    };
    exception AccountBlocked {
      string reason;
    };
    void withdraw(in CashAmount amount)
      raises(InsufficientFunds, AccountBlocked);
    ...
  };
};
```

IDL includes:

The `#include` statement allows importing other IDL files:

```
// IDL
#include <BaseInterface.idl>
...
```

3. The CORBA IDL (5/11)

IDL operation invocation semantics:

IDL supports 2 operation invocation semantics:

A. Synchronous:

In synchronous operation calls, the caller is blocked until the operation terminates and has returned.
Synchronous operations correspond to operation calls on objects.

B. Oneway:

In oneway operation invocations, the client is not blocked during the invocation but may continue with other work (asynchronous operation).
E.g. oneway operations may be used to send logging information to a log server where it is not necessary to wait for the operation to complete.
N.B.: In oneway operations, there is no feedback to the client if the operation succeeded or failed.

```
//IDL
module BankSimple
{    ...
     interface Account {
          oneway void notice(in string text);
          ...
     };
};
```

3. The CORBA IDL (6/11)

Passing context information in operations:

Contexts allow mapping a set of identifiers to a set of string values. In the code that the IDL compiler generates, the context is passed like an ordinary argument.

```
//IDL
module BankSimple
{
    ...
    interface Account {
        void deposit(in CashAmount amount)
        context ("sys_time", "sys_location");
        ...
    };
};
```

Generated code:

```
public interface AccountOperations
{
    void withdraw (float amount, org.omg.CORBA.Context $context);
}
```

3. The CORBA IDL (7/11)

IDL interface inheritance:

Interfaces may inherit all operations and attributes of super-interfaces.
Multiple inheritance is possible.

```
// IDL
module BankSimple {
    interface Account {...};

    interface CheckingAccount : Account {...};

    interface SavingsAccount : Account {...};

    interface PremiumAccount :
        CheckingAccount, SavingsAccount {
        ...
    };
};
```

IDL `Object` interface type:

`Object` is the „mother of all interfaces“ (user defined interfaces implicitly inherit from `Object`).

Example interface Object operations:

Object::_create_request(...):	**Creation of requests using DII (Dynamic Invocation Interface).**
Object::_duplicate(...):	**Duplicate object references.**
Object::_narrow(...):	**Convert an object to a more specific type (kind of a dynamic typecast).**
Object::_is_a(...):	**Check whether an object may be narrowed to the specified type.**

3. The CORBA IDL (8/11)

IDL forward declarations:

Similar to C++, IDL allows forward declarations.

Forward declarations are required when interfaces reference each other.

```
// IDL
module BankSimple {
    interface Account; // Forward declaration of Account
    interface Bank {
        Account create_account (in string name);
        Account find_account (in string name);
    };
    // Full definition of Account.
    interface Account {
      ...
};
```

Constants in IDL:

Like other languages, IDL allows defining constants.

```
module BankSimple {
    interface Bank {
        const long MaxAccounts = 10000;
        const float Factor = (10.0 - 6.5) * 3.91;
        ...
    };
};
```

3. The CORBA IDL (9/11)

Standard IDL data types (1/3):

A. Basic types:

IDL defines the following basic types.

```
short, unsigned short, long, float, double, char, boolean, octet, any (=arbitrary IDL type)
```

B. Complex types:

IDL complex types are enumerations and structure types.

```
enum Currency {pound, dollar, yen, franc};

struct CustomerDetails {
    string name;
    short age;
};
```

C. Union:

Unions allow defining the type of a type member based on an argument.

```
// IDL
struct DateStructure {
  short Day;
  short Month;
  short Year;
};
union Date switch (short) {
    case 1: string stringFormat;
    case 2: long digitalFormat; //if argument short=1
    default: DateStructure structFormat; //defines the default format
};
```

3. The CORBA IDL (10/11)

Standard IDL data types (2/3):

D. String:

IDL supports bounded and unbounded strings.

