# Implementing CORBA Applications

1. Introduction
2. CORBA Naming Service
3. Implementation: General Approach
5. C++ Implementations
6. Java Implementations
7. CORBA Trading Service

## 1. Introduction

-To enable distributed computing, CORBA specification addresses several key challenges such as *portability* and *interoperability*.

- CORBA server and client applications may be started on different machines with exactly the same results.
- Communication between the client and the server are handled transparently in a platform- and language-independent manner, provided there is a CORBA ORB available for the platform and programming languages involved.

-To achieve these functionality, CORBA uses standard mechanisms, services, and protocols. Some of the most essential of these artifacts are the ORB and CORBA Services such as Naming and Trading.

- The ORB acts as a mediator for any interaction between CORBA objects
- The Naming Service and the Trading services are some of the main mechanisms used by the ORB to locate objects in the network.

-CORBA application programming relies on these standard mechanisms and the high-level abstraction of the object interfaces captured in IDL.

## 2. CORBA Naming Service

### *Motivation*

•Object-oriented middleware uses object references to address server objects

-We need to find a way to get hold of these object references without assuming physical locations

•A name is a sequence of character strings that can be bound to an object reference

-A name binding can be resolved to obtain the object reference

•The CORBA ***Naming Service***:

- –Allows locating components by external names
- –Similar to white pages

• The CORBA ***Trading Service***:

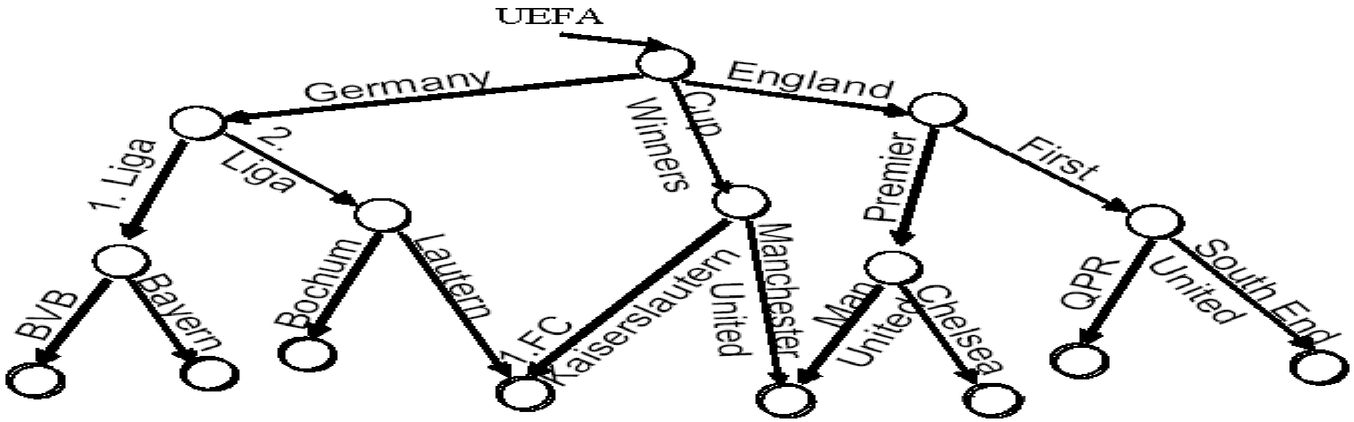
- –Locating components by service characteristics
- –Similar to yellow pages

## *Common Principles*

- There may be many server objects in a distributed object system
- Server objects may have several names
  - Leads to large number of name bindings
- Name space has to be arranged in a hierarchy to avoid
  - –Name clashes
  - –Poor performance when binding/resolving names
- Hierarchy achieved by naming contexts

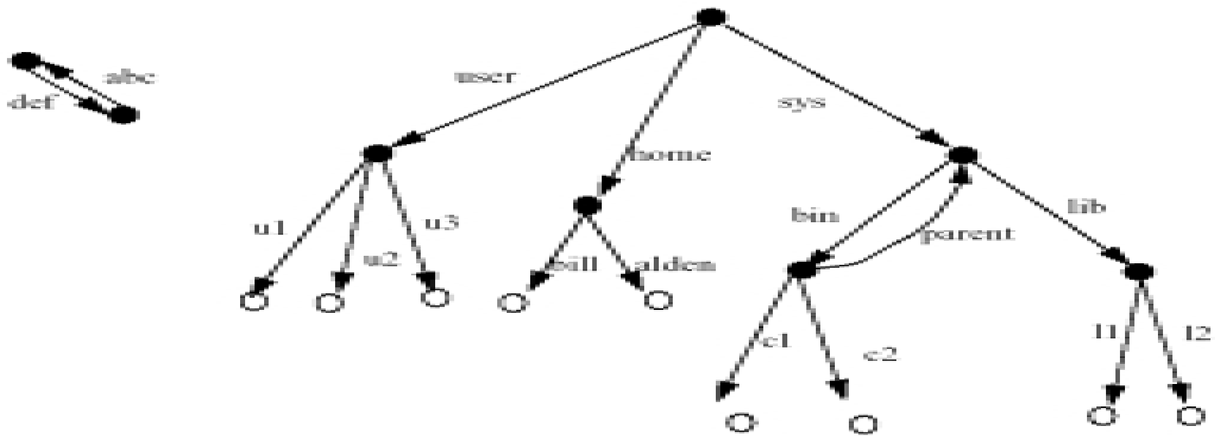
## *Naming Graph*

- Names are composed of possibly more than one component
- Used to describe traversals across several naming contexts



-Example of name components: ("UEFA","England", "Premier", "Chelsea")

## *Other Example of Naming Graphs*



•Name bindings are managed by name servers

-Not every server object needs a name

-Server objects may have several names (aliases)

-Name servers must store bindings persistently

-Name servers should be tuned towards efficiently resolving name bindings

-Name server may itself be distributed

## *CORBA Naming Service*

•Supports bindings of names to CORBA object references.

-Names are scoped in naming contexts.

-Multiple names can be defined for object references.

-Not all object references need names.

•Names are composed of simple names.

-Simple names are value-kind pairs.

-*Value* attribute is used for resolving names.

-*Kind* attribute is used to provide information about the role of the object.

```
//IDL type for name
module CosNaming {
            typedef string Istring;
            struct NameComponent {
                                    Istring id;
                                    Istring kind;
                        };
            typedef sequence <NameComponent> Name;
                        ...
};
```

•Naming Service is specified by two IDL interfaces:

