Distributed Systems

6. CORBA
(Common Object Request Broker Architecture)

CORBA Standardization (1)

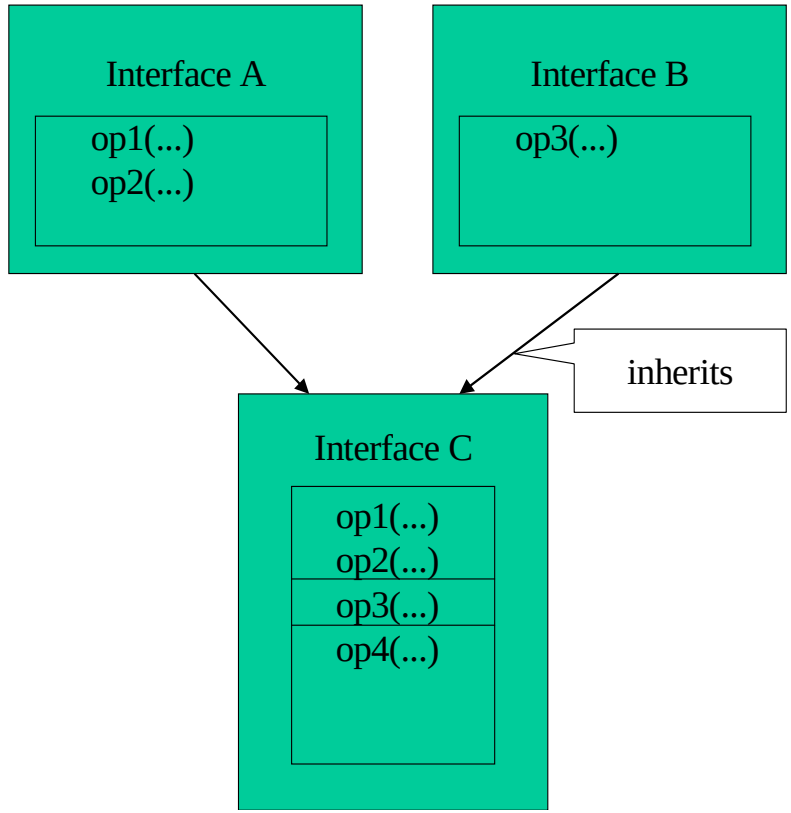
- CORBA is a (part of a) standard of OMG (Object Management Group) – not a software
- Many CORBA-based products are on the market (partly free-ware)
  - Over 70 ORBs + hundreds of other products
- Main features
  - Heterogeneous Distributed Computing
  - Programming language- and platform-independent
  - Component/Object-based software development
  - Internet Inter-ORB Protocol (IIOP)
    - Enables ORBs of different vendors to cooperate
- Newest version: CORBA 3

CORBA Objects (1)

- Objects
  - Objects model an entity or a concept
  - Objects have a unique identification
  - Operations
    - Actions performed by the objects
    - Signature: name, set of parameters, set of result types
    - Requests accepted by an object may (but need not to) be executed in parallel

CORBA Objects (2)

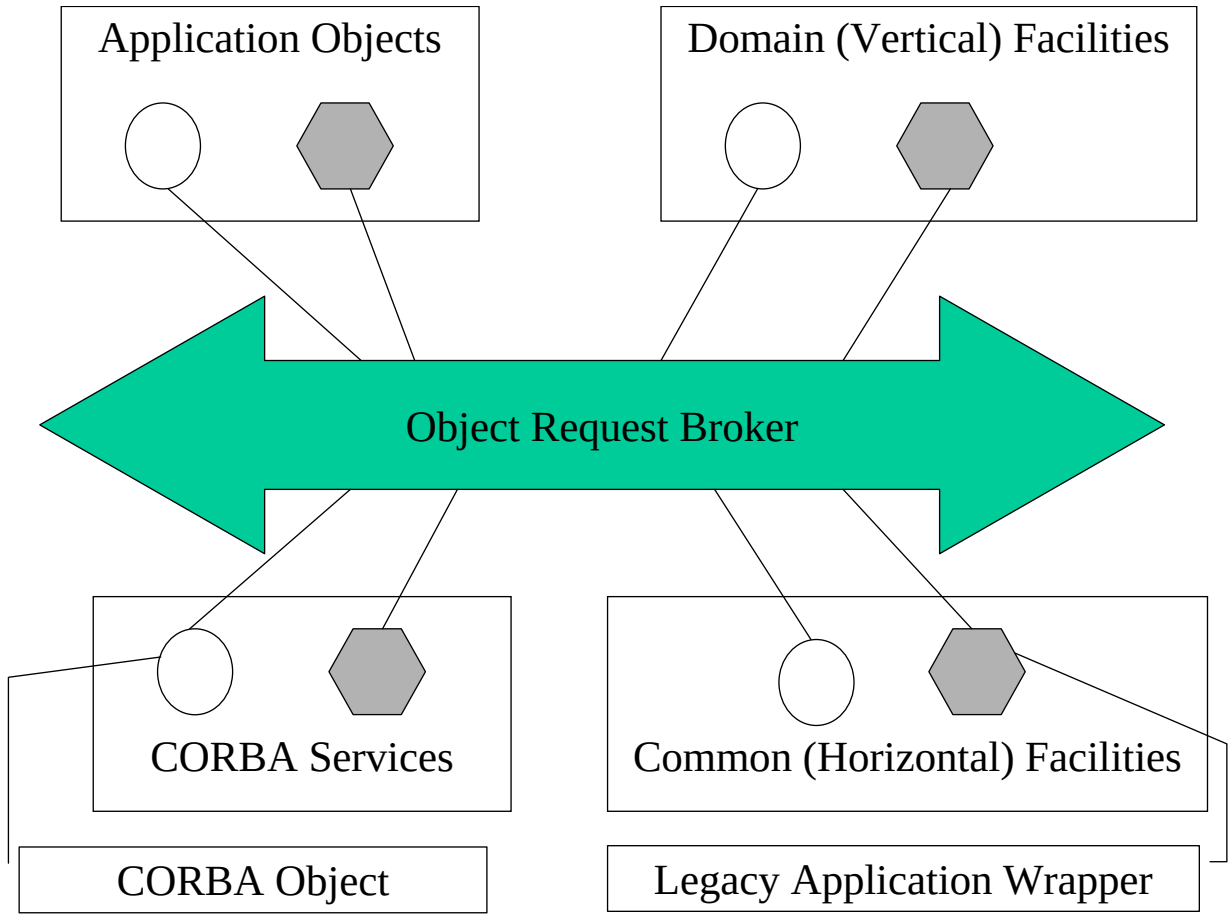
- Interfaces
  - An interface is a collection of signatures
  - Inheritance is defined: If interface C inherits from interface A, then C offers all operations of A and maybe more. Therefore, C is substitutable for A
  - Substitutability is not symmetrical, but transitive



