The UNIVERSITY *of* EDINBURGH
SCHOOL *of* COMPUTER SCIENCE

CS4/MSc

Distributed Systems

Jane Hillston

jeh@dcs.ed.ac.uk

Room 1604

(Lecture 7, CORBA in detail, January 28th 2002)

CORBA Components

In this lecture we consider the logical components of the CORBA standard.

ORB core Represents the *communication modules* of the generic architecture presented in Lecture 4, and basic environment.

Object adapter Looks after the CORBA objects, linking implementations to object references.

Implementation repository A source of servers — it keeps a mapping of object adapters to implementations and can activate servers if necessary to provide instantiated implementations.

Interface repository Keeps a register of IDL interfaces and can supply details of the registered objects.

Dynamic invocation interface Allows clients to construct a remote invocation request when an appropriate proxy is not available.

Dynamic skeleton interface Allows servers to accept invocations on CORBA objects without specific skeletons.

Object Adapters

In many ORB implementations servants interact with the ORB via an **object adapter**. It includes the roles of *remote reference module* and *dispatcher* in the generic model of Lecture 4. It has the following tasks:

- it generates and interprets object references;
- it knows where to locate or how to instantiate a servant for each object in its domain;
- it also manages the run-time environment of the servants.

The object adapter gives each CORBA object a unique **object ID** which forms part of its remote object reference; this incorporates the object adapter's own name. Once registered with the object adapter the object is maintained in a table known as the **active object map**.

Originally CORBA specified the **Basic Object Adapter (BOA)**. Later the **Portable Object Adapter (POA)** was introduced to eliminate inconsistencies which had appeared in BOA implementations.

Basic Object Adapter

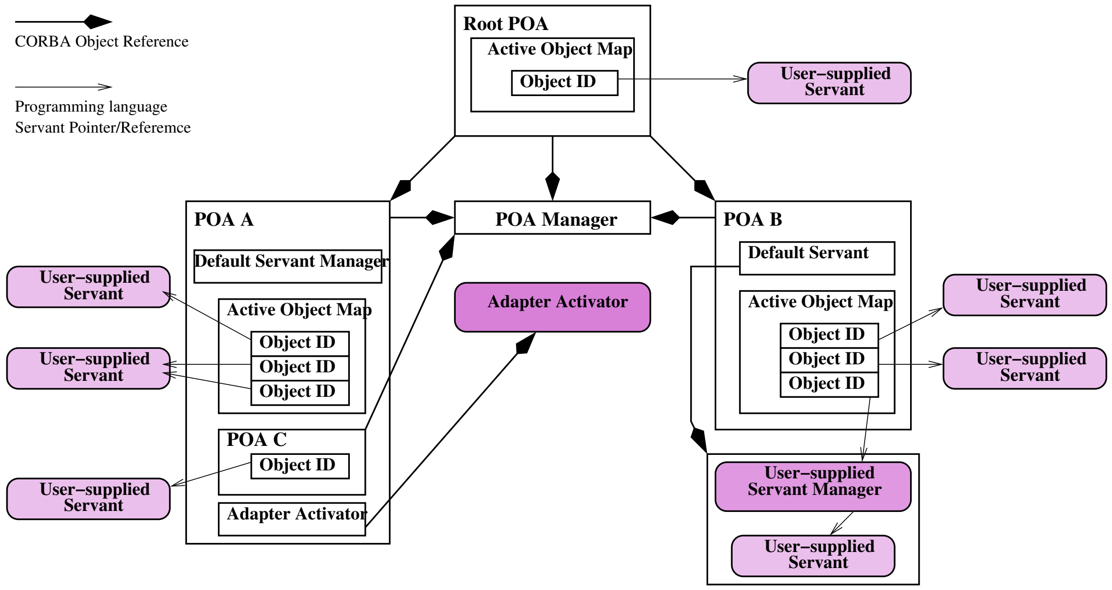
There is no explicit BOA in the JDK ORB, but when it is supported:

```
try{
  ORB orb = ORB.init(args, null);
  org.omg.CORBA.BOA boa = orb.BOA_init();

  HelloServant helloRef = new HelloServant();
  boa.obj_is_ready(helloRef);

  org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
    ............
  ncRef.rebind(path, helloRef);

  boa.impl_is_ready();
}
```