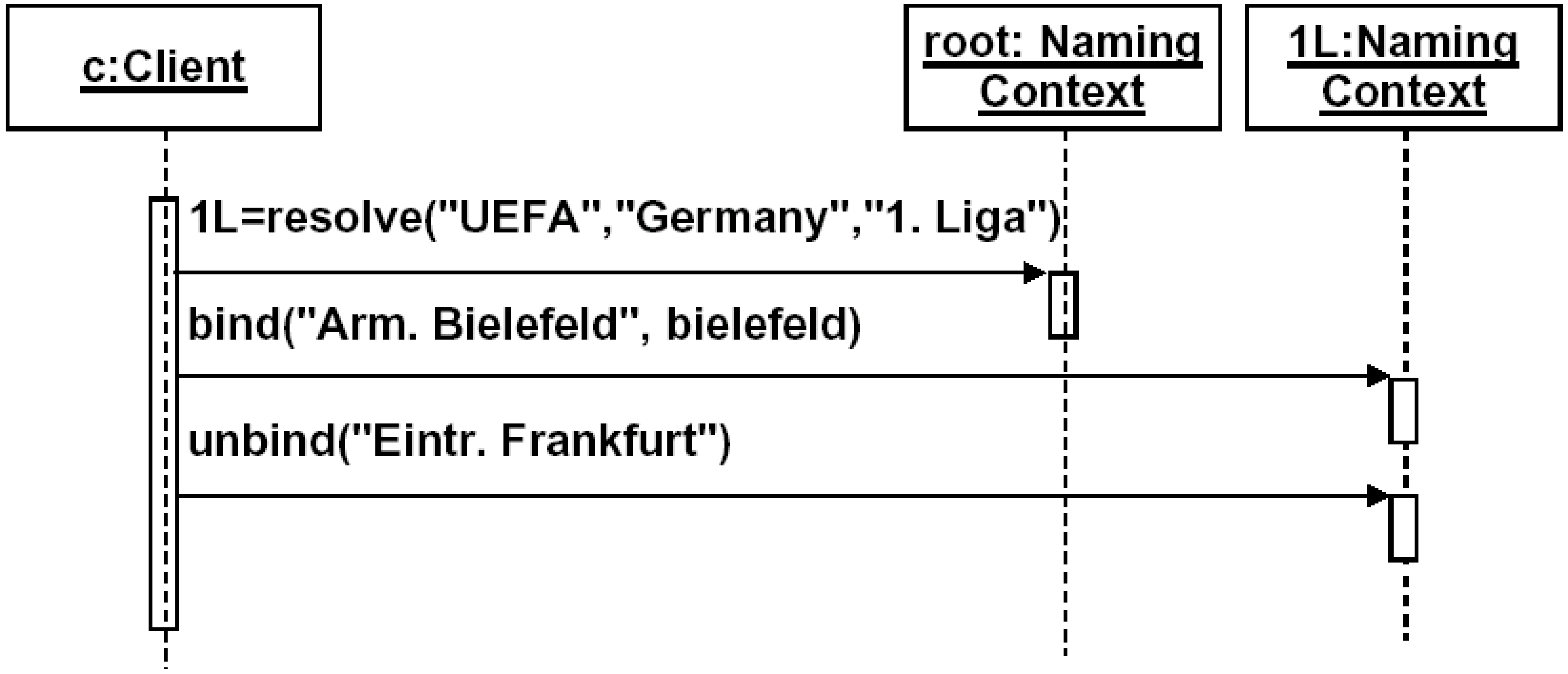
– ***NamingContext*** defines operations to bind objects to names and resolve name bindings.

– ***BindingIterator*** defines operations to iterate over a set of names defined in a naming context.

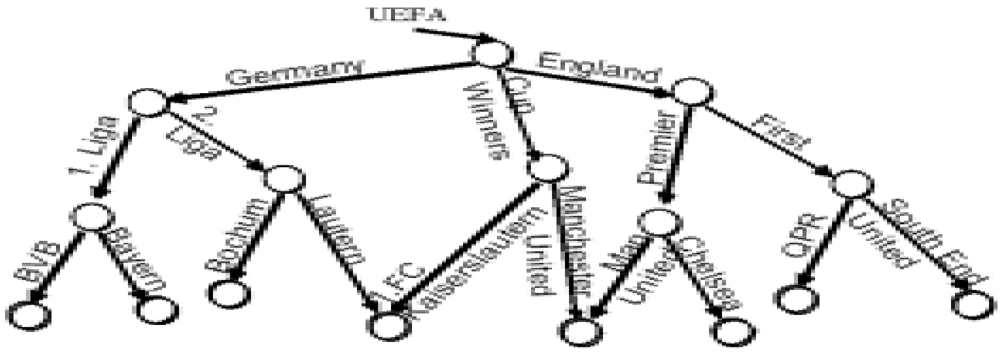
```
interface NamingContext {
    void bind(in Name n, in Object obj) raises (NotFound, ...);

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

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

    NamingContext new_context();
    NamingContext bind_new_context(in Name n) raises (NotFound, ...);

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

## *Example: Naming Scenario-Binding*

**Promote Bielefeld to German '1. Liga' and relegate Frankfurt**

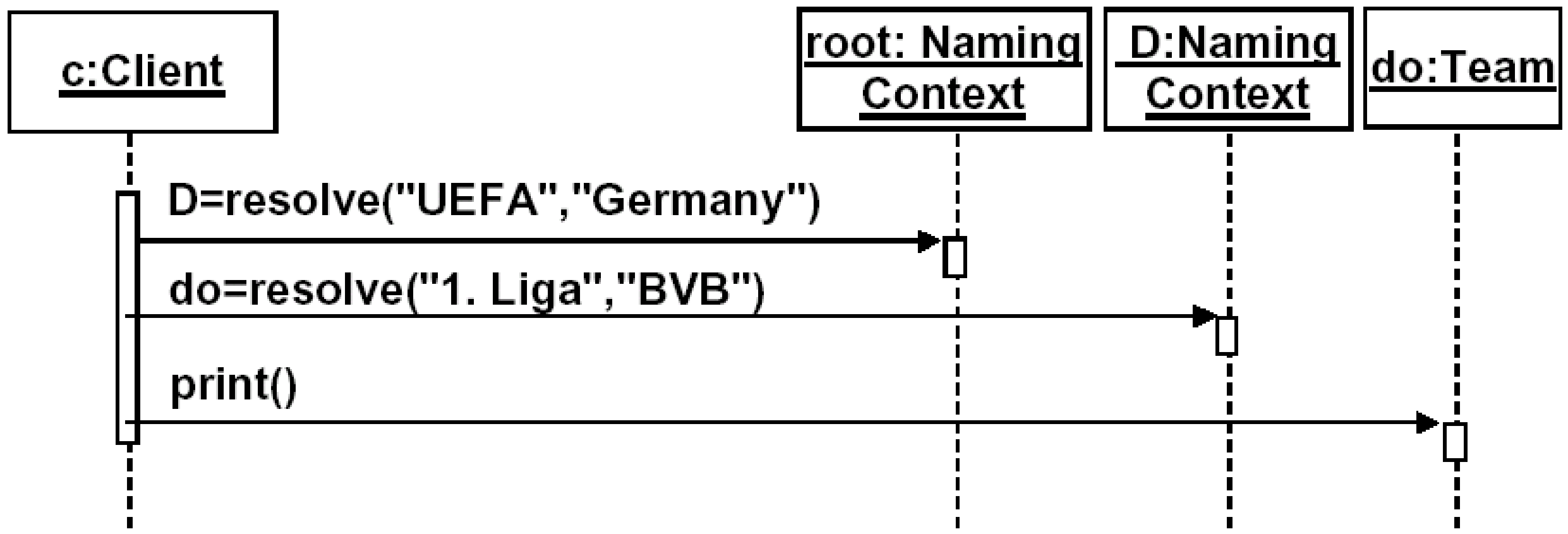


c:Client
root: Naming Context
1L:Naming Context
1L=resolve("UEFA","Germany","1. Liga")
bind("Arm. Bielefeld", bielefeld)
unbind("Eintr. Frankfurt")



## *Example: Naming Scenario-Resolving*

**Print squad of Borussia Dortmund**



## •Bootstrapping Naming Service

-How to get the Root Naming Context?

- •ORB Interface provides for initialization

