# CORBA - Common Object Request Broker Architecture

# Review of some benefits of XDR & RPC

- XDR takes care of providing translation between differing data representations.
- RPC provides a portable, high-level programming interface.
  - The remote procedure interface defines all communication.
  - Clients and servers can be anywhere.
  - Formal interface for finding servers/services (portmapper).

# RPC for OOP

- We could extend RPC to handle OOP.
  - Calling an object method is like calling a remote procedure.
  - What about data members?
- We could make sure every object knows how to package up all data members (called marshaling) and send to the remote method.

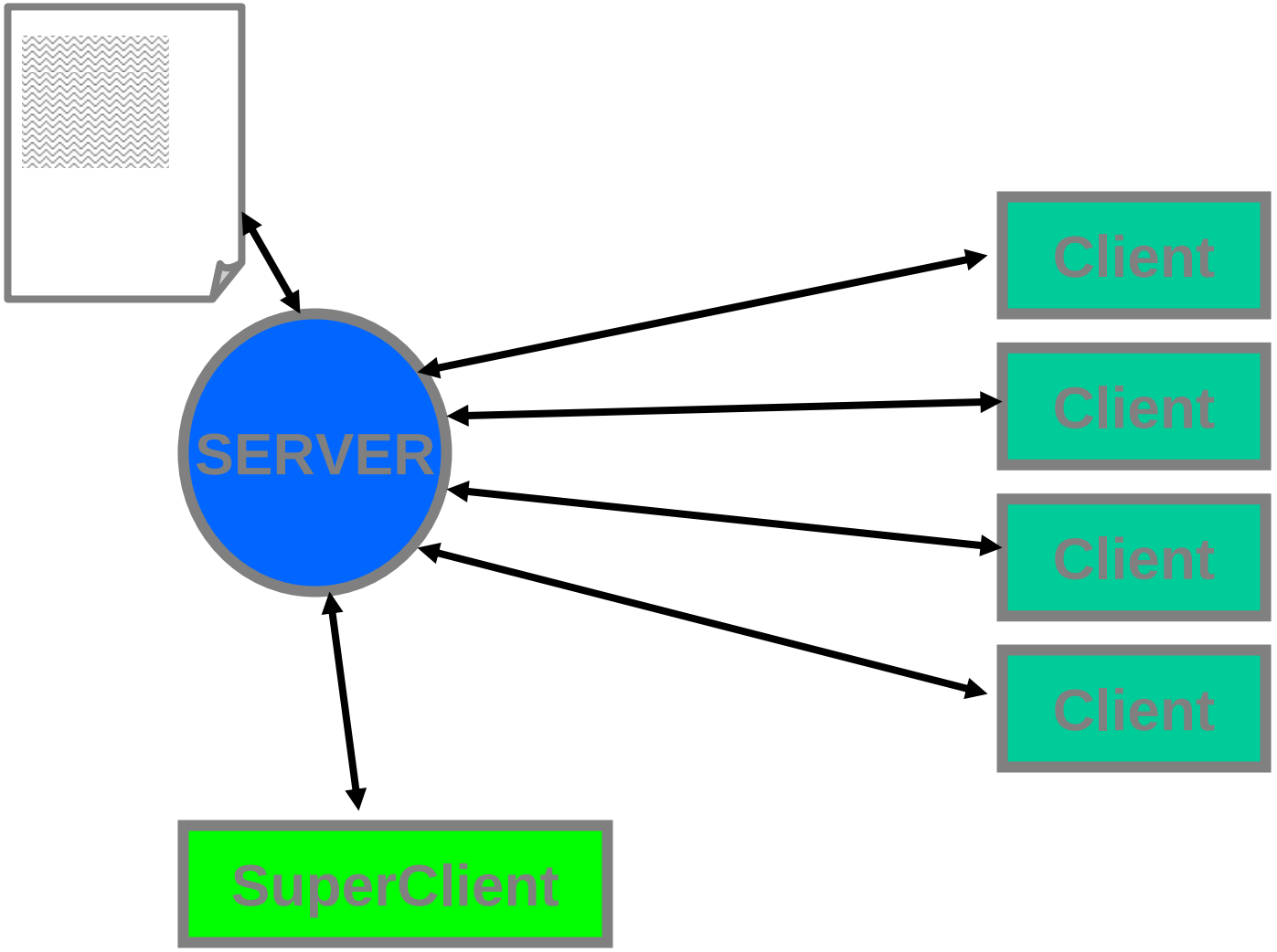
# Before looking at CORBA

- Think about developing a challenging network application.
- Our example will start simple and grow.

# A sample application - multiedit

- Consider a network application that supports shared document editing.
- Initially we will assume that the only thing in the document is text.
- We want to support many users:
  - All users can can see the current document.
  - All users can change the current document.

# A Server-Centric system



# Server-Centric System

- The Server has the document.
- Anyone can view the document.
- SuperClient determines who can change the document.
- Updates (changes to the document sent to all clients) could be:
  - Initiated by the server - this is called a “push”.
  - Initiated by the client - this is called a “pull”.

# Server software requirements

- Database of documents.
- Handle client requests
  - create a new document
  - get a copy of a document
  - make a change to a document
  - request recent changes