```
//IDL
interface Account {
    // A bounded string with maximum length 10.
    attribute string<10> sortCode;
    // An unbounded string.
    readonly attribute string name;
};
```

E. Sequence:

A sequence is similar to a one-dimensional array. Again there are bounded and unbounded sequences. A sequence must be defined as a type before being used as an argument in an operation or attribute.

```
// IDL
module BankSimple {
    struct LimitedAccounts {
        string bankSortCode<10>;
        // Maximum length of sequence is 50.
        sequence<Account, 50> accounts;
    };
    struct UnlimitedAccounts {
        string bankSortCode<10>;
        // No maximum length of sequence.
        sequence<Account> accounts;
    };
};
```

```
// IDL
  module BankSimple {
    typedef sequence<string> CustomerSeq;
    interface Account {
      void getCustomerList(out CustomerSeq names);
        ...
      };
    };
```

3. The CORBA IDL (11/11)

Standard IDL data types (3/3):

F. Array:

Arrays are always fixed-size.

Arrays may be multi-dimensional.

Like sequences, arrays must be defined as typedef before being used as attributes or operation arguments.

```
//IDL
struct CustomerAccountInfo {
    string name;
    Account accounts[3];
};
typedef Account AccountArray[100];
interface Bank {
    readonly attribute AccountArray accounts;
};
```

G. Fixed:

Fixed point numbers are represented as a tuple {digit, scale}. Digit is the length of the number, while scale defines the position of the decimal point.

```
module BankSimple {
    typedef fixed<10,4> ExchangeRate; //range: (+/-)999999.9999