CORBA Reference Model (1)

- Architectural framework for the standardization of
  - Interfaces to infrastructure
  - Services that applications can use
- Object Request Broker (ORB)
- Corba Services
  - Naming, event, persistence, concurrency control etc.
- Common (horizontal) Facilities
  - Information-, systems-, interface-, task-management etc.
- Domain Facilities
  - Telecommunications, e-commerce, health-care etc.
- Application Interfaces
- Corba supports 3-tier client/server solutions
  - View Objects
  - Server Objects
  - Legacy Applications

CORBA Reference Model (2)



Overview of CORBA Services

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

The ORB message bus (1)

- *CORBA specifies of the functionality of the ORB (the "message bus").*
- Location Transparency
  - It should be equally easy to invoke operations on an object in a remote address space as in the same address space
- Programming Language Transparency
  - Interface Definition Language (OMG IDL) is provided
  - The client programmer needs only the definitions in IDL
  - The interface between client and sever is completely independent from the programming language – even from the actual ORB technology (at least principally)
  - The IDL compiler generates
    - Code, based on the language mappings
    - Stub code (translates – marshals – the data structures of the implementation language into serial "wire"-format)
    - Skeleton code (unmarshals the serialized representation for the implementation – maybe in a different language)

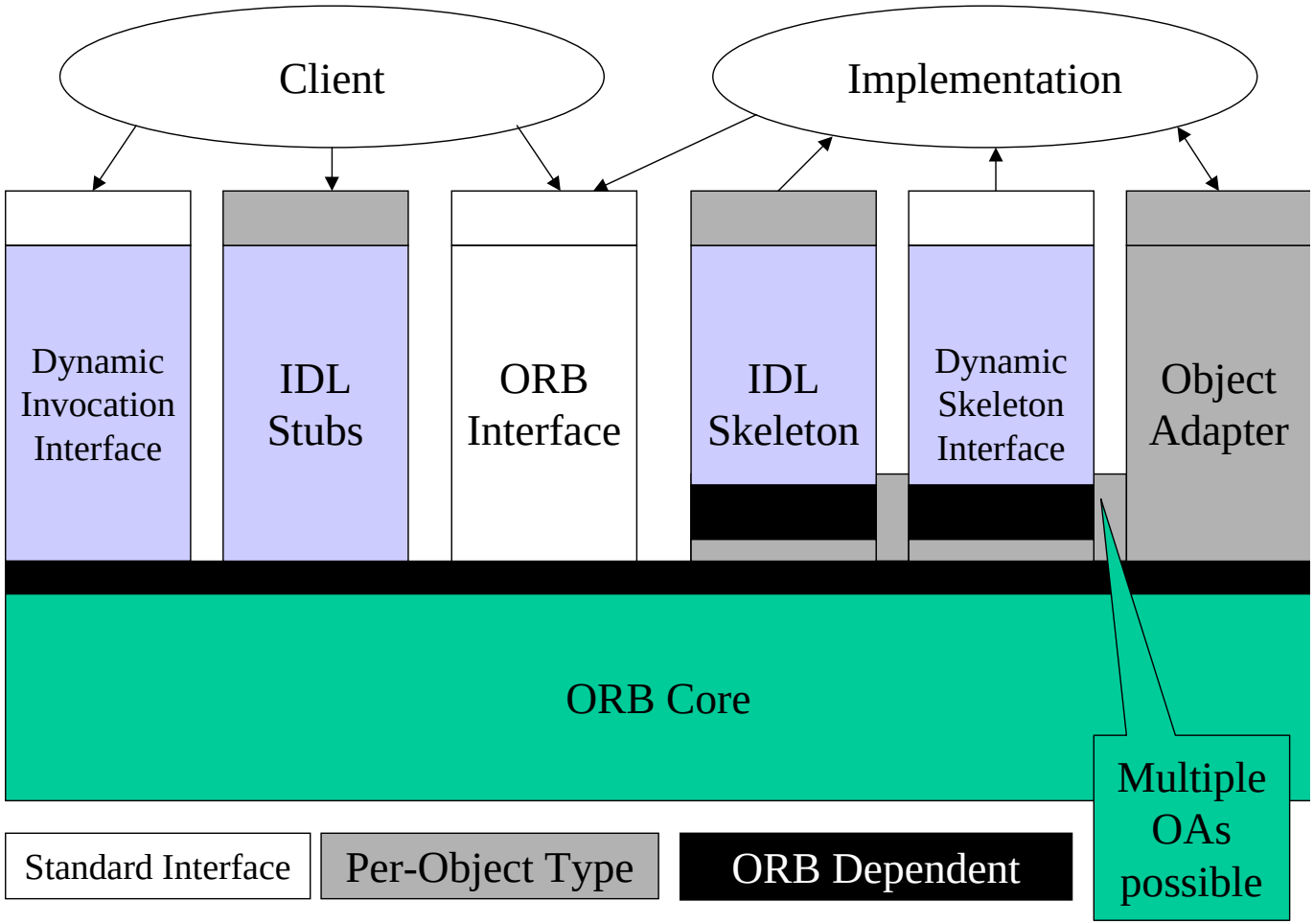
The ORB message bus (2)