- Handle SuperClient requests
  - Establish current privileged client.
  - Undo changes ?

# Client software

- Need to be able to display (render) a document.
- Request privileged status.
- Translate user actions into a change request and send to server.
- Ask for a document.
- Ask for recent changes.

# SuperClient software

- Determine current privileged client.
- Set (change) privileged client.
- Request previous document version (undo changes).

# Assumptions

- We haven’t discussed everything (security, network efficiency, robustness, …) but - *let’s assume we have developed multi-edit and it works great*.
- Now assume we want to provide support for more elaborate documents.

# Document Entity Types

- We want to be able to use multi-edit to create and modify documents containing more than plain old text.
  - Tables
  - Graphs
  - Drawing Primitives (lines, arcs, shapes, etc).

# Adding new features

- Changes we need include:
  - Server: possibly document storage and retrieval, and new change request protocol to support new types of data.
  - Client:
    - GUI needs to support creation/modification of new document entity types.
    - Needs to know about new document entity types.

# Creeping Features

- Adding on new entity types one-by-one becomes tedious and repetitive.
- We eventually get bored, make mistakes.
- There must be a better way!

# OOP to the Rescue

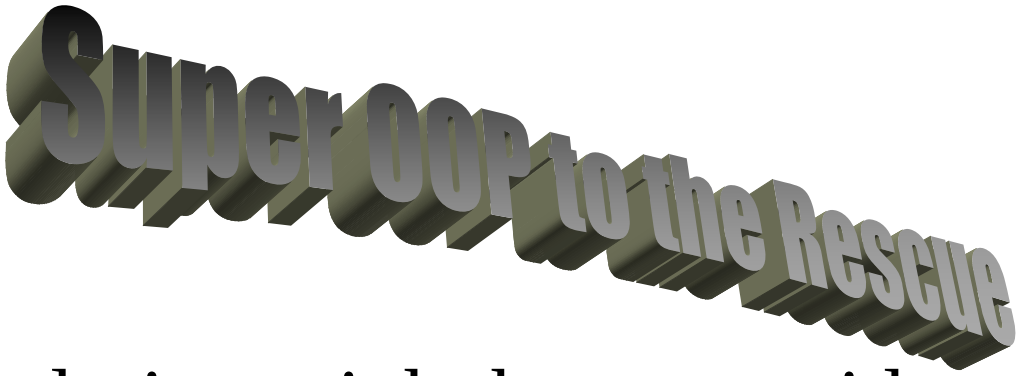
- View document entities as objects.
- Each entity could now:
  - Know how to render  itself.
  - Provide information to document manager about special features
  - Know how to save/retrieve itself from a file.
- Using objects, we focus on each object separately and never get bored.

