Introduction to Distributed Objects with CORBA

Douglas C. Schmidt

Washington University, St. Louis

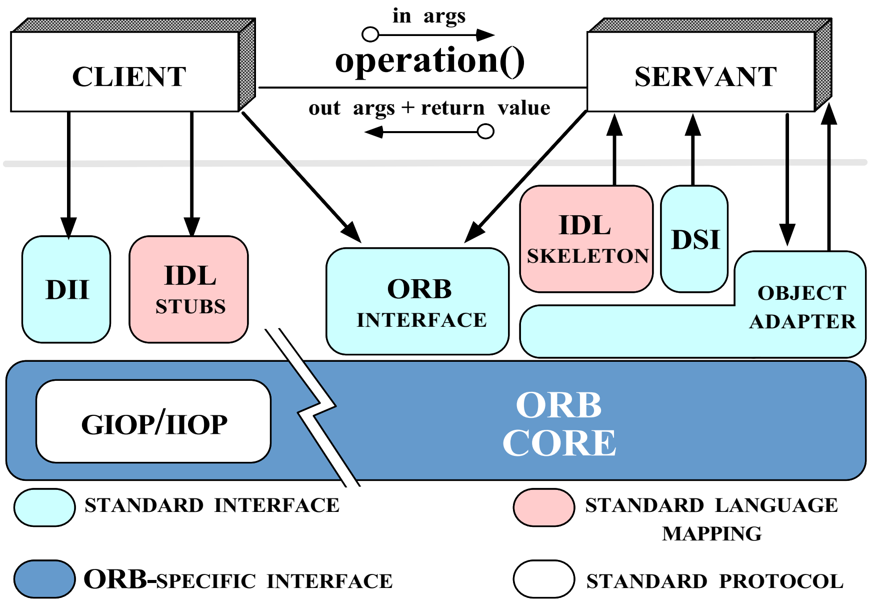
http://www.cs.wustl.edu/~schmidt/

schmidt@cs.wustl.edu

Motivation

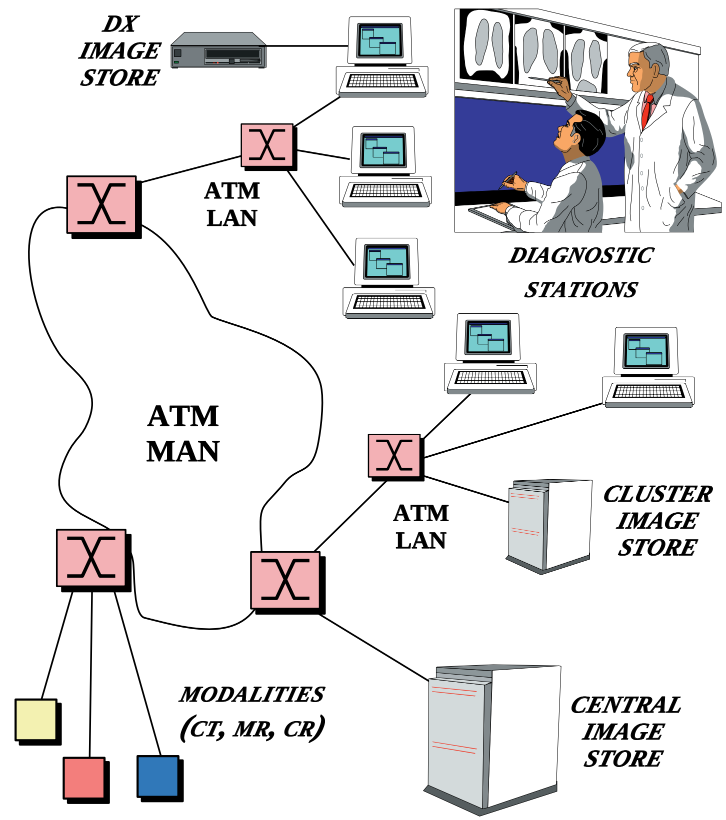
- Typical state of affairs today is the "Distribution Crisis"
  - Computers and networks get faster and cheaper
  - Communication software gets slower, buggier, more expensive
- *Accidental complexity* is one source of problems, *e.g.*,
  - Incompatible software infrastructures
  - Continuous rediscovery and reinvention of core concepts and components
- *Inherent complexity* is another source of problems
  - *e.g.*, latency, partial failures, partitioning, causal ordering, etc.