- Different implementations of the same service are possible
- Different ORBs can cooperate via the
  - General Inter-ORB Protocol (GIOP), realized over TCP as the
  - Internet Inter-ORB Protocol (IIOP)
- Trading Service
  - Tries to find the service, with the best fitting QoS (Quality of Service) – e.g. performance, cost, location etc.
- Pseudo Objects
  - Defined in Pseudo-IDL (PIDL)
  - They are usually just linked to the CORBA-application
  - Their operations are called in the same way as those of real CORBA objects
  - Their references and data types cannot be passed to real CORBA objects as parameters

ORB structure (1)

- Stub/Skeleton
  - To convey requests and replies over the ORB
  - Must be static: known at compilation time
- Dynamic Invocation/Skeleton
  - For interfaces built dynamically at run-time
  - Provides the generic invoke operation
    - Object reference, method id, list of in/out parameters
- ORB interface
  - Mainly for initialization and object reference manipulation
- Object Adapter
  - Extra facilities, e.g. management of operating system processes, notification about ready to receive requests

ORB structure (2)



Object Reference Model (1)

- Object Reference
  - A reference is a handle to an object
    - A given reference always denotes the same object
    - Several distinct references may denote the same object
  - Objects can be passed as parameters or results
    - Reference semantics
    - Copy semantics – since CORBA 2.0, as *valuetype*
  - References are opaque to their clients
- Object Implementation
  - Provides actual implementation
  - The methods are accessed by the skeleton

Object Reference Model (2)

- Types
  - Object types are subtypes of the type Object (inheritance)
  - Non-object types (numeric, string, Boolean)
  - Type Any (can store any legitimate value of a CORBA type in a self-describing manner)
  - Structured types (structures, arrays, sequences, unions)
- Interfaces
  - Inheritance
- Attributes
  - Attributes in an interface are logically equivalent to an
    - Accessor and a modifier operation
    - Similar to C# properties