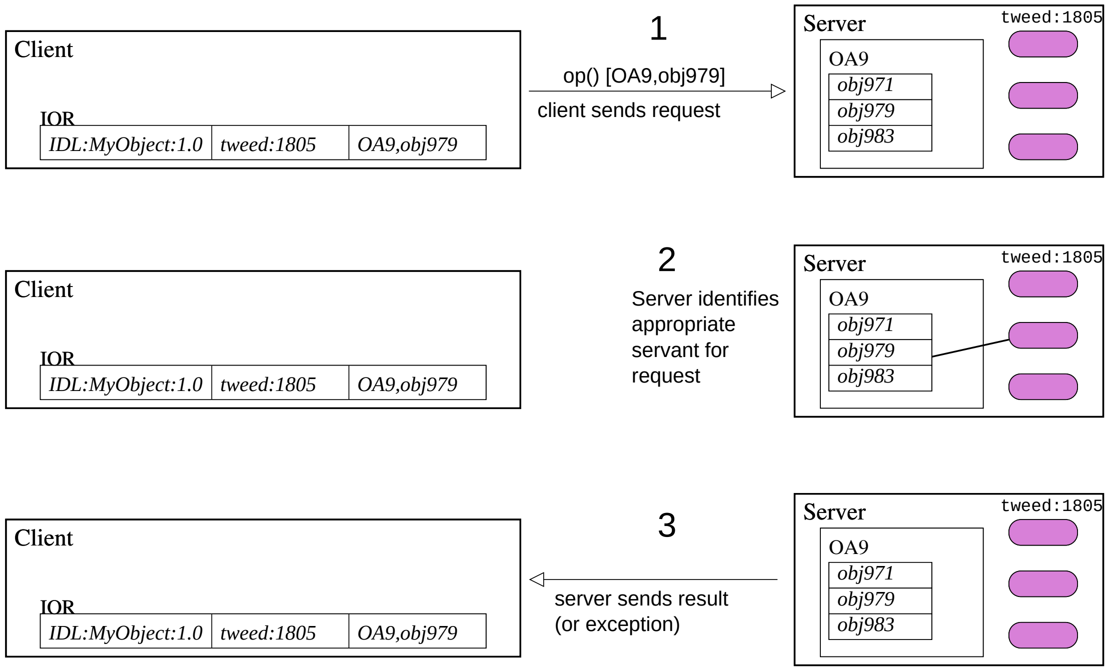
POA Architecture(1)

- POA support different policies, for example with respect to what to do if a request arrives for a servant which is not currently running. For this reason more than one POA may be running on each host—each POA can only support one set of policies.
- The lifetime of a servant is regarded as a distinct from the lifetime of an object: over the lifetime of an object it may be represented by several servants, instantiated in different servers.
- The role of the POA is to ensure that a servant is matched to an object when a method is invoked. This may mean starting a server.
- The action of providing a running servant to serve requests on a particular object ID is termed **incarnation**.
- The action of breaking the association between a servant and an object ID is termed **etherealization**.
- A POA may have a **default servant** to which all incoming requests for object IDs not in the Active Object Map are dispatched.

POA Architecture(2)



