# *The OMG Approach*

CORBA, CCM, OMA, and MDA

## *Object Management Group*

- The Object Management Group (OMG), founded in 1989, is by far the largest consortium in the computing industry.
- OMG operates as a non-profit organization aiming at the standardization of "whatever it takes" to achieve interoperability on all levels of an open market for "objects."
- By 2002, around 800 member companies had joined OMG.

# *Object Interoperability*

- Originally, OMG concentrated on solving one fundamental problem: *How can distributed object-oriented systems implemented in different languages and running on different platforms interact?*
  - Code generated by two C++ compilers on the same platform was incompatible.
  - Differing object models from language to language made this worse.
  - Differences between platforms coupled by low-level socket communication or – in better cases – by RPC packages completed the picture of deep gaps everywhere.
- The outcome of tackling these "wiring" problems was the Common Object Request Broker Architecture (CORBA) first released in 1991.

# *OMG IDL*

- Interface definitions are written in OMG IDL.
- An ORB-specific IDL compiler creates stubs and skeletons that "glue" the client and server, respectively, to the ORB.
- Since the object handling a request may be written in a different language and running on a different platform than the client, the marshalling and de-marshalling of parameters is more complex than with RPC or Java RMI.

# *Invoking a Local Service*

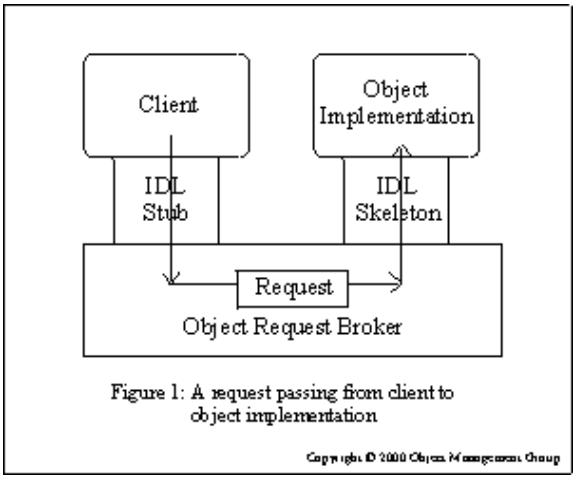


Figure 1: A request passing from client to object implementation



- Client and server could be implemented in different languages.
- This diagram doesn't show *any* of CORBA's mechanisms for load balancing, resource control, or fault tolerance on the server side.

# *Invoking a Remote Service*

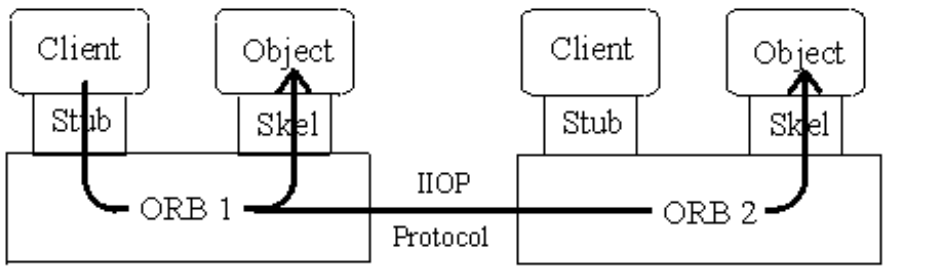


Figure 2: Interoperability uses ORB-to-ORB communication



- Client and server could be in different languages, running on different platforms, on different machines.
- ORBs communicate via Internet Inter-ORB Protocol (IIOP)

# *Interoperability Standards*

- From the beginning, the goal behind CORBA was to enable open interconnection of a wide variety of languages, implementations, and platforms.
- Thus, OMG never settled on "binary" standards (standards at the level of deployable executables).
  - Everything is carefully standardized to allow for many different implementations and for individual vendors of CORBA-compliant products to add value.
  - The downside of this very open approach is that individual CORBA-compliant products cannot interoperate on an efficient binary level, but must engage instead in relatively expensive high-level protocols such as IIOP.

# *Invoking a Remote Service (2)*

- The same interface definition (in OMG IDL) is compiled twice – once for each ORB.
  - ORB vendors supply IDL compilers that produce stubs and skeletons that work with their particular ORB.
  - As long as the glue for each ORB was produced from the same IDL specification, and the ORBs communicate using the same inter-orb protocol (e.g. IIOP), the remote invocations will work the same as local invocations.
- The separation into calling client and called object does not impose an asymmetric architecture, such as client-server computing:
  - The same process can be both issuing and receiving calls.