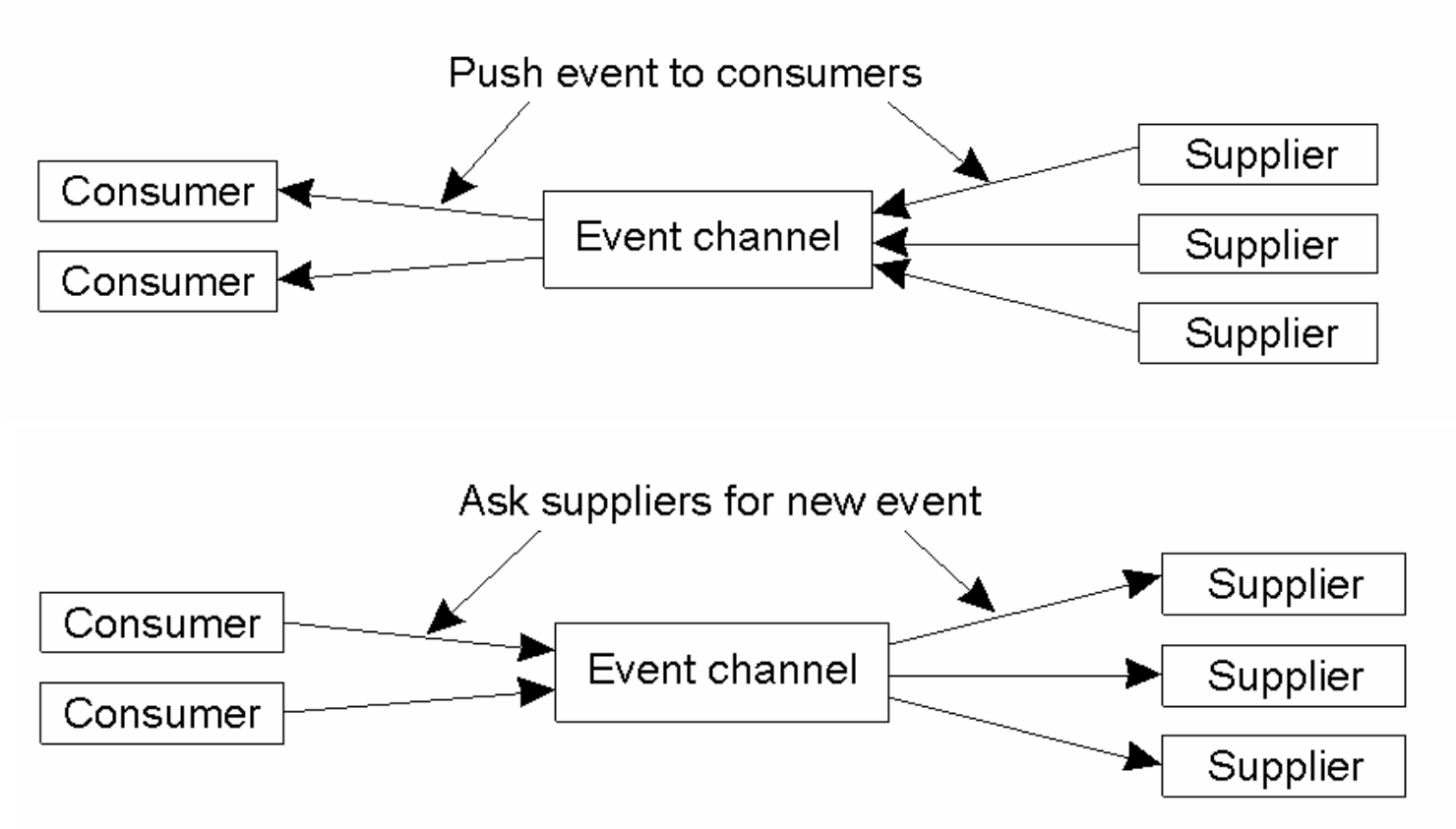
Operation execution and semantics

- Synchronous
  - This is the “normal” way of communication
  - The client waits until the operation ends
  - At-most-once semantics
- Deferred Synchronous
  - Execution is asynchronous, the result can be polled
  - Only for dynamic invocation
- One-way
  - The client invokes an operation and the ORB does its best to deliver it (“best-effort”)
  - No response, no result

Event and Notification Services (1)

- Method call alone is too restricted
  - 1:n communication
- Event service allows
  - n:m communication
  - Push and pull model is supported
- Drawbacks
  - No persistence of the events
    - The partners need to be connected to the channel
    - Events are lost for non-connected consumers
    - Event propagation is unreliable
  - No filtering
- Notification service provides filtering for the events

Event and Notification Services (2)



Messaging (1)

- Message exchange
  - Persistent and asynchronous
- Callback
  - Client provides additional interface with each request
  - Response is delivered by a call via this interface
  - Can be fully implemented by the client

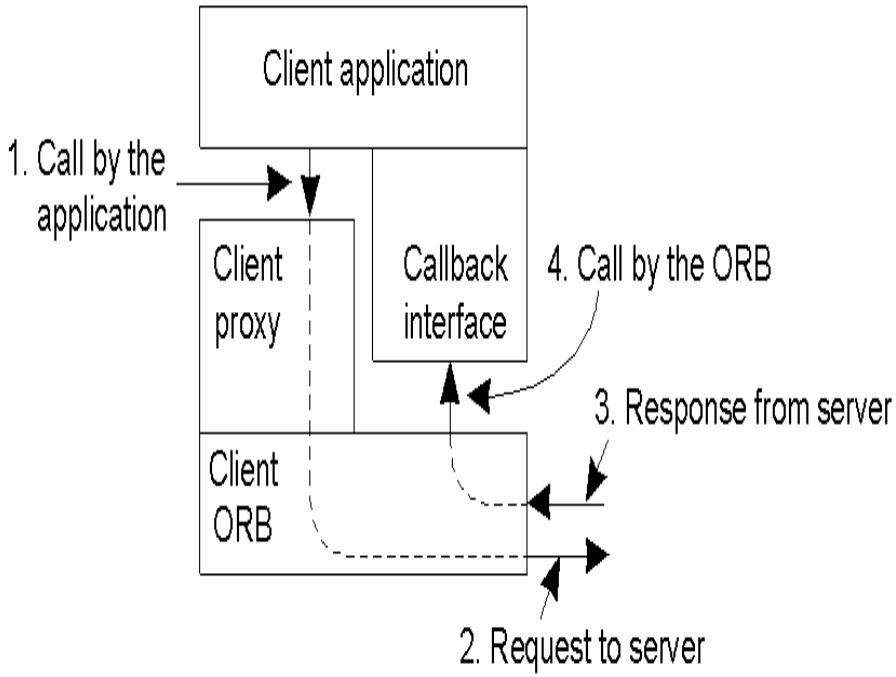
```
int add(in int i, int j, out int k);  ⇒
void send_add(in int i, int j,);  // Called by client
void rcv_add(in int k);           // Called by client's ORB
```

Messaging (2)

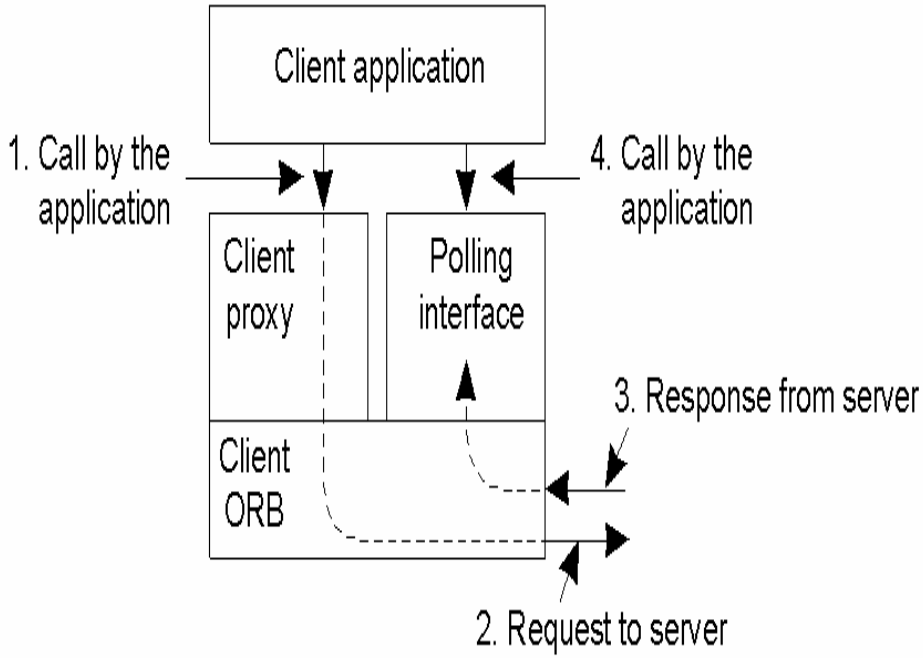
- Polling
  - The ORB provides an interface with polling operations
  - The called operation returns a *valuetype* object
  - This can be used for polling or waiting for the response
  - The polling operation is automatically generated by the client's ORB

```
int add(in int i, int j, out int k); ⇒
void sendpoll_add(in int i, in j,);  // Called by client
void replypoll_add(out int k);  // Called by client, impl. by ORB
```