    struct Rates {
        ExchangeRate USRate;
        ExchangeRate UKRate;
        ExchangeRate IRRate;
    };
};
```

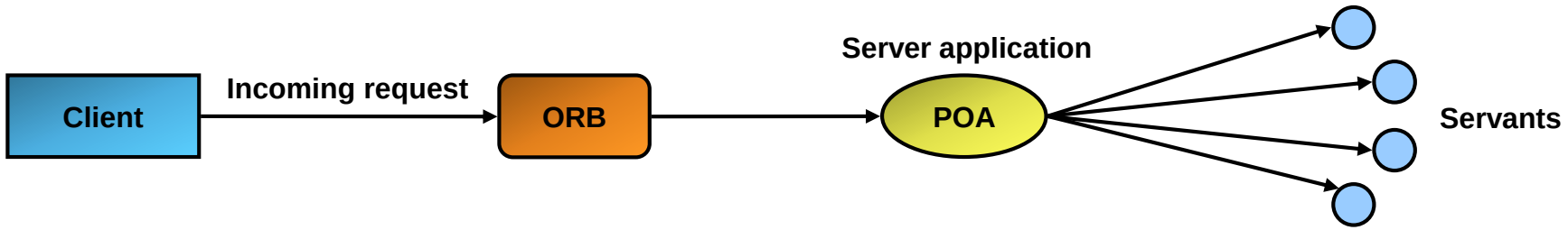
4. Server Programming Models - How to Implement the Server (1/5)

Over time, different ways of implementation emerged for the server side of the interface defined in the IDL.

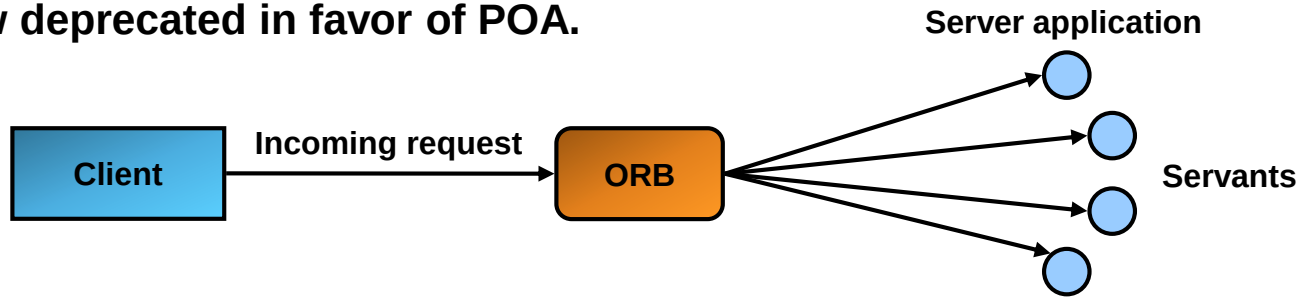
A. POA model versus ImplBase model (deprecated):

POA (Portable Object Adaptor) provides a standard interface for object implementations (portability of servants across different vendors).

POA allows 1 servant instance to serve multiple object identities (POA dispatches request to the right servant object).



ImplBase is less portable since it does not use a standard interface. The ImplBase model is now deprecated in favor of POA.



4. Server Programming Models - How to Implement the Server (2/5)

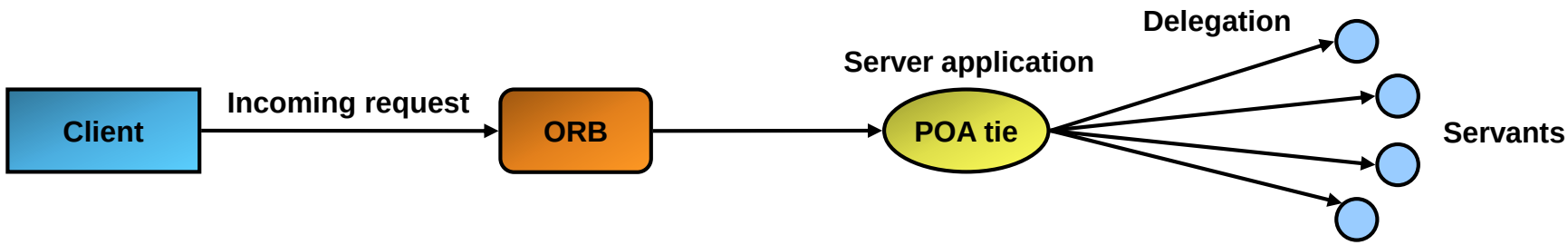
B. Inheritance versus delegation (tie):

In the inheritance model, a servant object implements either the ImplBase (deprecated) or POA base class.

In the delegation model, the IDL interface is implemented with 2 classes:

a. IDL-generated tie-class that dispatches requests to a delegate

b. User-defined implementation class (delegate)