# Using the Network.

- Developing an OO multi-edit that could be extended easily by adding new entity objects is nothing new.
- What we really want is to be able to provide an application framework and publish the interface so that anyone could develop new entity objects, and our customers could incorporate these new objects (over the network).

# Potential Problems

- How do we make sure everyone using multi-edit has a version that knows about all the new entity objects (requires recompilation).
- What happens to our documents when somebody changes an entity object definition?




- A better solution might be to provide some way to build a system based on dynamic, distributed objects.
- Object definitions are not static, they can be established and referenced at run time.
- For each object we can provide a single implementation that acts as a server.

# CORBA

- The notion of having objects distributed across the network has been around for a while.
- The Object Management Group (OMG) was formed in 1989 to create a set of standards that would facilitate the development of distributed object-oriented applications.

# CORBA

- The Common Object Request Broker Architecture (CORBA) is an open distributed object computing infrastructure being standardized by the Object Management Group (OMG).
- CORBA automates many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and demarshalling; and operation dispatching.

# Object Management Group

- OMG creates specifications, not implementations.
- Some Key Specifications:
  - OMA: Object Management Architecture.
  - CORBA: Common Object Request Broker Architecture.

# OMA Object Model

- Objects provide services.
- Clients makes a request to an object for a service.
- Client doesn’t need to know where the object is, or anything about how the object is implemented!
- Object interface must be known (public) - provides signature for each object method.

# Summary

- Concepts of CORBA
  - What are Objects in CORBA
  - Data Flow in CORBA
  - Definitions
- CORBA details
  - Request Invocation
  - Object References
  - The Portable Object Adapter (POA)
  - More About CORBA

# What is CORBA (Executive summary)

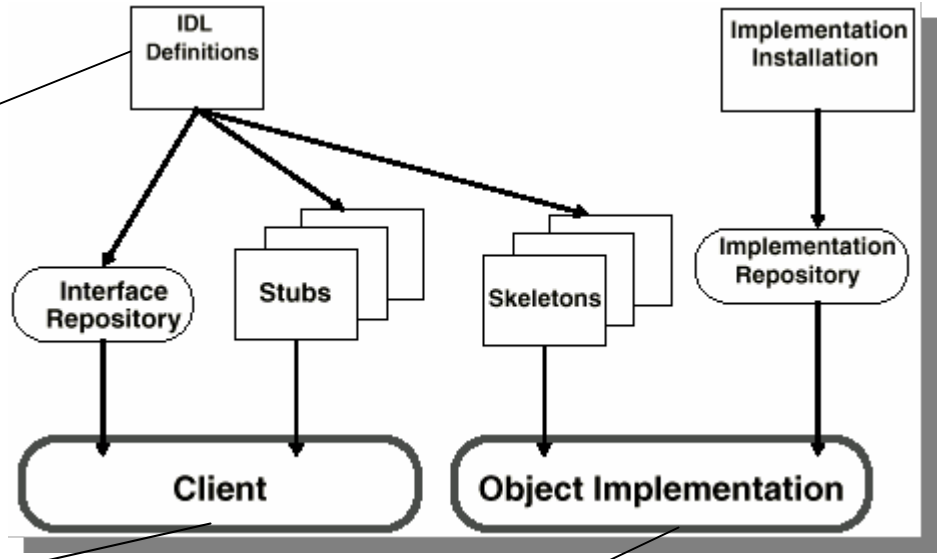
- ORB: Object Request Broker = manages remote access to objects
- CORBA: Common ORB Architecture = software bus for distributed objects
- CORBA provides a framework for distributed OO programming
  - remote objects are (nearly) transparently accessible from the local program
  - uses the client-server paradigm
  - platform and language independent

"an OO version of RPC"

  - but a framework rather than a technology => lot of theory

# How does CORBA work (Programmer summary)

```
Interface PS {
    attribute double current;
readonly attribute Ulong status;
    octet on(in octet value);
    void test(out long result) ;
        }
```




```
try {
PS aPS = PSHelper.bind(ORB,"PS1");
    if ( ! aPS.on(1) ) return;
    aPS.set_current(3.1415);
    print (aPS.status());
} catch (CORBAexception) {…}
```