Messaging (3)



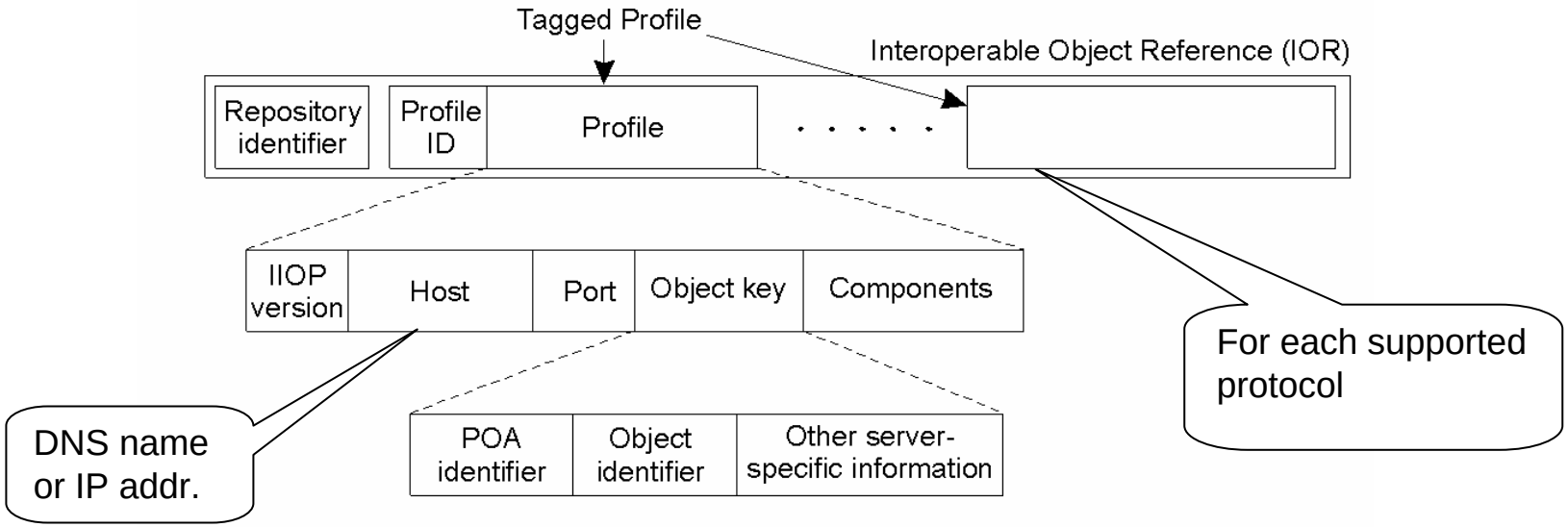
Callback



Polling

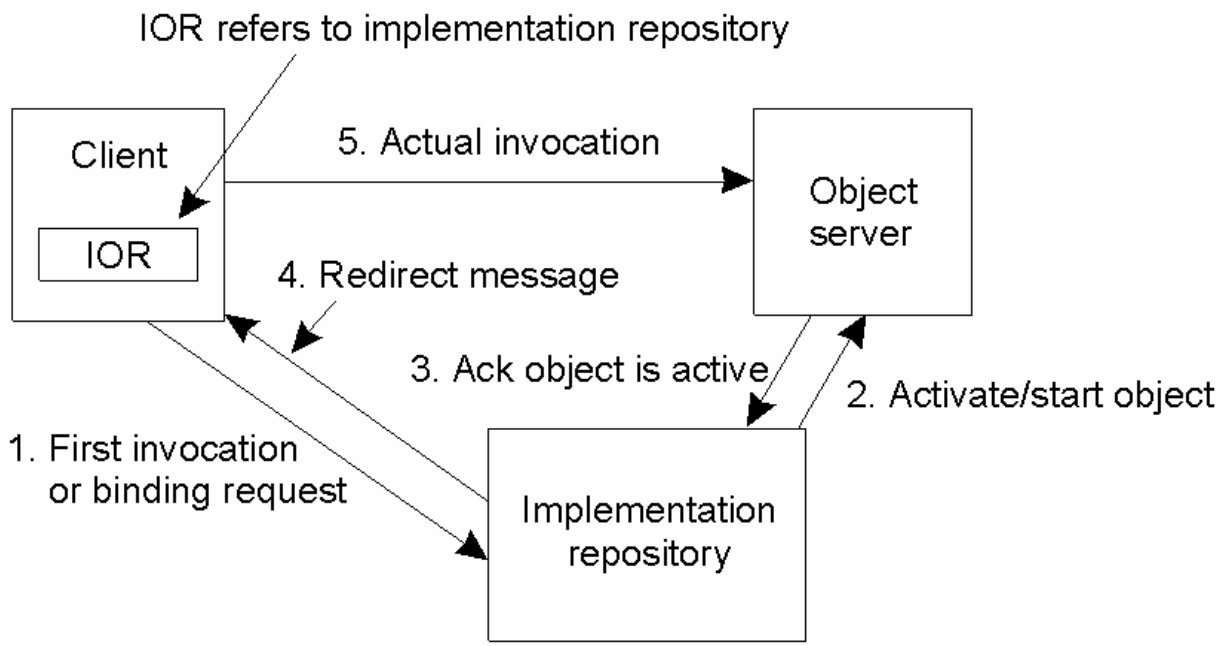
Naming (1)