Candidate Solution: CORBA

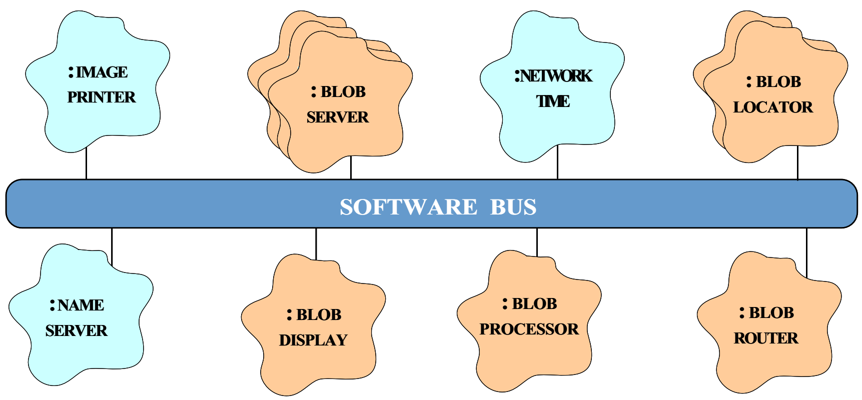


- Goals
  1. *Simplify development of distributed applications*
  2. *Provide flexible foundation for higher-level services*