```
module CORBA {
      interface ORB {
            typedef string ObjectId;
            typedef sequence <ObjectId> ObjectIdList;
            exception InvalidName{};
            ObjectIdList list_initial_services();
            Object resolve_initial_references
                  (in ObjectId identifier)
                              raises(InvalidName);
      }
}
```

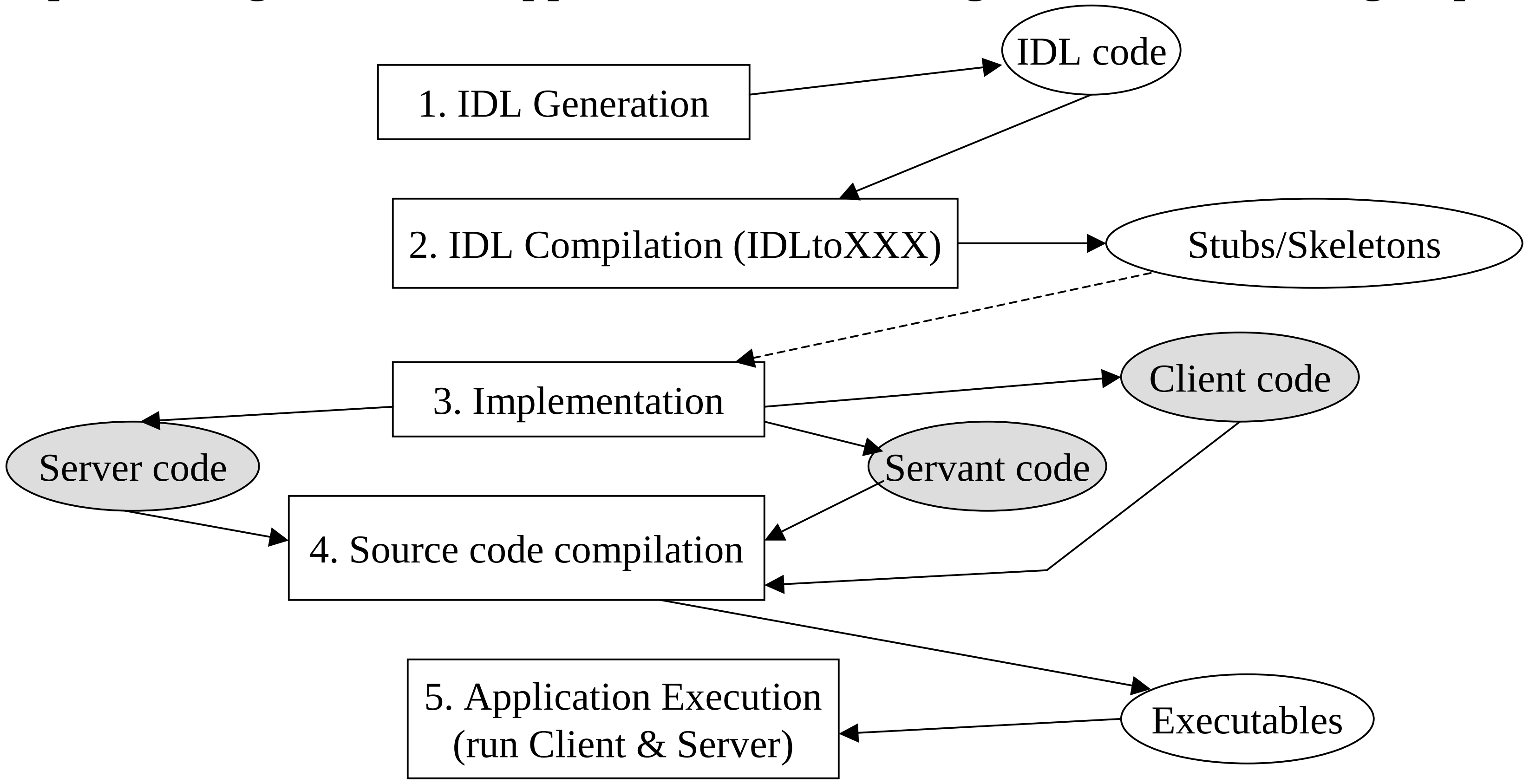
# *Example: Sample CORBA Server Code (Java)*

```
//get reference to  rootpoa & activate the POAManager
Object  poaRef = orb.resolve_initial_references("RootPOA");
POA rootpoa =  POAHelper.narrow(poaRef);
 rootpoa.the_POAManager().activate();

 //create servant (implementation object) and connect it with the ORB
   FunctionsImpl f= new FunctionsImpl();
   f.setORB(orb);

//get object reference from the servant
  Object ref = rootpoa.servant_to_reference(f);
 Functions fref = FunctionsHelper.narrow(ref);

 //get the root naming context; use NamingContextExt  instead of
 //NamingContext-is part of the interoperable naming service (INS).
  Object objRef= orb.resolve_initial_references("NameService");
  NamingContextExt ncRef=NamingContextExtHelper.narrow(objRef);

  //bind the Object Reference in Naming
  String name ="Calc";
  NameComponent path[] = ncRef.to_name(name);
  ncRef.rebind(path,fref);
```