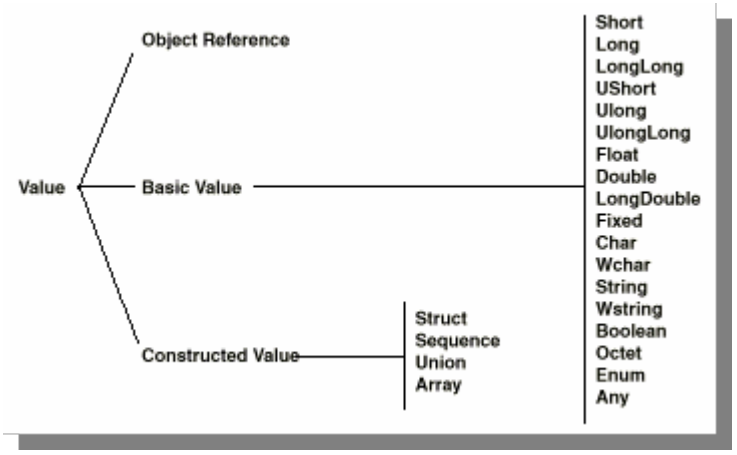
```
class PSimp extends PSImplBase{…};
…
PS thisPS = new PSimp("PS1") ;
BOA.obj_is_ready(thisPS);
BOA.impl_is_ready();
```

# CORBA Features

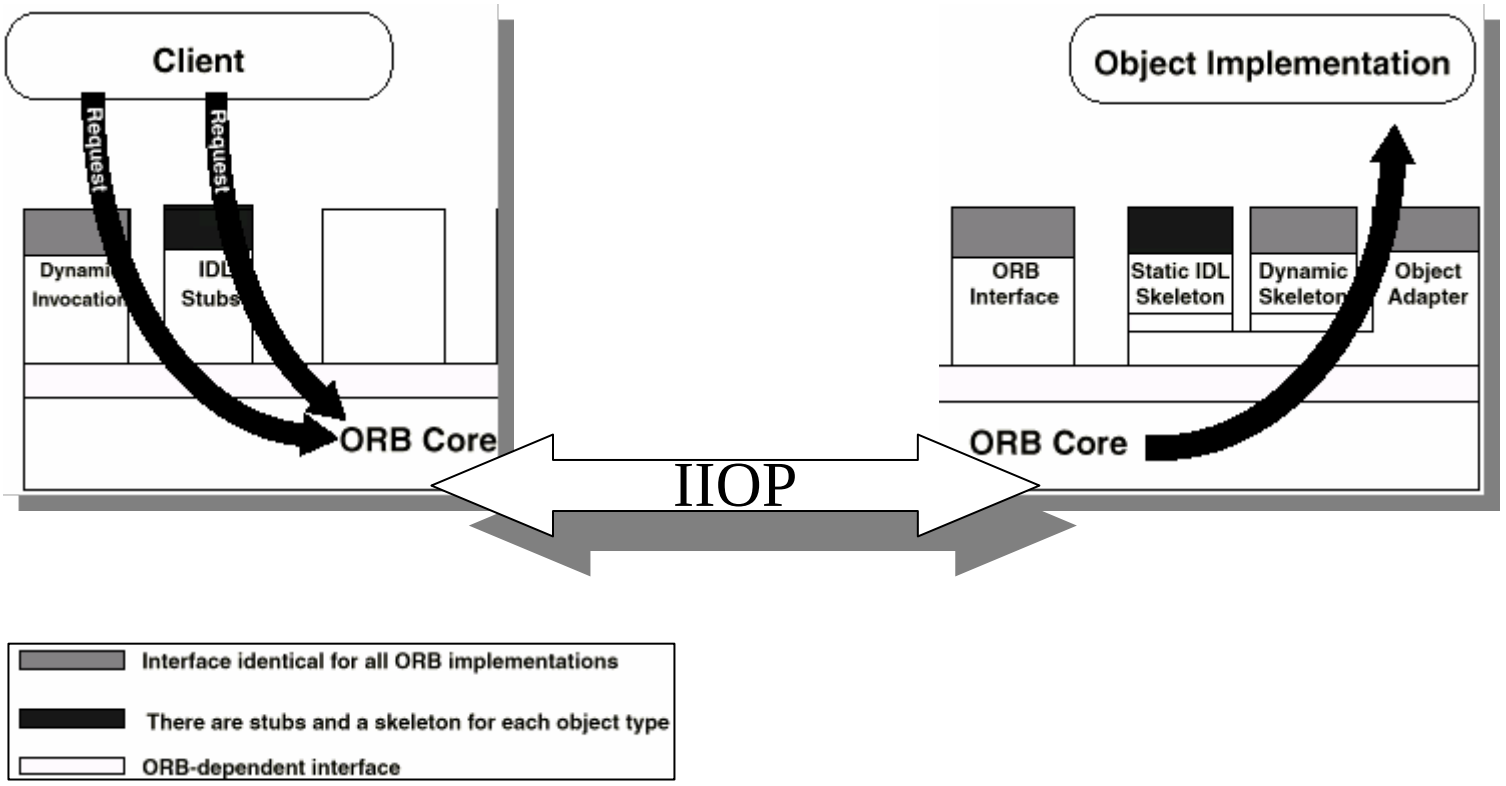
- Don’t worry about unique terminology - these are just words!
  - CORBA object
  - request, target object, object reference
  - client, server, servant
- Features
  - Interface Definition Language (IDL)
  - language mapping
    - official: C, C++, Samlltalk, COBOL, Ada, Java
    - also: Eiffel, Modula 3, Perl, **Tcl**, Objective-C, Python
  - Operation invocation and dispatch facilities
    - static (known at compile-time)
    - dynamic (determined at run-time)
  - Object adapters
    - Design pattern: adapt CORBA  object interface to servant
  - Inter-ORB Protocol