Distributed Medical Imaging

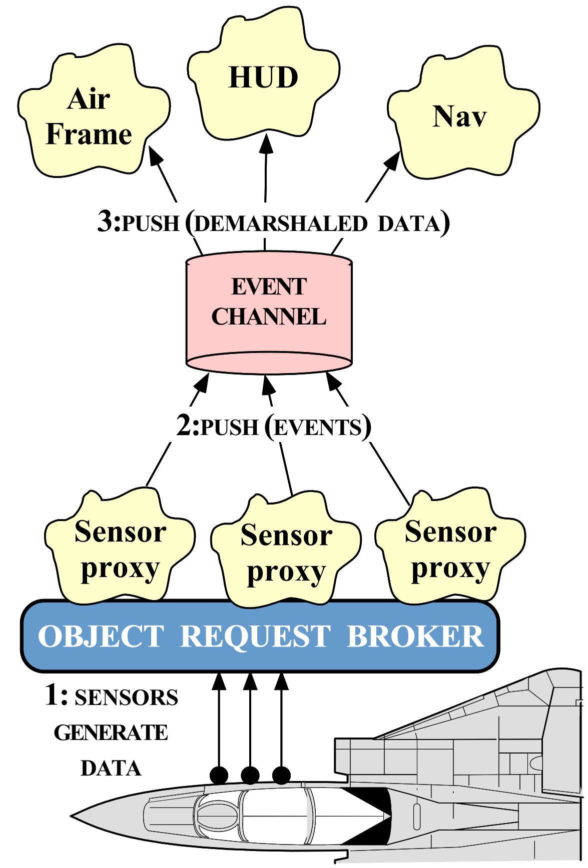


Distributed Objects in Medical Imaging

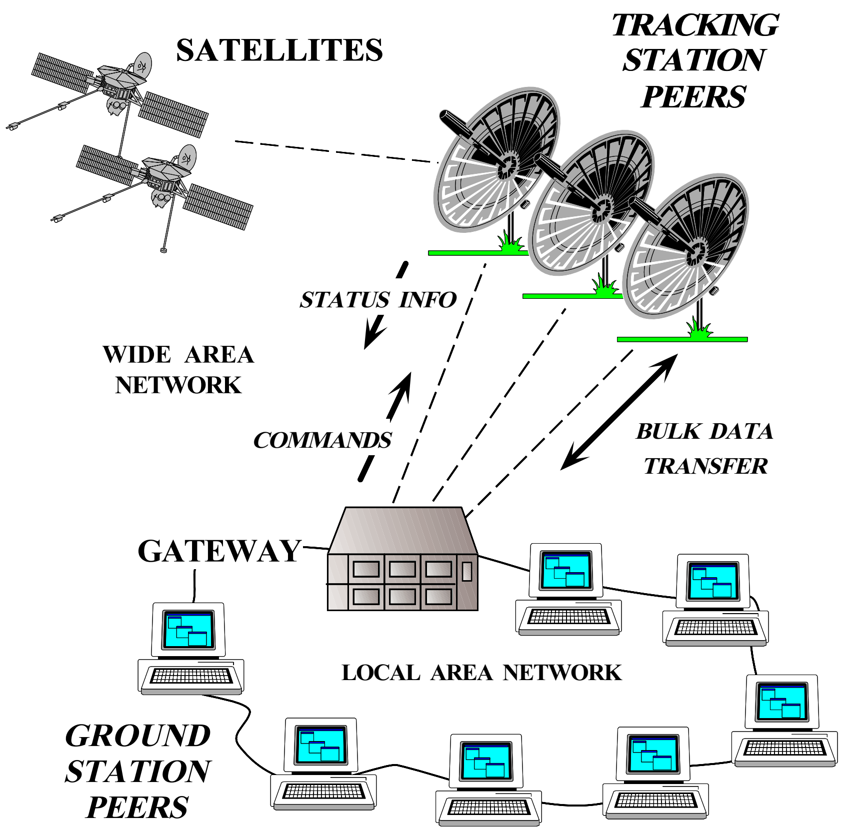


- "Blob" == Binary Large OBject

Real-time Avionics



Telecommunications



Outline

- Motivation
- Example CORBA Applications
- Coping with Changing Requirements
- Overview of CORBA Architecture
- Evaluations and Recommendations

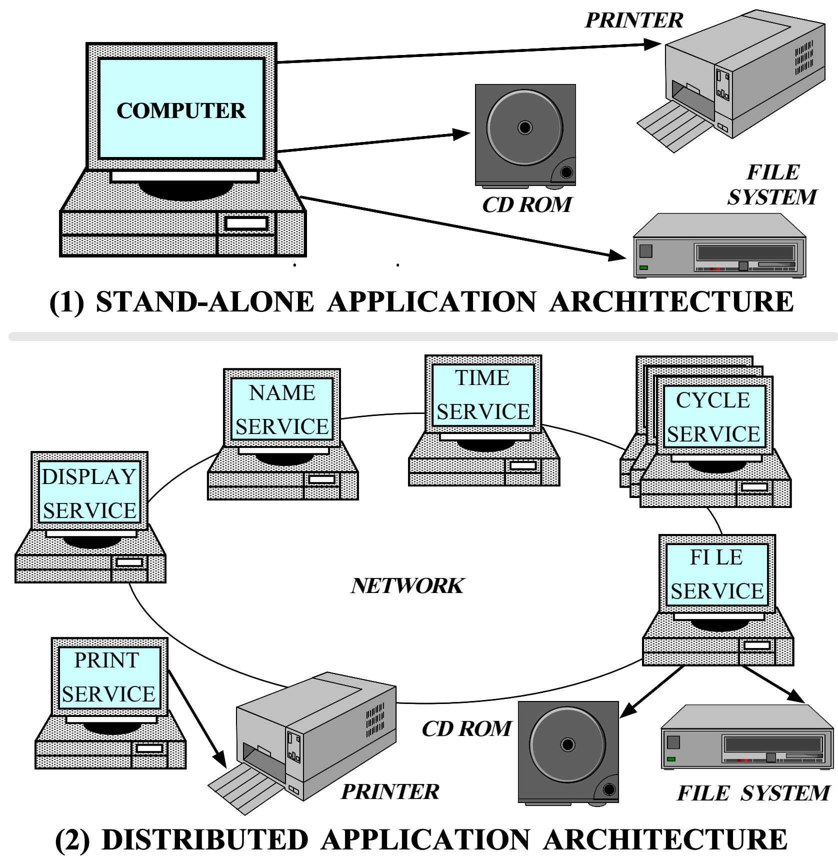
Motivation

- Developing distributed applications whose components collaborate *efficiently*, *reliably*, *transparently*, and *scalably* is hard

- To help address this challenge, the Object Management Group (OMG) is specifying the *Common Object Request Broker Architecture* (CORBA)
  - OMG is a consortium of computer companies
    - *e.g.*, Sun, HP, DEC, IBM, IONA, Visigenic, etc.