```
//IDL code
module Calculator {
  interface Functions {
    float square_root (in float number);
    float power (in float base, in float exponent
    };
 };
```

# 3. Implementation: General Approach

-Implementing a CORBA application involves in general the following steps:

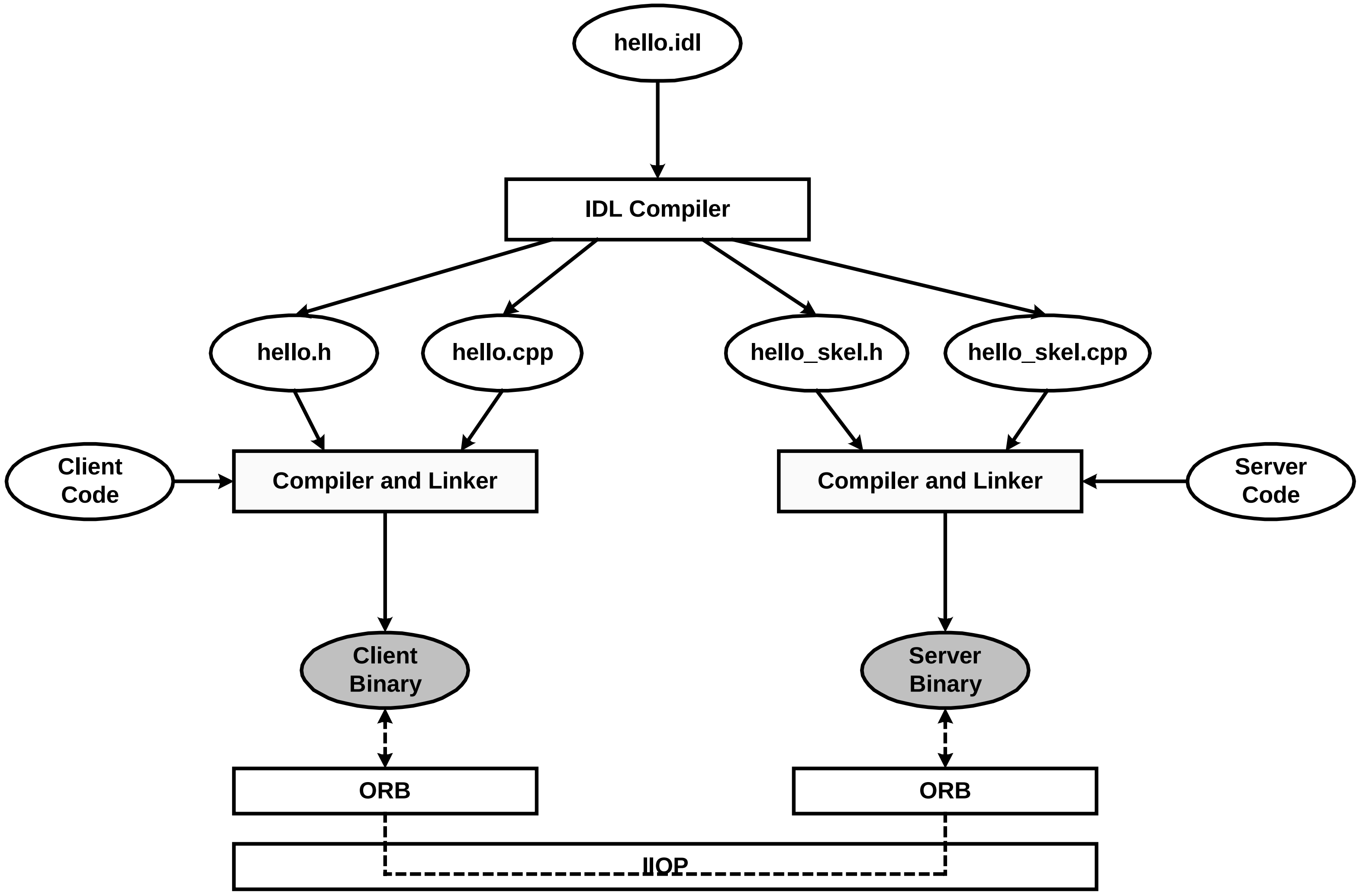


-The Servant corresponds to the remote CORBA object class

-The Client and Server may either be implemented in the same language or in different languages (e.g. C++ & Java). CORBA applications can interact, regardless of the language they are written in.

# 4. C++ Implementation

## *IDL to C++ Compiler Functionality*



## *Example of C++ Implementations*

### *IDL Code*

```
// Hello.idl
interface Hello {
        void say_hello();
};
```

### *Compiling the IDL*

```
> idl Hello.idl
```

This command will create the files:

- Hello.h : Header file containing Hello.idl ’s translated data types and interface stubs
- Hello.cpp: Source file containing Hello.idl ’s translated data types and interface stubs
- Hello_skel.h: Header file containing skeletons for Hello.idl ’s interfaces
- Hello_skel.cpp: Source file containing skeletons for Hello.idl ’s interfaces

## *Servant Definition and Implementation*

To implement the server, we need to define an implementation class for the Hello interface. To do this, we create a class Hello_impl that is derived from the “skeleton” class POA_Hello , defined in the file Hello_skel.h .

### *//Hello_impl.h : servant definition*

```
#include <Hello_skel.h> //contains the skeleton class POA_Hello

class Hello_impl : public POA_Hello, public PortableServer::RefCountServantBase {
                    public:
                        virtual void say_hello() throw(CORBA::SystemException);
};
```

### *//Hello_impl.cpp: servant implementation*

```
#include <iostream.h>
#include <OB/CORBA.h> //contains definitions for the standard CORBA classes
#include <Hello_impl.h>

void Hello_impl::say_hello() throw(CORBA::SystemException) {
          cout << "Hello World!" << endl;
}
```

## *Server Program*

To simplify exception handling and ORB destruction, we split the server into two functions: main() and run() , where main() only creates the ORB, and calls run()

*//Server.cpp*

```
#include <OB/CORBA.h>
#include <Hello_impl.h>
#include <fstream.h>
int run(CORBA::ORB_ptr);  // Forward declaration for the run() function.

int main(int argc, char* argv[]) {
        int status = EXIT_SUCCESS; // Exit status
        CORBA::ORB_var orb;
        try{ //initialize the ORB using the parameters with which the program was started.
                orb = CORBA::ORB_init(argc, argv);
                status = run(orb);}
        catch (const CORBA::Exception&) {status = EXIT_FAILURE;}

        if(!CORBA::is_nil(orb)) { //If the ORB was successfully created, destroy it to free resources.
                try {
                        orb -> destroy();
                }
                catch(const CORBA::Exception&) {status = EXIT_FAILURE; }
        }
        return status; //If there was no error, EXIT_SUCCESS is returned,or EXIT_FAILURE otherwise
}
```

## //Server.cpp (ctd.)

```
int run(CORBA::ORB_ptr orb) {

            //Use the ORB reference to obtain a reference to the Root POA.
            CORBA::Object_var poaObj = orb -> resolve_initial_references("RootPOA");
            PortableServer::POA_var rootPoa = PortableServer::POA::_narrow(poaObj);

             //Use the Root POA used to obtain a reference to its POA Manager.
            PortableServer::POAManager_var manager = rootPoa -> the_POAManager();


  /*Create and assign a servant object to a ServantBase_var variable. The servant is then used to incarnate
   a CORBA object, using the _this() operation. ServantBase_var and Hello_var , like all _var types, are
  “smart” pointer, i.e., servant and hello will release their assigned object automatically when they go
  out of scope.*/

             Hello_impl* helloImpl = new Hello_impl();
             PortableServer::ServantBase_var servant = helloImpl;
             Hello_var hello = helloImpl -> _this();
```

## *//Server.cpp-*int run(CORBA::ORB_ptr orb) (ctd.)

```
/*The client must be able to access the implementation object. This can be
done by saving a “stringified” object reference to a file, which can then be
read by the client and converted back to the actual object reference.*/

        CORBA::String_var s = orb -> object_to_string(hello);
        const char* refFile = "Hello.ref";
        ofstream out(refFile);
        out << s << endl;
        out.close();


/*The server must activate the POA Manager to allow the Root POA to start processing requests, and
  then inform the ORB that it is ready to accept requests.*/

        manager -> activate();
        orb -> run();

        return EXIT_SUCCESS;
}
```

## *Client Implementation*

Several segments of the client program are similar to the server program; for instance, the code to initialize and destroy the ORB is the same.

*//Client.cpp*

```
#include <OB/CORBA.h>
#include <Hello.h> //In contrast to the server, the client does not need to include Hello_impl.h .
                   //Only the generated file Hello.h is needed.
#include <fstream.h>

int run(CORBA::ORB_ptr);

int main(int argc, char* argv[]) {
          ... // Same as for the server
}
```

## *//Client.cpp (ctd.)*