# What are Objects in CORBA

- Objects are abstract: not realized by any particular technology
  - An object system is a collection of objects that isolates the requestor of services (clients) from the providers of services by a well-defined **encapsulating interface**
- Objects "talk" through requests: operation, target object, zero or more parameters, optional request context
- Objects are described with interfaces
  - operations (methods)
  - attributes (properties)
  - Standard data types are supported
    - object references
    - Any



# Data Flow in CORBA



# Some Definitions

- ORB:
  - find the object implementation for the request, prepare the object implementation to receive the request and communicate the data making up the request.
  - ORB throws exceptions
  - ORB implementation is not defined in CORBA
- Object Adapter (POA, BOA, …)
  - provides ORB services to particular groups of object implementations
  - generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations.
- IIOP: Internet Inter-ORB Protocol
  - ORB's of different vendors can talk
  - TCP/IP implementation of GIOP

# More Definitions

- IDL: Interface Definition Language
  - IDL is the means by which a particular object implementation tells its potential clients what operations are available and how they should be invoked.

- Language mapping: recipe how to generate stubs&skeletons from IDL
  - Clients see objects and ORB interfaces through the perspective of a language mapping, bringing the object right up to the programmer's level.

- Interface Repository: where all interfaces are stored network-wide
  - provides information on interfaces at run-time

DII: Dynamic Invocation Interface

  - construct a remote method call at run-time without the use of stubs

# Object References

- Clients don’t have objects, they just have object references.
- Object references can be persistent (saved for use later).