Binding a transient IOR



Client as Applet: `HelloApplet.java`

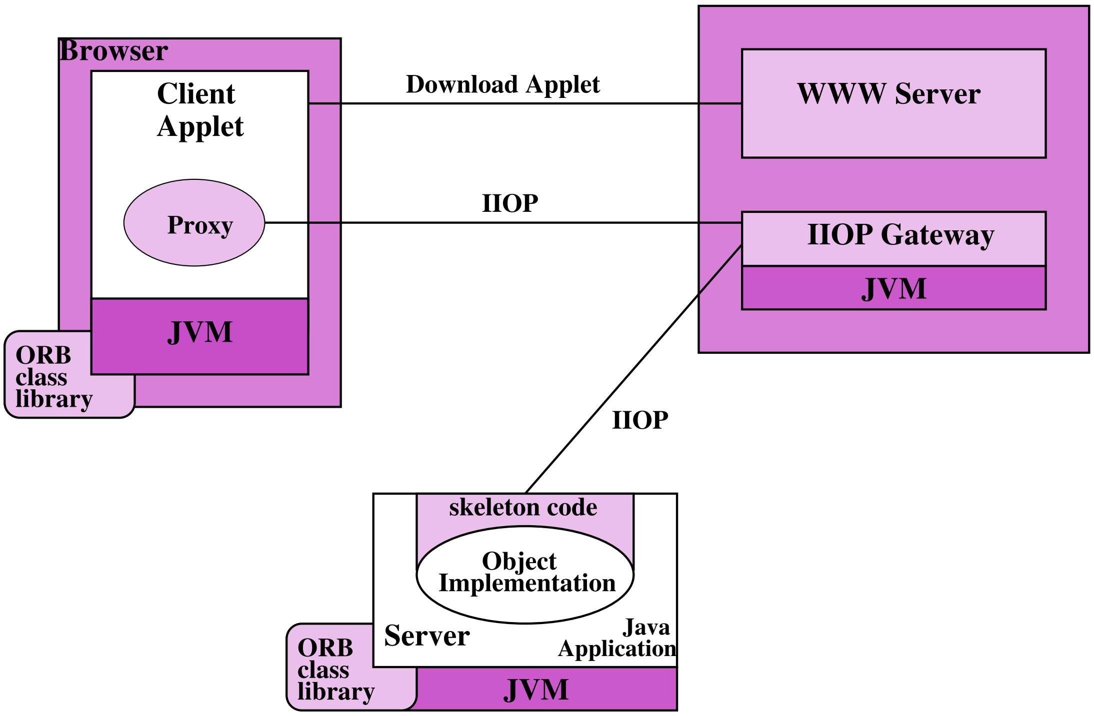
The major difference of which the programmer needs to be aware of when the client is an applet is that the way in which the orb is initialised is different:

```
public void init() {
  try {
    // set properties to ensure the correct ORB is initialised
    Properties props = new Properties();
    props.put("org.omg.CORBA.ORBClass", "com.sun.CORBA.iiop.ORB");

    // create and initialize the ORB
    ORB orb = ORB.init(this, props);
    ........
```

Sandboxing Implications

- Sandboxing prevents an unsigned applet from gaining access to local resources, in particular networking capabilities.
- Network calls are limited to using connections with the host from which the applet was downloaded — this is in conflict with CORBA *location transparency*. The unsigned applet can only invoke operations on objects that are local or resident on its host of origin.
- The problem is overcome by an **IIOP gateway**. This is a process running on the applet's host.
- The client's stub code sends all its remote requests to the IIOP gateway which is then responsible for forwarding the requests to the target object.
- Similarly any response to the invocation is routed back to the client via the IIOP gateway.