Combining POA/ImplBase with Inheritance/Delegation:

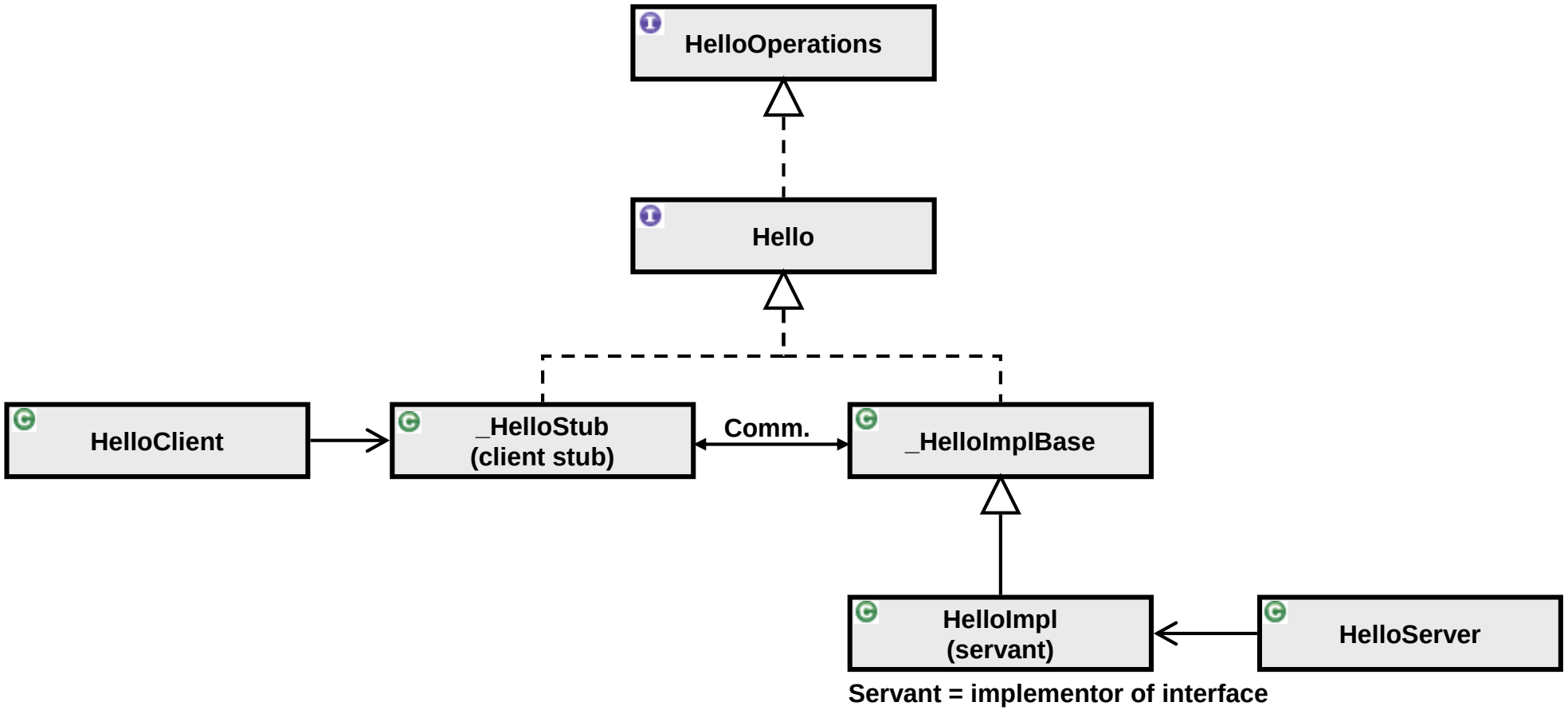
POA and ImplBase can be combined with inheritance and delegation.

	POA	ImplBase
Inheritance	POA-Inheritance Model	ImplBase – Inheritance Model (deprecated)
Tie	POA – Tie Model	ImplBase – Tie Model (deprecated)

4. Server Programming Models - How to Implement the Server (3/5)

ImplBase-Inheritance Model (deprecated):

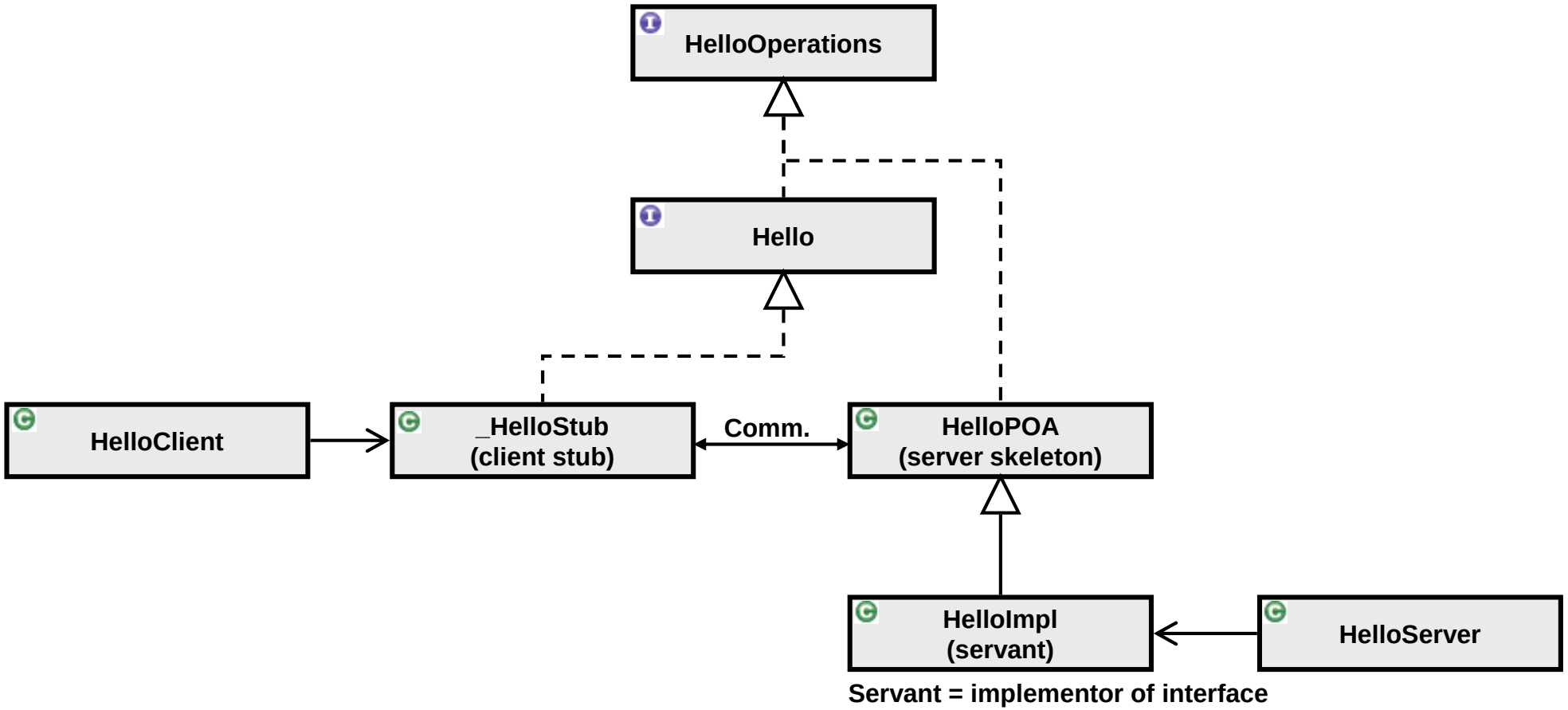
The server (servant) class directly implements ImplBase class (_HelloImplBase in the example).



4. Server Programming Models - How to Implement the Server (4/5)

POA Inheritance Model:

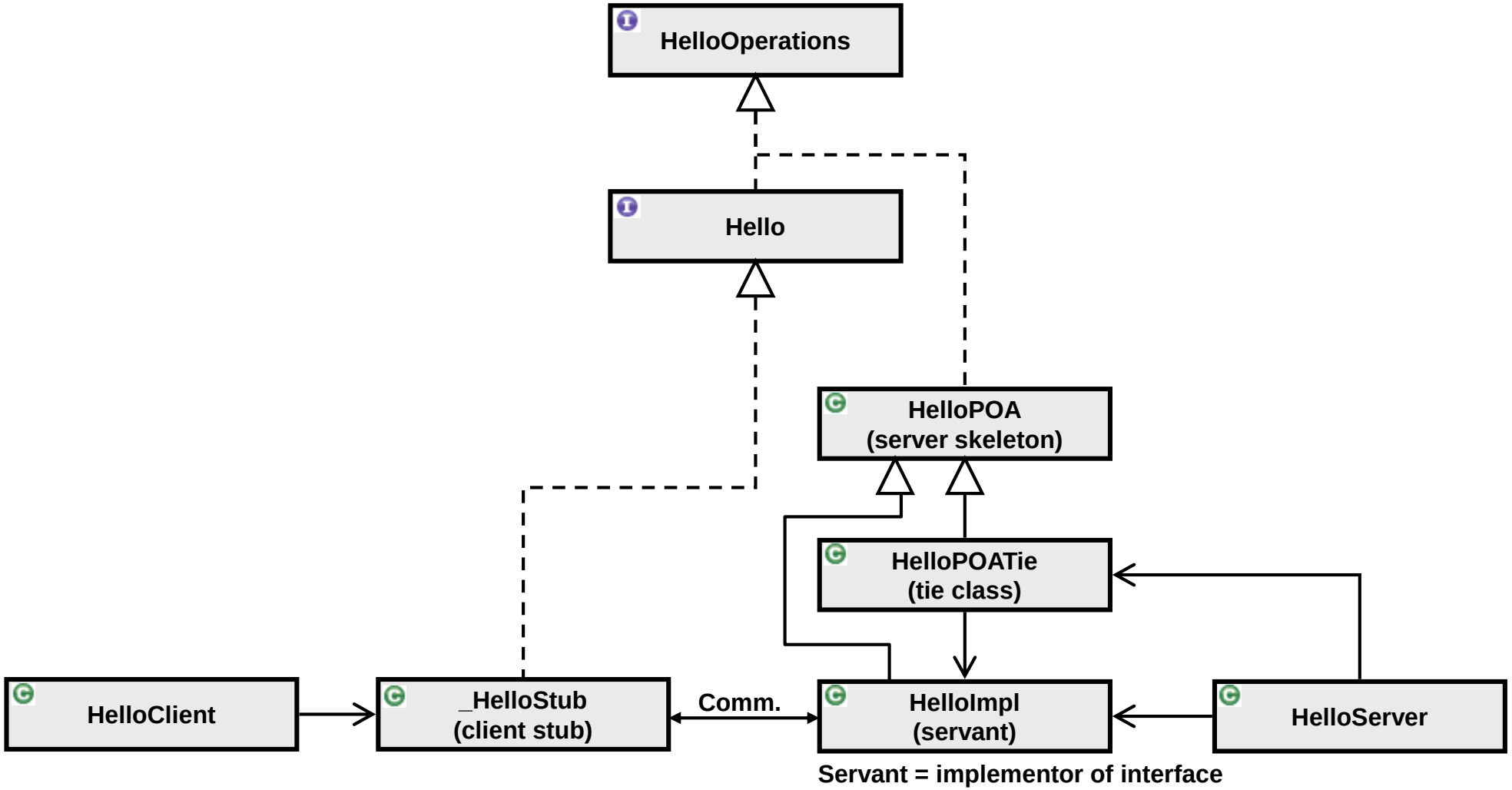
The POA is the standard interface for connecting servants to the ORB.



4. Server Programming Models - How to Implement the Server (5/5)

POA/Tie Model:

The POA-Tie dispatches requests to servant delegates.



5. COS – CORBA Object Services

COS defines additional services that extend the CORBA 2.0 standard.
These service are, however, rarely used in practice.

Service	Description
Life Cycle Service	Support for the creation, copying, moving and deleting of objects.
Persistence	Common interface for storing / retrieving objects from storage.