```
int run(CORBA::ORB_ptr orb) {

/* The “stringified” object reference written by the server is read and converted to a CORBA::Object
   object reference. It’s not necessary to obtain a reference to the Root POA or its POA Manager,
   because they are only needed by server applications.*/

            const char* refFile = "Hello.ref";
            ifstream in(refFile);
            char s[2048];
            in >> s;
            CORBA::Object_var obj = orb -> string_to_object(s);

           //Generates a Hello object reference from theCORBA::Object object reference.
            Hello_var hello = Hello::_narrow(obj);

           //Invoke the say_hello operation on the hello object reference
            hello -> say_hello();

            return 0;
}
```

## *Compiling and Linking*

-Compiling Hello.cpp results in an object file with the following name:
• **UNI**X: Hello.o
• **Windo**ws: Hello.obj
-You must link both the client and the server with the file for your platform. The compiled Hello_skel.cpp and Hello_impl.cpp are only needed by the server.

-In practice, compiling and linking is compiler- and platform-dependent. Many compilers require unique options to generate correct code.

-To build Orbacus programs, you must at least link with the Orbacus library for your platform:
• **UNIX**: libOB.a
• **Windo**ws: ob.lib

## *Running the Application*

-The “Hello World!” application consists of two parts:
• The client program
• The server program
-Start the server first, since it must create the file Hello.ref that the client needs in order to connect to the server. As soon as the server is running, you can start the client.

# 6. Java Implementations

## *Compiling the IDL*

```
jidl --package hello Hello.idl
```

For every construct in the IDL file that maps to a Java class or interface, a separate class file is generated. Directories are automatically created for those IDL constructs that map to a Java package (e.g., a module ).

This command generates several Java source files on which the actual implementation will be based:

- Hello.java
- HelloHelper.java
- HelloHolder.java
- HelloOperations.java
- HelloPOA.java
- _HelloStub.java

## ***Servant Class***

The servant class Hello_impl must inherit from the generated class HelloPOA .

```
// Hello_impl.java

package hello;

public class Hello_impl extends HelloPOA {
        public void say_hello()  {
                System.out.println("Hello World!");
        }
}
```

## ***Server Class***

The Server class holds the server's main() and run() methods:

```
// Server.java
package hello;

public class Server {
        public static void main(String args[]) {
                //Properties  necessary to use the Orbacus ORB instead of the JDK’s ORB.
                java.util.Properties props = System.getProperties();
                props.put("org.omg.CORBA.ORBClass",  "com.ooc.OBServer.ORB");
                props.put("org.omg.CORBA.ORBSingletonClass","com.ooc.CORBA.ORBSingleton");
```

*//Server.java- main() (ctd.)*

```
int status = 0; //exit status
org.omg.CORBA.ORB orb = null;
try {
        //initialize the ORB
        orb = org.omg.CORBA.ORB.init(args, props);
        status = run(orb);
}
catch(Exception ex) { ex.printStackTrace(); 25 status = 1; }




if(orb != null) { //If the ORB was successfully created, destroy it to free resources.
        try { orb.destroy();  }
        catch(Exception ex) {
                ex.printStackTrace();

//The exit status is returned. If there was no error, 0 is returned, or 1otherwise.
                status = 1;
        }
}
```

}

## *//Server.java (ctd.)*

```
      static int run(org.omg.CORBA.ORB orb) throws org.omg.CORBA.UserException {
       //Get a reference to the Root POA, and use it to obtain a reference to its POA Manager.
         org.omg.PortableServer.POA rootPOA =
              org.omg.PortableServer.POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         org.omg.PortableServer.POAManager manager =  rootPOA.the_POAManager();

  //Create a servant of type Hello_impl, which is used to incarnate a CORBA object.
        Hello_impl helloImpl = new Hello_impl();
        Hello hello = helloImpl._this(orb);

        try { //The object reference is “stringified” and written to a file.
               String ref = orb.object_to_string(hello);
               String refFile = "Hello.ref";
               java.io.PrintWriter out = new java.io.PrintWriter(new java.io.FileOutputStream(refFile));
               out.println(ref);
               out.close();
            } catch(java.io.IOException ex) {
                 ex.printStackTrace();
                 return 1;
            }

//The server enters its event loop to receive incoming requests.
       manager.activate();
       orb.run();
       return 0;
      }
} //end Sever.java
```

## *Client Implementation*

*//Client.java*

```
package hello;

public class Client {
        public static void main(String args[]) {
                ... // Same as for the server
        }

        static int run(org.omg.CORBA.ORB orb) {
                org.omg.CORBA.Object obj = null;