# Object References

- Several references to one object
- Are strongly typed (at compile&run time)
- Implemented by proxies

- But how do you get a reference?
  - Bootstrap
    - via well known entry point (Naming service)
    - via reference-to-string (known URL, filename)
  - from a Object method call

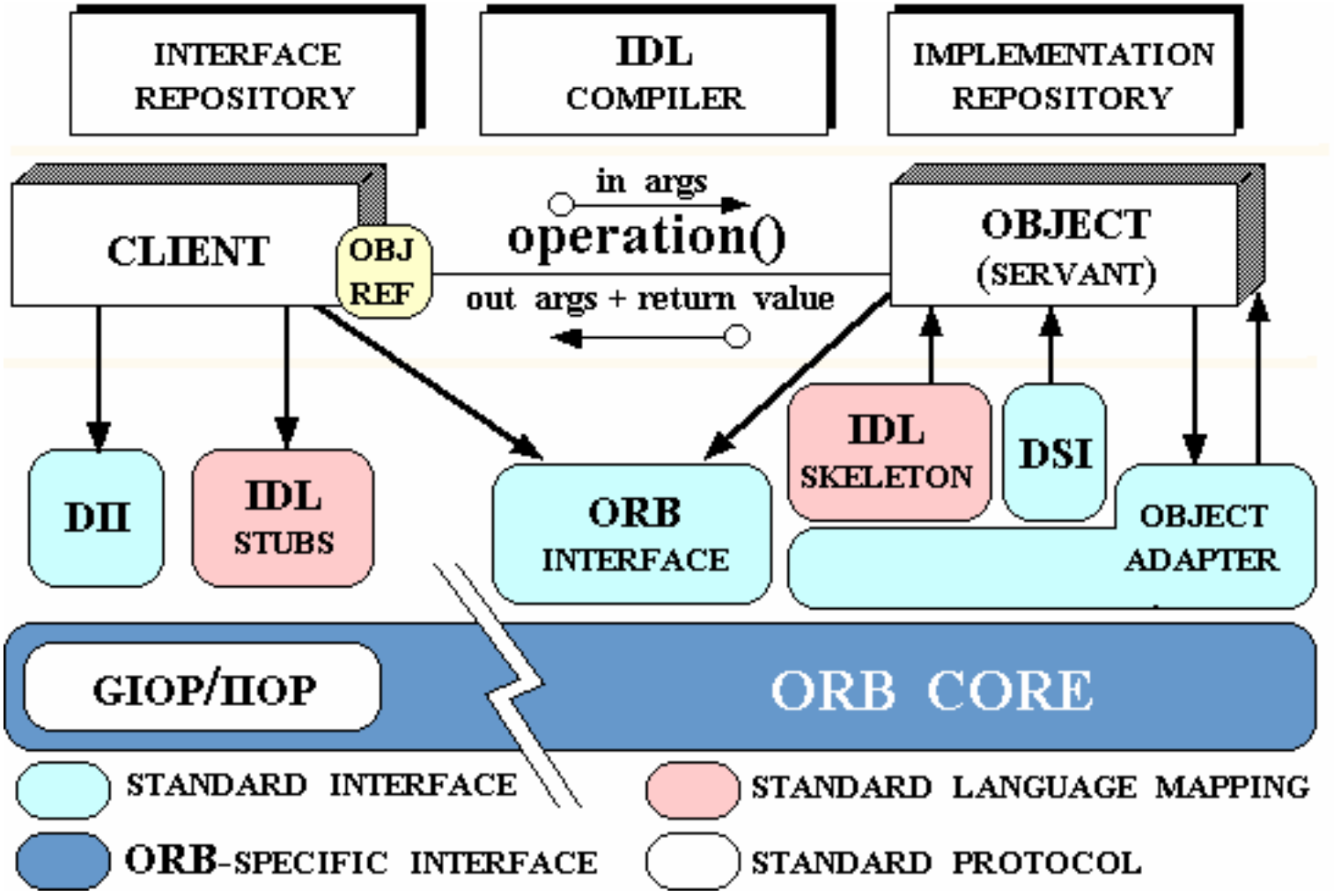
# Accessing Remote Methods

- Clients can call remote methods in 2 ways:
  - Static Invocation: using stubs built at compile time (just like with RPC).
  - Dynamic Invocation: actual method call is created on the fly. It is possible for a client to discover new objects at run time and access the object methods.