- Interoperable Object Reference (IOR)
  - Provides valid references between (different) ORBs
- *Naming service* maps readable names to IORs
  - IOR structure:



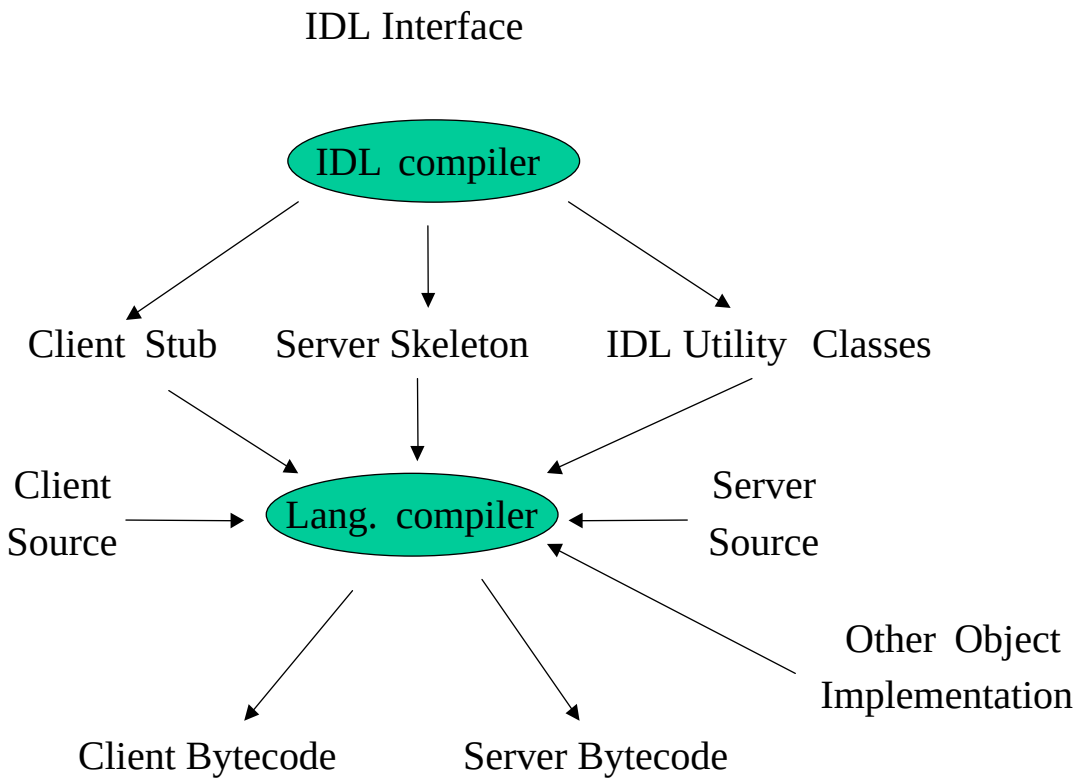
Naming (2)

- *Binding* maps IOR to servant
  - *Direct binding*: provides a pointer to the client proxy
  - *Indirect binding*: first a direct binding to the repository



Create a CORBA Application

- Define interfaces using IDL
- Implement the CORBA classes
- Develop the server program
- Develop the client program(s)
- Start the server and the client(s)



Example Application (1) – IDL Interface

- Arithmetic server
- “Adds” two arrays element by element

```
// Arithmetic IDL interface
module Arith {
        interface Add {
            const unsigned short SIZE = 10;
            typedef long array [SIZE];
            void sum_arrays(in array a, in array b, out array c);
        };
};
```

Example Application (2) – Generated packages

- Modules are mapped to packages (in Java)
- The IDL compiler generates a number of files, e.g.:
  - Add.java
    - The *Arith* interface declaration
  - AddOperations.java
    - Declares the *sum_arrays* method
  - _st_Add.java Stub
    - Code for the *Arith* object on the client side
  - _sk_Add.java Stub
    - Code for *Arith* object implementation on the server side
  - _example_Add.java
    - Code you can fill into implement the *Arith* object
  - Holder and helper classes
    - Hold *out* and *inout* parameters (instance variable *value*)

Example Application (3) – Generated Interface

```
/**
 * Generated by the idl2java compiler.
*/
public interface Add extends org.omg.CORBA.Object {
        final public static short SIZE = (short) 10;
        public void sum_arrays(
            int a[],
            int b[],
            Arith.AddPackage.arrayHolder c
            );
}
```