# *OMG IDL*

- For each object type, you define an interface in OMG IDL.
  - Any client that wants to invoke an operation on the object *must* use this IDL interface to specify the operation it wants to perform, and to marshal the arguments that it sends.
  - When the invocation reaches the target object, the *same* interface definition is used there to unmarshal the arguments so that the object can perform the requested operation with them. The interface definition is then used to marshal the results for their trip back, and to unmarshal them when they reach their destination.
- The IDL interface definition is independent of programming language, but OMG has standardized mappings from IDL to C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript.

# *IDL Language Mappings*

- A mapping assigns a language variable type to each IDL variable type, and a translation from IDL's operation format to the language's invocation of a member function or other operation invocation format.
- Language mappings are very precise:
  - When you apply an OMG language mapping to an IDL file, you always get the *same* language constructs out.
  - This provides portability in addition to predictability.
- *IDL Compilers* implement language mappings in software.
  - Every ORB comes with one or more IDL compilers, one for each language that it supports.
  - The compiler will produce at least two files: one for the client stub; the other for the object skeleton.

# *IDL Example*

```
module Example {

  struct Date {
    unsigned short Day;
    unsigned short Month;
    unsigned short Year;
  }

  interface Ufo {
    readonly attribute unsigned long ID;
    readonly attribute string Name;
    readonly attribute Date FirstContact;
    unsigned long Contacts ();
    void RegisterContact (in Date dateOfContact);
  }
}
```

# *Object References*

- In CORBA, every object instance has its own unique *object reference*, an identifying electronic token.
- Clients use the object references to direct their invocations, identifying to the ORB the exact instance they want to invoke (Ensuring, for example, that the books you select go into your own shopping cart, and not into your neighbor's.)
- The client acts as if it's invoking an operation on the object instance, but it's actually invoking on the IDL stub which acts as a proxy. Passing through the stub on the client side, the invocation continues through the ORB (Object Request Broker), and the skeleton on the implementation side, to get to the object where it is executed.

# *Object References (2)*

- Although the ORB can tell from the object reference if the target object is remote, the client can not.
  - The user may know this also, because of other knowledge - for instance, that all accounting objects run on the mainframe at the main office in Tulsa.
- There is nothing in the object reference token that the client holds and uses at invocation time that identifies the location of the target object.
- This ensures *location transparency* - the CORBA principle that simplifies the design of distributed object computing applications.
- Easy ways for the client to obtain an object reference include the *Naming Service* and the *Trader Service*.

# *Naming and Trader Services*

- The Naming Service allows you to associate abstract names with CORBA objects and allows clients to find those objects by looking up the corresponding names.
- An object in the CORBA Trading Service does not have a name. Rather, a server advertises an object in the Trading Service based on the kind of service provided by the object.
- A client locates objects of interest by asking the Trading Service to find all objects that provide a particular service. The client can further restrict the search to select only those objects with particular characteristics.

# *Dynamic Invocation Interface*

- Using the DII, a client can invoke an operation on a new type of object that it's just discovered (for example, using the Trader Service) without using a precompiled stub.
- DII-based invocations differ from stub-based invocations in the same way that scripts differ from programs: they're interpreted at run time, and not compiled in a previous step.
- The client programmer must write code to retrieve the object's IDL interface definition from the ORB's interface repository, and construct an invocation using interfaces defined on the ORB.

# *IDL to C++ and Java Mappings*

| IDL | C++ | Java |
|---|---|---|
| module | namespace | package |
| interface | class (abstract) | interface |
| object reference (of type T) | T_var and T_ptr<br>Implemented as counted pointers. Use CORBA::duplicate and CORBA::release for memory management. | T |
| out and inout parameters (of type T) | T_ptr& | THolder with public *value* field of type T |
| string | char*<br>Must use:<br>CORBA::string_alloc<br>CORBA::string_dup<br>CORBA::string_free<br>for dynamic allocation | java.lang.String |

# *IDL to C++ and Java Mappings (2)*

| IDL | C++ | Java |
|---|---|---|
| short | CORBA::Short | short |
| long | CORBA::Long | int |
| long long | CORBA::LongLong | long |
| unsigned short | CORBA::UShort | short |
| unsigned long | CORBA::ULong | int |
| unsigned long long | CORBA::ULongLong | long |
| float | CORBA::Float | float |
| double | CORBA::Double | double |
| long double | CORBA::LongDouble | ??? |
| char | CORBA::Char | char |
| wchar | CORBA::WChar | char |
| boolean | CORBA::Boolean | boolean |
| octet | CORBA::Octet | byte |