                try { //The stringified object reference is read and converted to an object.
                        String refFile = "Hello.ref";
                        java.io.BufferedReader in =
                                new java.io.BufferedReader( new java.io.FileReader(refFile));
                        String ref = in.readLine();
                        obj = orb.string_to_object(ref);
                } catch(java.io.IOException ex) {
                                        ex.printStackTrace();
                                        return 1;
                                        }
                Hello hello = HelloHelper.narrow(obj); //“Narrowed” to a reference to a Hello object.
                hello.say_hello(); //Invoke the say_hello method
                return 0;
        }
} //end Client.java
```

## *Compiling the Application*

-Ensure that your CLASSPATH environment variable includes the current working directory as well as the Orbacus for Java classes (i.e the OB.jar file):

- **UNIX**: CLASSPATH=.:your_orbacus_directory/lib/OB.jar:$CLASSPATH
  export CLASSPATH
- **Windows:** set CLASSPATH=.;your_orbacus_directory\lib\OBE.jar;%CLASSPATH%

-Use ***javac*** to compile the implementation classes and the classes generated by the Orbacus IDL-to-Java translator.

***javac hello/*.java***

## *Running the Application*

1. Start the 'Hello World' Java server: ***java hello.Server***
2. Start the 'Hello World' Java client: ***java hello.Client***

**Note:** make sure that your CLASSPATH environment variable includes the OBE.jar file.
You may use a C++ server together with a Java client (or vice versa).

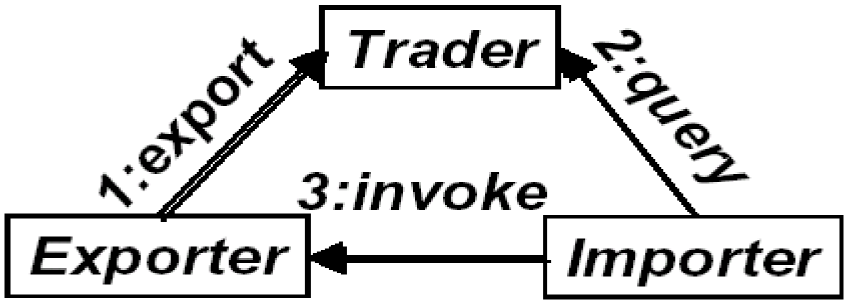
## 7. CORBA Trading Service

### *Rationale*

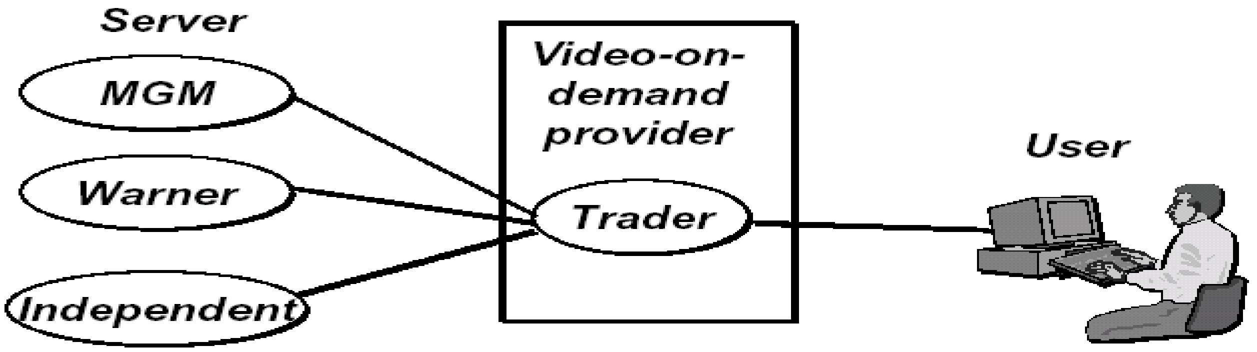
- Locating objects in location transparent way
- Naming simple but may not be suitable when
  - clients do not know server
  - there are multiple servers to choose from
- Trading supports locating servers based on service functionality and quality
- Naming ⇔ White pages
- Trading ⇔Yellow Pages

## *Trading Operation*

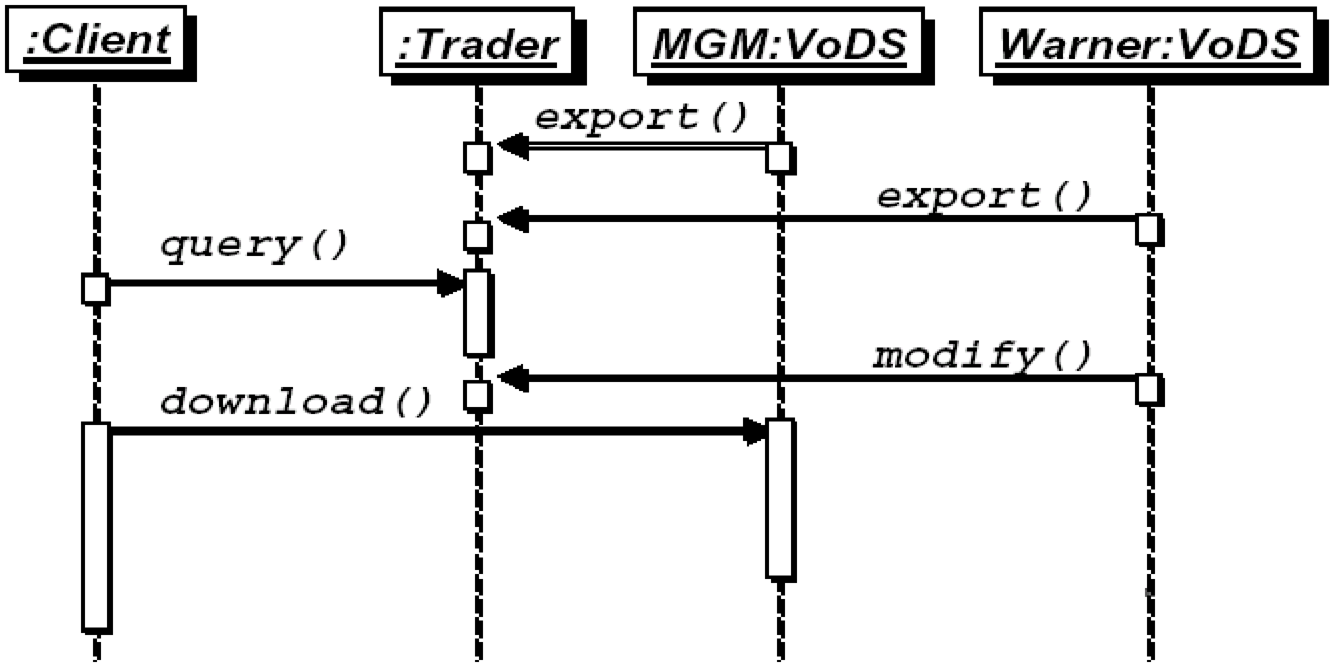
- Trader operates as broker between client and server.
- Enables client to change perspective from ´who?´ to ´what?´
- Similar ideas in:
  - mortgage broker
  - insurance broker
- Clients ask trader for
  - a service of a certain type
  - at a certain level of quality
- Trader supports
  - service matching
  - service shopping
- Server registers service with trader.
- Server defines assured quality of service:
  - Static QoS definition
  - Dynamic QoS definition



## *Example: Hongkong Telecom video-on-demand*



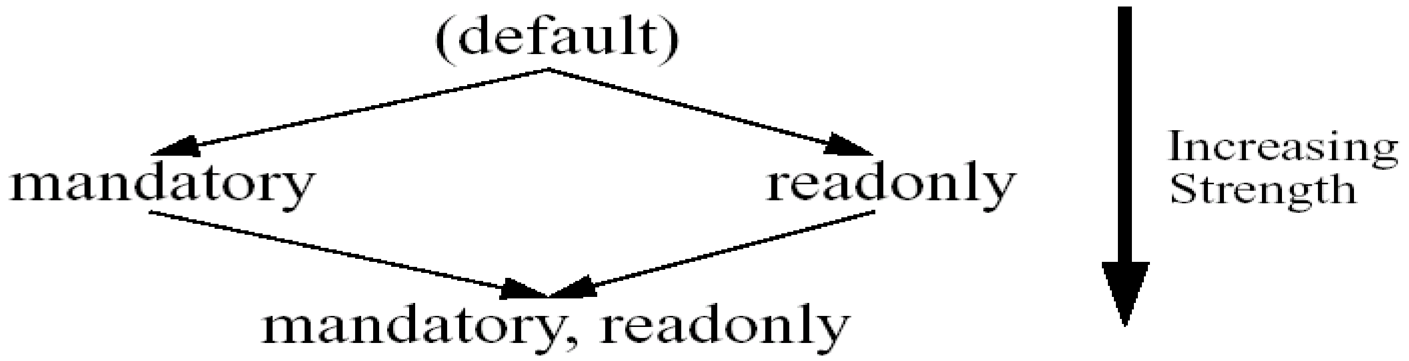
*The Trading Process*



## *Service Type Definition*

- Service types define
  - Functionality provided by a service and
  - Qualities of Service (QoS) provision.
- Functionality defined by object type
- QoS defined based on properties, i.e.
  - property name
  - property type
  - property value
  - property mode
    - mandatory/optional
    - readonly/modifiable

### *Property Strength*



## *Service Type Example*

```
typedef enum {VGA,SVGA,XGA} Resolution;
service video_on_demand {
        interface VideoServer;
        readonly mandatory property float fee;
        readonly mandatory property Resolution res;
        modifiable optional property float bandwidth;
}
```

## *Service Type Hierarchy*

- An object type might have several implementations with different QoS
- Same object type might be used in different service types.
- Service type S is subtype of service S’ iff
  - object type of S is identical or subtype of object type of S’
  - S has at least all properties defined for S’
- Subtype relationship can be exploited by trader for service matching purposes

## *Constraint Definition*

- Importer defines the desired qualities of service as part of the query:
  - *Example:*