Example Application (4) - Implementation

```
public class AddImpl extends Arith._AddImplBase {
        /** Construct a persistently named object. */
        public AddImpl(java.lang.String name) { super(name); }
        /** Construct a transient object. */
        public AddImpl() { super(); }
        public void sum_arrays(
            int[] a, int[] b,
            Arith.AddPackage.arrayHolder c
        ) {
            c.value = new int[Arith.AddPackage.SIZE.value];
            for (int i = 0; i < Arith.AddPackage.SIZE.value; i++) {
                        // Own interface to hold the constant SIZE
            c.value[i] = a[i] + b[i];
}       }   }
```

- This can be compiled (javac AddImpl.java)

Example Application (5) - server class

```
public class Server {
public static void main(String argv[]) {
        try {
            ORB orb = ORB.init();              // Initialize the ORB
            BOA boa = orb.BOA_init();          // Initialize Basic Obejct Adapter

            // Create the AddImpl object and give it an external name
            AddImpl arr = new AddImpl("Arithmetic Server");

            // Export the newly created object.
            boa.obj_is_ready(arr);
            System.out.println(arr + " is ready. ");

            // Wait for incoming requests
            boa.impl_is_ready();
        } catch (SystemException se) { se.printStackTrace();  }   }    }
```

Example Application (6) - client class

```
public class Client {
public static void main(String argv[]) {
      int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
      int b[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
      Arith.AddPackage.arrayHolder result =
            new Arith.AddPackage.arrayHolder();
      try { ORB orb = ORB.init();    // Initialize the ORB
            // Locate an Add object via the external name
            Arith.Add add =
                Arith.AddHelper.bind(orb, “ArithmeticServer”);
            add.sum_arrays(a, b, result);
            System.out.print(“The sum is: “);
            for (int i = 0; i < Arith.AddPackage.SIZE.value; i++) {
                System.out.print(result.value[i]+" “);
            } System.out.println();
      } catch (SystemException se) { se.printStackTrace();   }    }     }
```

OMG IDL (1)

- No programming statements, purely *declarative*
- Identifiers, preprocessing comments as in C++
- Keywords all are lower case, other identifiers may not differ only in case, because of different mappings
- Modules serve for avoiding name clashes
  - Both a module and an interface introduce a new scope
  - Modules can contain interfaces and nested modules
- An interface name in the same scope can be used as a type name
- Interfaces in other name scopes must be qualified (::)
  - E.g. *RoomBooking::Room* is the full name of the following interface: *module RoomBooking { interface Room { . . . }; };*

OMG IDL (2) – Nested modules

```
module outer {
    module inner {  // nested module
            interface inside {};
    }; // module inner

    interface outside {  // can refer to inner as a local name
            inner::inside get_inside();
            // returns a reference of type ::outer::inner::inside
    }; // interface outside
}; // module outer
```

OMG IDL (3) – mutually referential IFs

- An interface type must be forward declared before use

```
interface A;        // forward declaration – no {}

interface B {       // B can use forward declared interface A
    A get_an_A();
};
interface A{
    B get_a_B();
};
```

OMG IDL (4) – re-opening of modules

```
module X {
        interface A;        // forward declaration of A
}; // close X to allow interfaces A needs to be declared
module Y {
        interface B; {      // B can use X::A as a type name
                X::A get_an_A();
        }; // B
}; // Y
module X {                  // re-open X
        interface C; {      // C can use A unqualified – same scope
                A get_an_A();
        }; // C
        interface A; {      // A can use Y::B as a type name
                Y::B get_a_B();
        }; // A
}; // X
```

OMG IDL (5) – Inheritance

- The *derived* interface inherits from the *base* interface. Syntax:
  - derived interface : base interface
- All interfaces implicitly inherit from *CORBA::Object*
- E.g. in Java-mapping an interface *A* maps to a Java interface *A*, which extends *org.omg.CORBA.Object* (provided by the ORB)
- Data types are also inherited
- Non-object types can be re-declared in derived interfaces.
  - Do not use this feature – it is a bug in the design!

OMG IDL (6) – Inheritance example

```
module InheritanceExample {
    interface A {
        typedef unsigned short ushort;
        ushort op1();
    }; // A

    interface B : A     {  // B extends A – by op2
        boolean op2(ushort num);
    }; // B

}; // InheritanceExample
```

OMG IDL (7) – Multiple Inheritance

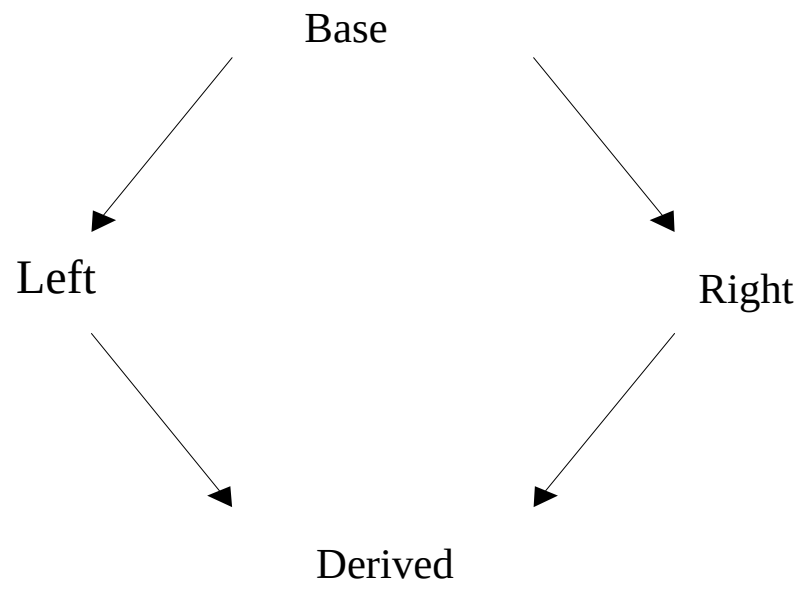
- An interface may inherit from several interfaces
- Syntax
  - The base interfaces separated by commas

interface C : A, B, Y::X { // extends A, B and Y::X . . . };