Naming Services	Primary object location service, mapping of human readable names to object references.
Event Service	Event notification mechanism (event supplier and consumer objects). Push (publish - subscribe) and pull (=polling) models.
Concurrency Control	Resource locking, transaction locks.
Transaction Service	Flat and nested transactions (sub-transactions).
Relationship Services	Explicit definition of relationships of objects.
Query Service	Mapping CORBA objects to relational DBs (a kind of ORM).
Licensing Service	Controlled access to objects.
Property Service	Association of properties to objects at runtime (properties are not part of IDL interface).
Time Service	Synchronization of clocks on different hosts.
Security Service	Authentication and propagation of credentials, encryption.

6. DII – Dynamic Invocation Interface

Static invocation:

In a static invocation, a call of an operation is performed through a client stub.
The client stub needs to be linked to the client (thus client stub must be available @ compile time).

Dynamic invocation with DII:

The dynamic invocation does not need or use a client stub but dynamically (@ run-time) constructs a client stub (=proxy object).
DII allows calling operations on objects that are not known @ client compile time.
From the server point of view, DII is identical to static invocation.
DII uses the IFR (InterFace Repository = central registry for interfaces) for locating the right server object.

DSI – Dynamic Skeleton Interface:

DSI is the server-side equivalent of DII.

7. ORB - Object Request Broker (1/4)

ORB functions:

A. Object location (providing client location transparency)
B. Data marshaling
C. Server side: Deliver requests to objects (=servants)
D. Client side: Return output values back to the client

ORB-Interoperability:

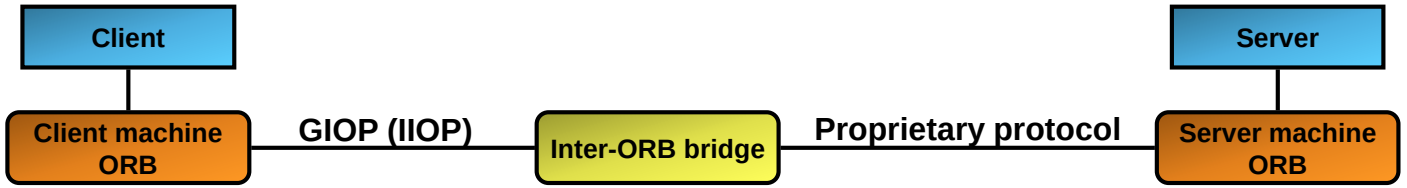
Interoperability between ORBs can be achieved with:

a. Employing ORBs that use the standard protocol GIOP (General Inter-ORB Protocol)

→ IIOP (Internet Inter-ORB Protocol) is a variant of GIOP
→ GIOP / IIOP uses interoperable object identifiers (IOR: Interoperable Object Reference)

b. Use of bridges between different ORB-protocols

→ Bridges translate from one ORB-protocol to another



7. ORB - Object Request Broker (2/4)

What is location transparency?

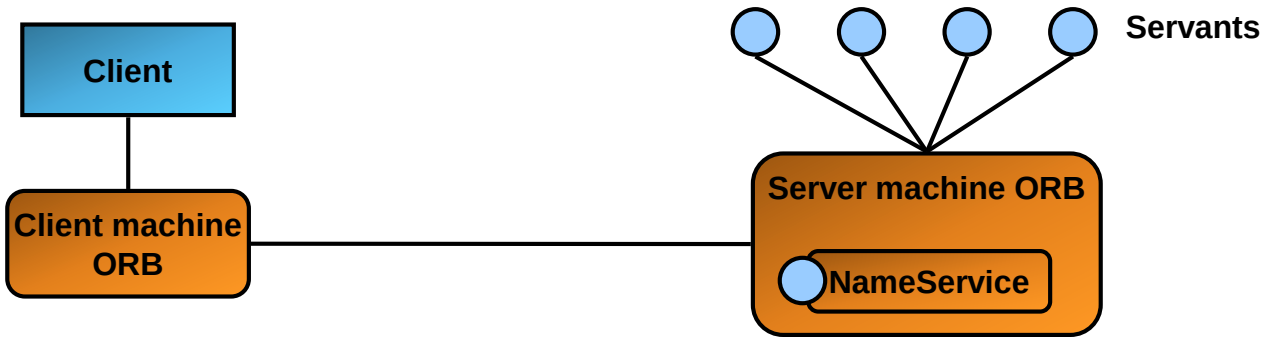
The location of remote objects (in which server they live) is hidden to the client application.
But: The location must be known to the client's ORB instance.

How to locate (find) remote objects:

1. Via direct ORB-connection:

```
ORB orb = ORB.init(„-ORBInitialPort 1234 -ORBInitialHost RemoteHost", null);
org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);
NameComponent nc = new NameComponent("Hello", "");
NameComponent path[] = {nc};
Hello helloImpl = HelloHelper.narrow(ncRef.resolve(path));
```

→ **Argument in `ORB.init()` specifies remote host and port.**
→ **Not true location transparency (remote host address must be passed into client application).**



7. ORB - Object Request Broker (3/4)

2. Use naming service:

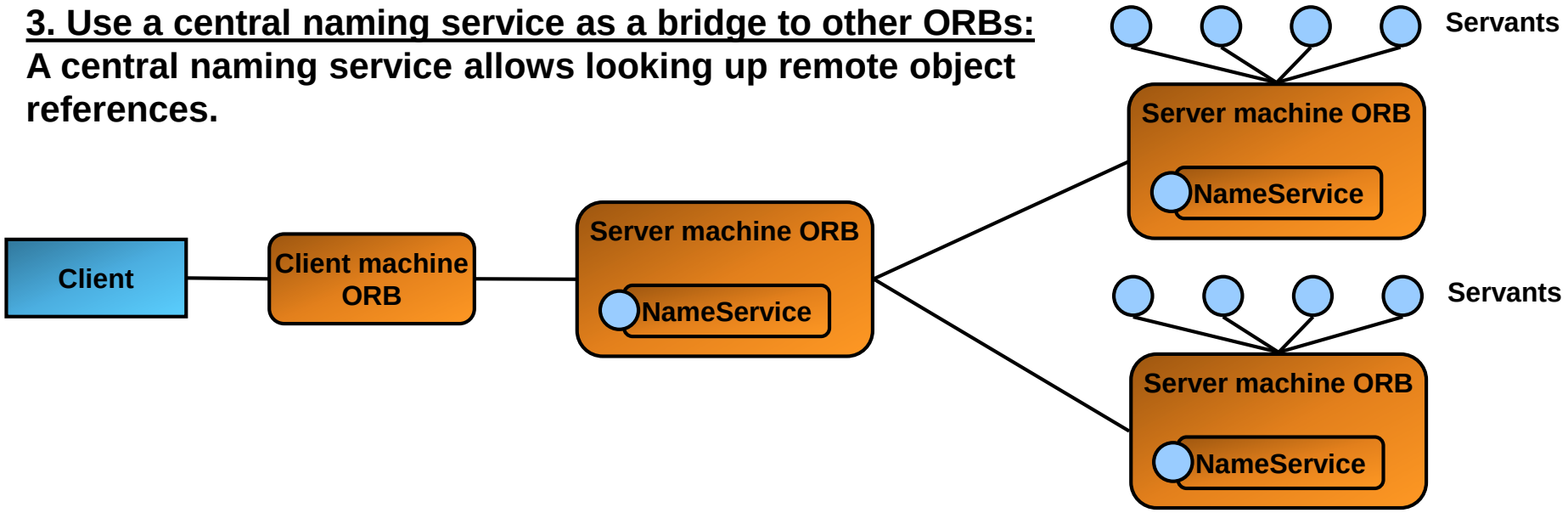
```
ORB orb1 = ORB.init(„-ORBInitialPort 10001 -ORBInitialHost orbhost1.net", null);
// Initialize another ORB reference
ORB orb2 = ORB.init(„-ORBInitialPort 10002 -ORBInitialHost orbhost2.net", null);
// Get references to the Naming Services
org.omg.CORBA.Object nc1Ref = orb1.resolve_initial_references("NameService");
org.omg.CORBA.Object nc2Ref = orb2.resolve_initial_references("NameService");