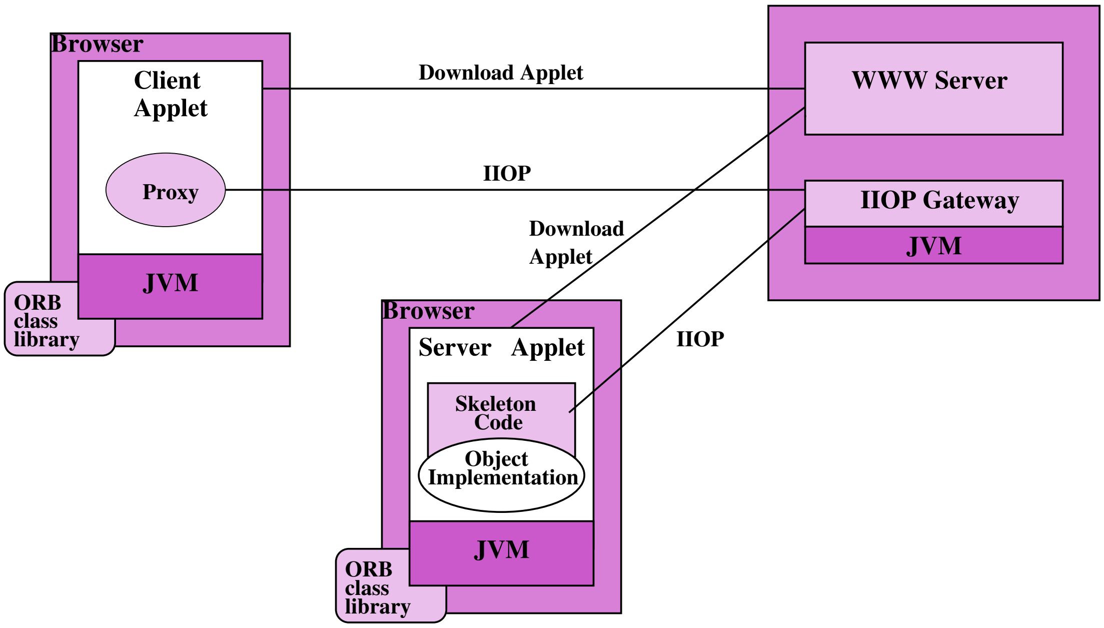
Client as Applet: diagram



Server as Java applet (1)

- A server can also be implemented as an applet.
- The associated object implementations cannot be made persistent and cannot make any data persistent because of the sandboxing restrictions on local resources.
- These objects will typically have transient object references.
- As with an applet client, for an unsigned applet, an IIOP gateway on the applet's host will be used to overcome networking restrictions whilst maintaining location transparency, unless invocations come from the originating host.
- In general, objects hosted by applets will be callback objects.
- Applets acting as servers need to handle two event loops: one to handle incoming CORBA requests and the other to deal with applet events such as those generated by the GUI.

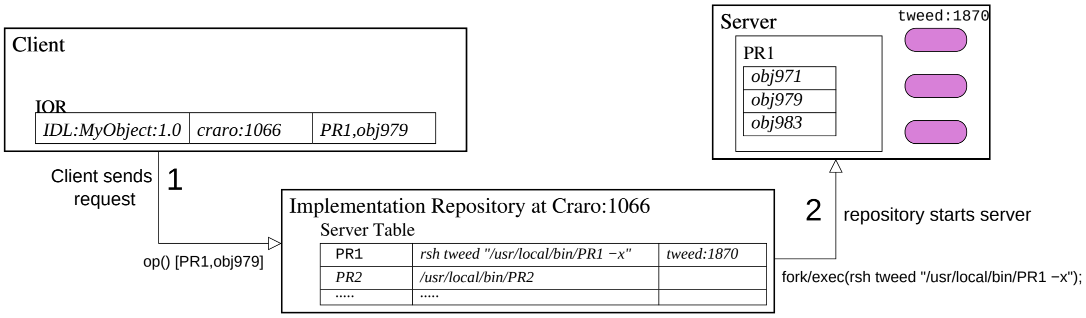
Server as Java applet (2)