- The names of the operations in each of the inherited interfaces (including the operations they inherit from other interfaces)
  - must be unique and
  - must not be re-declared
  - exception: diamond inheritance

OMG IDL (8) – Diamond Inheritance

- Diamond Inheritance
  - The operations are inherited into two or more classes from the same base class, without ambiguity
  - The inheritance graph is the shape of a diamond



OMG IDL (9) – Diamond Example

```
module DiamondInheritanceExample{
        interface Base {
                string BaseOp();
        }; // Base
        interface Left : Base {
                short LeftOp(in long LeftParam);
        }; // Left
        interface Right : Base {
                any RightOp(in long RightParam);
        }; // Right
        interface Derived : Left, Right {
                octet DerivedOp(in float DerivedInParam,
                                out unsigned long DerivedOutParam);
        }; // Derived
}; // DiamondInheritanceExample
```

OMG IDL (10) – Types and Constants

- Basic Types
  - [unsigned] short, long, float, double, char, boolean, string, octet, enum, any
  - With *typedef* aliases can be created
  - For template types *typedef* is required (e.g. bounded strings):

```
interface StringProcessor {
    typedef string <8> octstring;          // max. 8 octets
    typedef string <100> centastring; // max. 100 octets
    . . .
}; // StringProcessor
```

OMG IDL (11) – Any Type

- A container that can contain a value of any IDL type
- It identifies the type of its contents for type safe extraction of the value
- The pseudo-ID TypeCode is used for identification
- It is left to each language mapping to define mechanisms for inserting and extracting values from Anys
- In Java it is mapped to *org.omg.CORBA.any*
- This provides insert/extract method pairs for predefined IDL types (e.g. *insert_long*, *extract_long* etc.)
- Helper classes generated by the IDL compiler may use them. E.g. for a type *usertype*, in class *usertypeHelper*:

  public static void insert (Any a, usertype t) { . . .}

  public static usertype extract (Any a) { . . .}

OMG IDL (12) – Structures

- Similar to C and C++ structures or
- Pascal / Modula records
- Example:

```
interface HardwareStore {
        struct window_spec {
                glass_color color;
                float height;
                float width;
} // HardwareStore ;
```

OMG IDL (13) – Discriminated unions

- Exactly one declaration branch will be selected depending on a switch value
- Storage is allocated for the largest case
- The value is undefined, if neither a legal switch value, nor a default is available

```
enum fitting_kind {door_k, window_k, shelf_k};

    union fitting switch (fitting_kind) {
    case door_k:        door_spec door;
    case window_k:      window_spec window;
    default:            float width;
};
```

OMG IDL (14) – Sequences

- Sequences are template types
- They can grow at run-time
- Their elements are accessed by an index
- Sequences may be bounded or unbounded
- Characterized by a maximum length and a current length
- Only the current number of elements are transmitted

```
typedef sequence <fitting> HWOrderSeq;
                                    // unbounded sequence of orders
typedef <fitting, 10> HWOrderSeq10;
                                    // sequence of max. 10 orders
typedef sequence <sequence <fitting>, 3> ThreeSeq;
typedef sequence <sequence <fitting> > ManySeq;
```

OMG IDL (15) – Arrays

- Fixed length, the entire array will be transmitted
- Arrays can be declared by the help of *typedef*, or as a part of a union or struct

```
typedef window [10] WindowVec10;        // 10 Elements
typedef fitting [3] [10] FittingGrid;   // 3*10 Elements
```

- Exceptions
  - Syntactically similar to structures
  - Standard exceptions + user-defined exceptions
  - Generic handler can test the actual arguments

```
exception OrderTooLarge {
    long max_items;
    long num_items_submitted;
};
```

- Constants

OMG IDL (16) – Operations

- Syntactically similar to C++ function prototypes
- Name, return type (or void), parameter list (maybe empty)
- Raises – specifies exceptions that may be raised
- Context – describes the caller's environment
  - Similar to Unix environment variables
  - Must be used with care – free interpretation
- Parameter passing mode: *in*, *out*, *inout*
- One-way operations
  - No result type, no exception, no out or inout parameters – just starts an action and hopes the best

    oneway void printAccount(in short customer_id);

Dynamic Invocation Interface - DII (1)

- Enables clients to invoke operations on an interface for which it has no compiled stub code
- It is less efficient, but more flexible
- Requests
  - A request has an object reference and a target operation name, associated with it
  - Operations to add arguments
  - Call the operation via *invoke* (blocks the caller)
- Deferred Synchronous Invocation
  - The *send* operation returns immediately
  - Eventual result can be got via *get_response*
- Using DII (use the *–portable* flag of the idl2java compiler)

Dynamic Invocation Interface - DII (2)

1. Obtain a reference for the target object

```
CORBA.Object object;
object = orb.string_to_object(ior);
// ior: Internet object reference: name of the object
```

2. Create a Request object for the request

```
CORBA.Request request = object._request("open");
```

3. Initialize the request parameters and the results

```
request.arguments().add_value("name",
        new CORBA.Any().from_string(acctName),
        CORBA.ARG_IN.value);
request.result.value().from_Object(null);
```

4. Invoke the request

```
request.invoke();
```

5. Get result

```
float balance = request.result().value().to_float();
```