```
fee<10 AND res >=SGA AND bandwidth>=256
```

- In a query, trader matches only those offers that meet the constraint

## *Federated Traders*

- Scalability demands federation of traders
- A trader participating in a federation
  - – offers the services it knows about to other traders
  - – forwards queries it cannot satisfy to other traders
- Problems
  - – Non-termination of import
  - – Duplication of matched offers

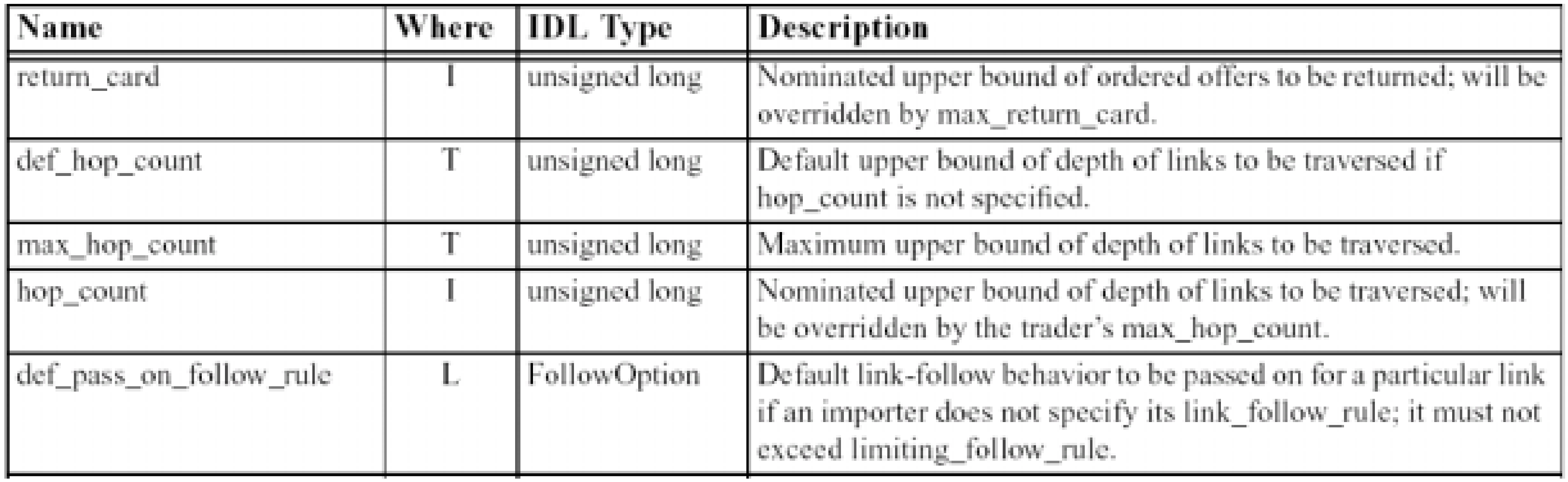
## *Trading Policies*

- Depending on constraint and available services, a large set of offer might be returned by a query.
- Trading policies are used to restrict the size of the matched offers
  - – Specification of an upper limit
  - – Restriction on service replacements
  - – Restriction on modifiable properties (these might change between match making and service requests)
- Policies provide information to affect trader behavior at run time.

- Policies are represented as name value pairs.