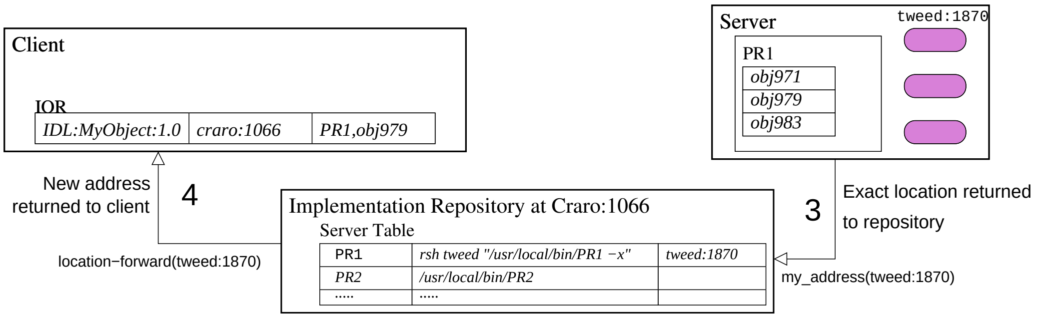


Implementation Repositories

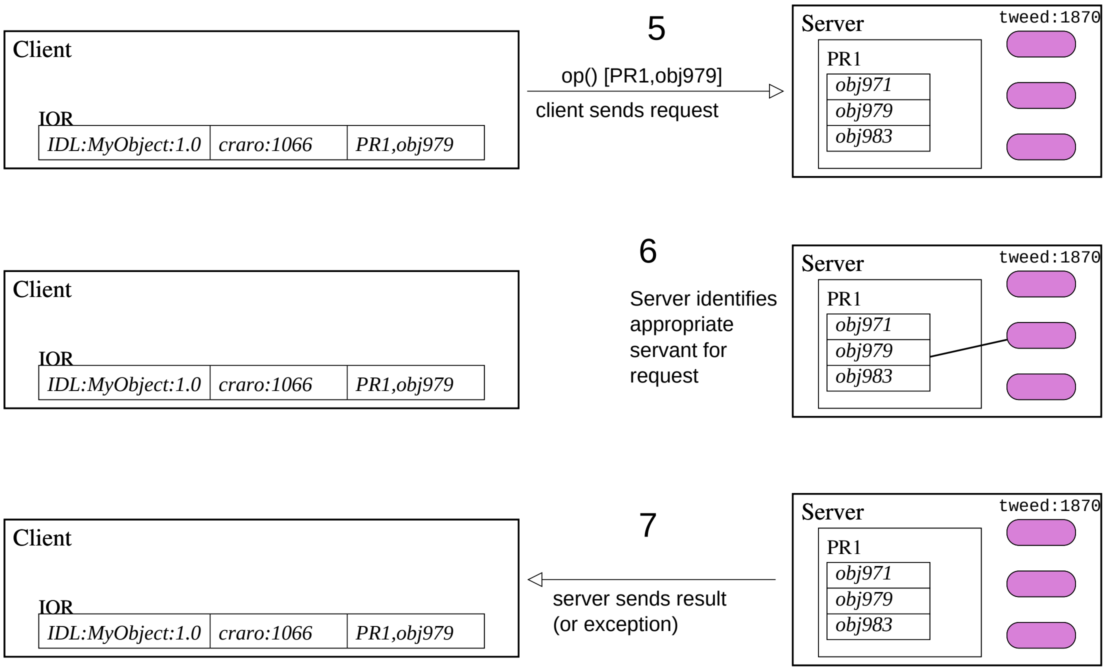
- A persistent IOR denotes the same persistent CORBA object across server and object adapter instantiations.
- Binding persistent IORs is carried out via an **implementation repository**.
- An implementation repository is a component proprietary to each ORB which stores information about where the executable code that implements objects resides and how to run it correctly.
- The implementation repository has several responsibilities:
  - maintain a register of known servers;
  - record which server is currently running at which host and port;
  - start servers on demand if registered for automatic activation.
- If a server creates persistent IORs, the server's host must be configured with the address of an implementation repository.

Binding a persistent IOR (1)





Binding a persistent IOR (2)



Persistent Binding (2)

- Use of an implementation repository has implications for scalability, performance and fault tolerance.
- If several hosts share an implementation repository forming a **location domain** then a server can transparently move to another host within the same domain.
- Note all objects implemented by the server must move together, or if the server has multiple POA, all objects referenced by the relevant POA must move together.
- Servers **can** migrate across location domains but only with negative impact on performance and scalability. This is achieved by maintaining entries in multiple implementation repositories either with explicit locations or with forwarding information.
- No commercial ORB currently support migration between location domains.

Dynamic Invocation

CORBA also provides mechanisms for making invocation requests on objects whose interface was not available at compile time. This means that no stub is available to the client process and no skeleton is present at the server. To overcome these omissions the **Dynamic Invocation Interface** (DII) and the **Dynamic Skeleton Inteface** (DSI) are provided.

- CORBA does not allow classes for stubs (or proxies) to be downloaded at runtime, so a client wishing to invoke a method on a remote object for which it has no stubs must do so via the DII.
- Using the DII the client explicitly builds the request—a invocation on a method with arguments.
- The DII is also used when the client wants to make a **deferred synchronous** invocation.
- When the DSI receives an invocation it inspects the contents of the request to discover its target object, the method to be invoked and the arguments. It then invokes the target.

Interface Repository

- The role of the interface repository is to provide information about registered IDL interfaces to clients and servers that require it.
- For any given interface type, this will be the names of the methods and the names and types of the arguments for each methods, as well as any exceptions.
- This adds a facility similar to *Java reflection* to CORBA objects.
- The IDL compiler assigns a unique type identifier to each IDL type in the interfaces it compiles. This is included in the remote object references of all objects of that type. It is known as the **repository ID**.
- The repository ID is used to look up an interface in the interface repository.
- The Java IDL ORB does not include an interface repository, or support dynamic invocation interfaces or dynamic skeleton interfaces.