// Narrow the Naming Service references to NamingContexts and use them
...
```

→ The client still has to deal with multiple remote naming service objects (1 for each remote ORB).

3. Use a central naming service as a bridge to other ORBs:

A central naming service allows looking up remote object references.



7. ORB - Object Request Broker (4/4)

4. Use IOR – Interoperable Object References:

IORs are stringified object references.

IORs contain information about the remote object in order to:

a. locate the remote object's home ORB (address, port)

b. construct a local stub from the string representation

The IOR can be obtained as follows:

```
ORB orb = ORB.init(args, null);
HelloImpl helloImpl = new HelloImpl();
helloImpl.setORB(orb);
orb.object_to_string(helloImpl)
```

An IOR looks like:

```
IOR:000000000000001749444c3a48656c6c6f4170702f48656c6c6f3a312e300000000000010000000000000006e00010200000000d3137322e32302
e39322e353100000ed000000019afabcb00000000024b3bc3e8000000080000000000000000140000000000000200000001000000200000000000001000
0100000002050100010001002000010109000000010001010000000026000000020002
```

The IOR must be passed to client in some way (web page, web service etc.).

8. Pros and cons of CORBA

- Language independence (different mappings defined, e.g. C++, Java, Python, COBOL)
- OS independence
- Viable integration technology for heterogenous legacy systems
- Strong data typing

- Complex standard, difficult to implement for CORBA product vendors
- Too complex to use compared to the features and services it provides
- „Firewall-unfriendly“ because CORBA (IIOP) uses arbitrary TCP ports
- Lack of security