```
typedef string PolicyName; // policy names restricted to Latin1
typedef sequence<PolicyName> PolicyNameSeq;
typedef any PolicyValue;
struct Policy {
        PolicyName name;
        PolicyValue value;
};
typedef sequence<Policy> PolicySeq;
```

## *Standardized Scoping Policies*



| Name | Where | IDL Type | Description |
|---|---|---|---|
| return_card | I | unsigned long | Nominated upper bound of ordered offers to be returned; will be overridden by max_return_card. |
| def_hop_count | T | unsigned long | Default upper bound of depth of links to be traversed if hop_count is not specified. |
| max_hop_count | T | unsigned long | Maximum upper bound of depth of links to be traversed. |
| hop_count | I | unsigned long | Nominated upper bound of depth of links to be traversed; will be overridden by the trader's max_hop_count. |
| def_pass_on_follow_rule | L | FollowOption | Default link-follow behavior to be passed on for a particular link if an importer does not specify its link_follow_rule; it must not exceed limiting_follow_rule. |

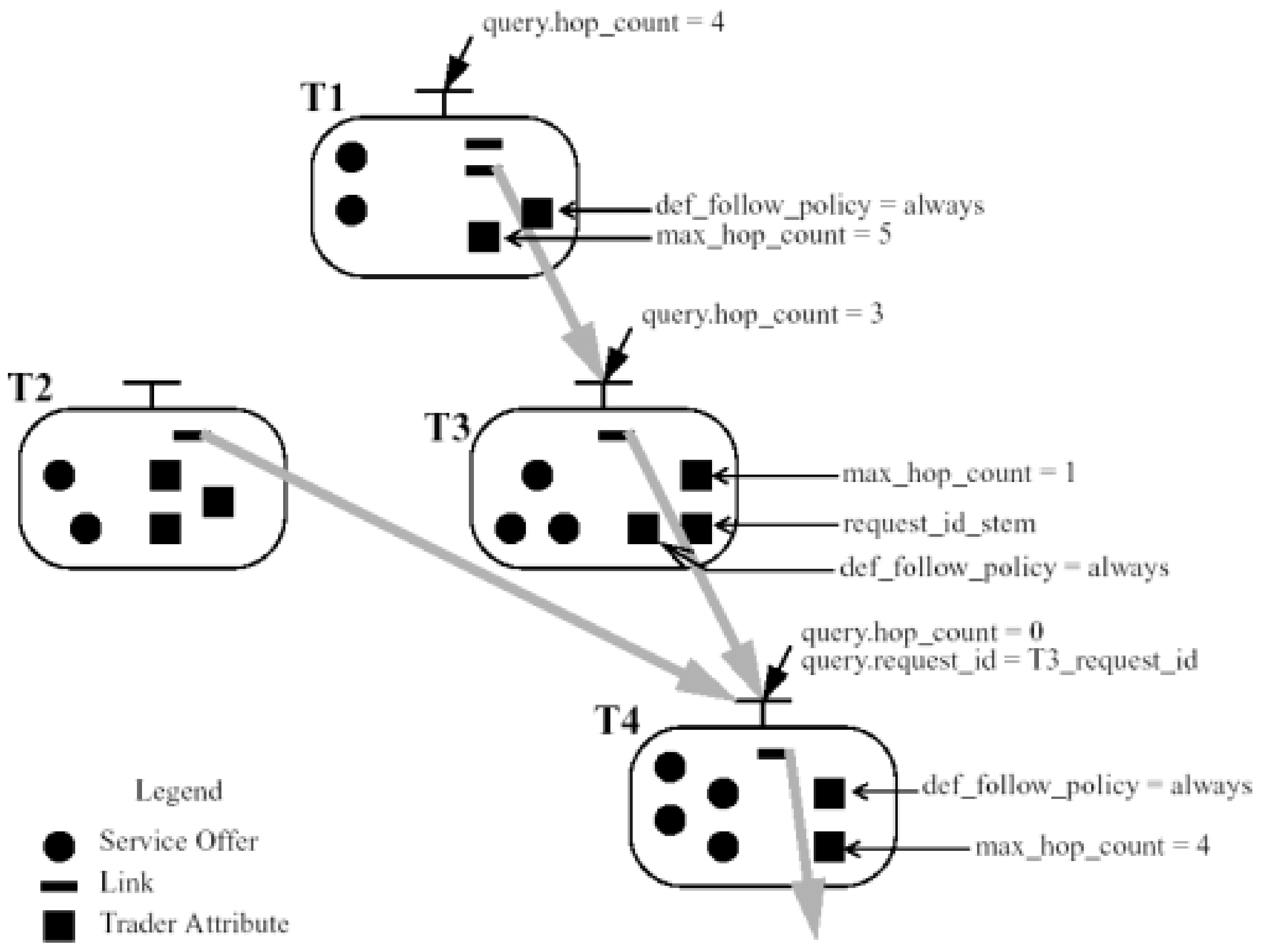
Different policies are associated with different roles in the performance of the tradingb function.
These roles are:

T = Trader

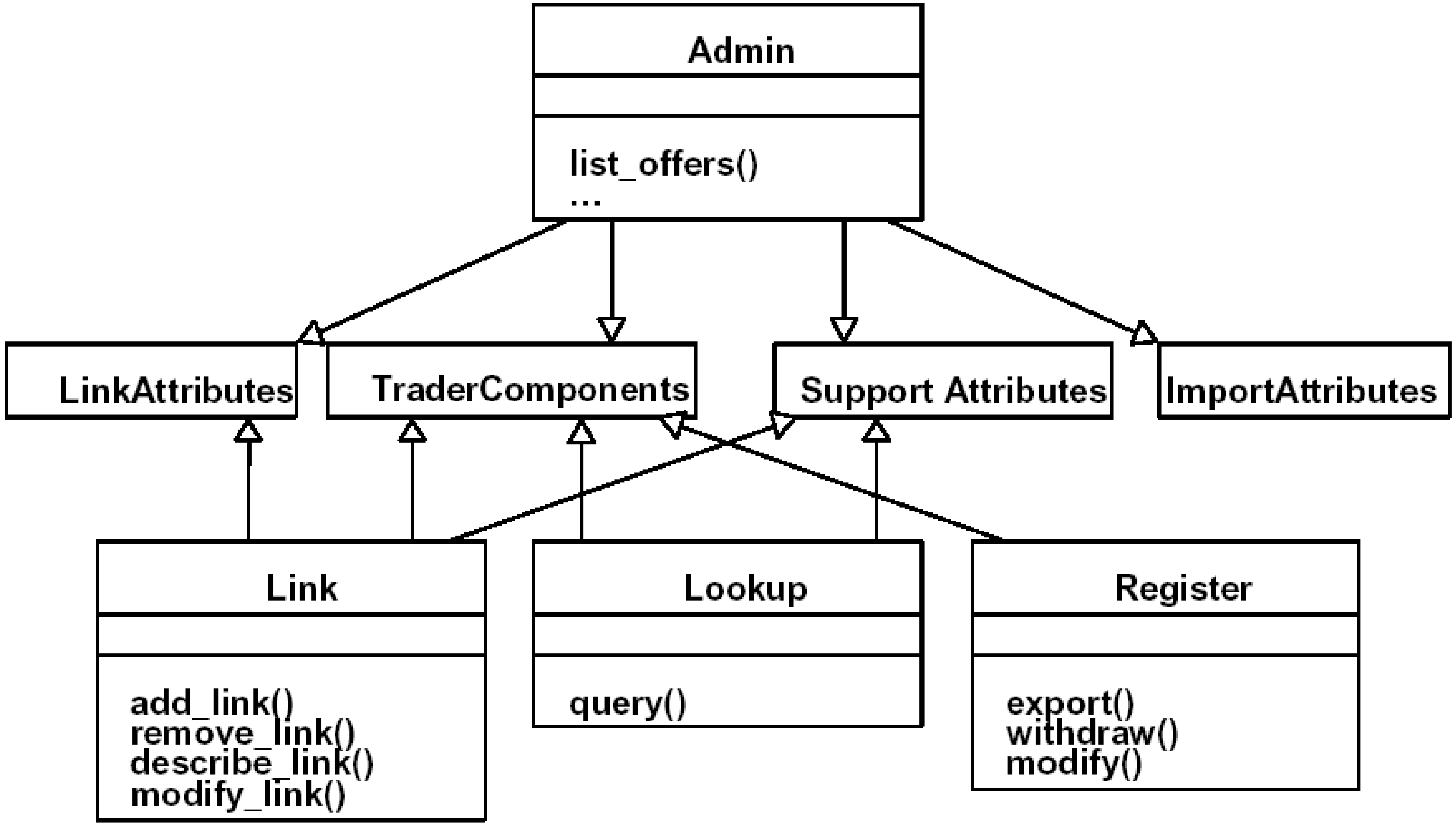
L = Link

I = Import

## *Example: Flow of a Query through a Trading Graph*



# *CORBA Trading Interfaces*



## *Defining Quality of Service*

```
typedef Istring PropertyName;
typedef sequence<PropertyName> PropertyNameSeq;
typedef any PropertyValue;
struct Property {
                 PropertyName name;
                 PropertyValue value;
};
typedef sequence<Property> PropertySeq;
enum HowManyProps {none, some, all}
union SpecifiedProps switch (HowManyProps) {
        case some : PropertyNameSeq prop_names;
};
```

Properties are <name, value> pairs. An exporter asserts values for properties of the service it is advertising. An importer can obtain these values about a service and constrain its search for appropriate offers based on the property values associated with such offers.

## *Trader Interface for Exporters*

```
interface Register {
OfferId export(in Object reference,
                 in ServiceTypeName type,
                 in PropertySeq properties) raises(...);
OfferId withdraw(in OfferId id) raises(...);
void modify(in OfferId id,
                         in PropertyNameSeq del_list,
                         in PropertySeq modify_list) raises (...);
};
```

## *Service Offers*

•A service offer is the information asserted by an exporter about the service it is advertising. It contains:

–the service type name,

–a reference to the interface that provides the service, and

–zero or more property values for the service.

•An exporter must specify a value for all mandatory properties specified in the associated service type.

- In addition, an exporter can nominate values for named properties that are not specified in the service type. In such case, the trader is not obliged to do property type checking.

```
struct Offer {
                        Object reference;
                        PropertySeq properties;
};
typedef sequence<Offer> OfferSeq;

struct OfferInfo {
                        Object reference;
                        ServiceTypeName type;
                        PropertySeq properties;
};
```

## *Trader Interface for Importers*

```
interface Lookup {
void query(      in ServiceTypeName type,
                         in Constraint const,
                         in Preference pref,
                         in PolicySeq policies,
                         in SpecifiedProps desired_props,
                         in unsigned long how_many,
                         out OfferSeq offers,
                         out OfferIterator offer_itr,
                         out PolicyNameSeq Limits_applied)
raises (IllegalServiceType, UnknownServiceType, IllegalConstraint, IllegalPreference,IllegalPolicyName, PolicyTypeMismatch,
        InvalidPolicyValue, IllegalPropertyName, DuplicatePropertyName, DuplicatePolicyName );
};
```