# Request Invocation

- This is transparently handled by the ORB
- Locate target object
- activate server application if not yet running
- transmit any arguments
- activate a servant if necessary
- wait for request to complete
- return any out/inout parameters and return value
- return exception if call fails

# CORBA




CORBA ORB Architecture

# CORBA

- **Object -** This is a CORBA programming entity that consists of an *identity*, an *interface*, and an *implementation*, which is known as a *Servant*.
- **Servant -** This is an implementation programming language entity that defines the operations that support a CORBA IDL interface. Servants can be written in a variety of languages, including C, C++, Java, Smalltalk, and Ada.
- **Client -** This is the program entity that invokes an operation on an object implementation. Accessing the services of a remote object should be transparent to the caller. Ideally, it should be as simple as calling a method on an object, i.e., obj->op(args). The remaining components in Figure 2 help to support this level of transparency.

# CORBA

- **Object Request Broker (ORB) -** The ORB provides a mechanism for transparently communicating client requests to target object implementations.
  - The ORB simplifies distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls.
  - When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.

# CORBA

- **ORB Interface -** An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries).
  - To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB.
  - This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

# CORBA

- **CORBA IDL stubs and skeletons -** CORBA IDL stubs and skeletons serve as the **glue** between the client and server applications, respectively, and the ORB.
  - The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler.
  - The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.

# CORBA

- **Dynamic Invocation Interface (DII) -** This interface allows a client to directly access the underlying request mechanisms provided by an ORB.
  - Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in.
  - Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking *deferred synchronous* (separate send and receive operations) and *oneway* (send-only) calls.

# CORBA

- **Dynamic Skeleton Interface (DSI) -** This is the server side's analogue to the client side's DII.
  - The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing.
  - The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.

# CORBA

- **Object Adapter -** This assists the ORB with delivering requests to the object and with activating the object.
  - More importantly, an object adapter associates object implementations with the ORB.
  - Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).

# Object Adapters

- Object Adapters provide a layer between object method requests and the servers that service the requests. Functions include:
  - generation of object references
  - starting up the actual server program(s)
  - handling security

# Basic Object Adapter

- Simplest Object Adapter, can support a number of different implementations:
  - one server that always is running
  - one program that can handle requests for multiple objects.
  - one program per object implementation.
  - one program for each object method.

# CORBA Services

- Some 20+ defined services
- check vendor for implementation and limitations !
- Some interesting services
  - Naming Service
    - “directory-based”
    - single or federated
  - Event Service
    - decouples suppliers from consumers
    - push or pull models
    - Notification Service ?
    - Messaging Service ?

# More About CORBA

- Other features of CORBA
  - **vendor specific implementations - check performance you need !**
  - Gateways to DCOM and OLE automation exist
  - CORBA Components
- Some buzzwords to know (and use)
  - thin client
  - three tier architecture
  - legacy systems
- Alternatives to CORBA:
  - sockets low level, used by CORBA
  - RPC not OO
  - RMI language dependent
  - DCOM maybe someday

# Building CORBA Applications

- Steps to build a CORBA application:
  1. Define the remote interface
  2. Compile the remote interface
  3. Implement the server
  4. Implement the client
  5. Start the applications

# Building CORBA Applications

- Define the remote interface
  - Define the interface for the remote object using the OMG's Interface Definition Langauge (IDL).
- Compile the remote interface
  - Compile the IDL and map the interface to the designated language.
- Implement the server
  - Use the skeletons it generates to develop the server application.
  - Implement the methods of the remote interface.
  - The server code includes a mechanism to start the ORB and wait for invocation from a remote client.

# Building CORBA Applications

- Implement the client
  - Use the stubs generated to develop the client application.
  - The client code builds on the stubs to start its ORB, look up the server using the name service, obtain a reference for the remote object, and call its method.
- Start the applications
  - Start the name service, then start the server, then run the client.