- Version 2.1 of the CORBA spec is now available
  - http://www.omg.org/corba/corbiiop.htm

Stand-alone vs. Distributed Application Architectures



Sources of Complexity

- Distributed application development exhibits both *inherent* and *accidental* complexity

- *Inherent complexity* results from fundamental challenges in the distributed application domain, *e.g.*,
  - Addressing the impact of latency
  - Detecting and recovering from partial failures of networks and hosts
  - Load balancing and service partitioning
  - Consistent ordering of distributed events

Sources of Complexity (cont'd)

- *Accidental complexity* results from limitations with tools and techniques used to develop distributed applications, *e.g.*,
  - Lack of type-safe, portable, re-entrant, and extensible system call interfaces and component libraries
  - Inadequate debugging support
  - Widespread use of *algorithmic* decomposition
    * Fine for *explaining* network programming concepts and algorithms but inadequate for *developing* large-scale distributed applications
  - Continuous rediscovery and reinvention of core concepts and components

Motivation for CORBA

- Simplifies application interworking
  - CORBA provides higher level integration than traditional "untyped TCP bytestreams"
- Provides a foundation for higher-level distributed object collaboration
  - *e.g.*, Windows OLE and the OMG Common Object Service Specification (COSS)
- Benefits for distributed programming similar to OO languages for non-distributed programming
  - *e.g.*, encapsulation, interface inheritance, and object-based exception handling

CORBA Contributions

- CORBA addresses two challenges of developing distributed systems:

1. Making distributed application development no more difficult than developing centralized programs
   - Easier said than done due to:
     * *Partial failures*
     * *Impact of latency*
     * *Load balancing*
     * *Event ordering*
2. Providing an infrastructure to integrate application components into a distributed system
   - *i.e.*, CORBA is an "enabling technology"

CORBA Quoter Example

- Ideally, to use a distributed service, we'd like it to look much like a non-distributed service:

```
int
main (void)
{
  // Use a factory to bind to any quoter.
  Quoter_var quoter = bind_quoter_service ();

  const char *stock_name = "ACME ORB Inc.";

  CORBA::Long value = quoter->get_quote (stock_name);
  cout << stock_name << " = " << value << endl;
  return 0;
}
```

- Unfortunately, life is harder when errors occur...

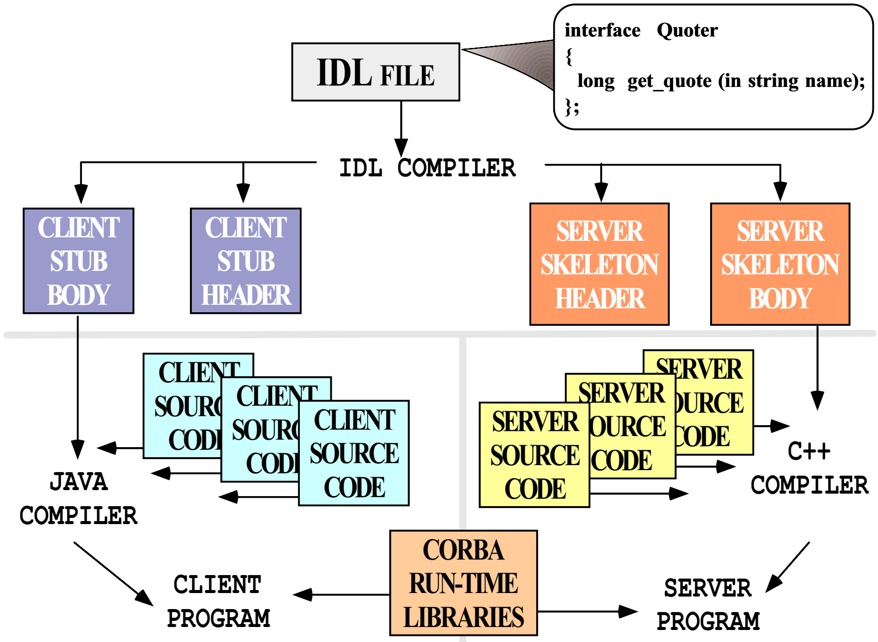
CORBA Quoter Interface

- We need to write an OMG IDL `interface` for our `Quoter` object
  - This interface is used by both clients and servers

```
// IDL interface is like a C++ class
// or Java interface.
interface Quoter
{
  exception Invalid_Stock {};

  long get_quote (in string stock_name)
    raises (Invalid_Stock);
};
```

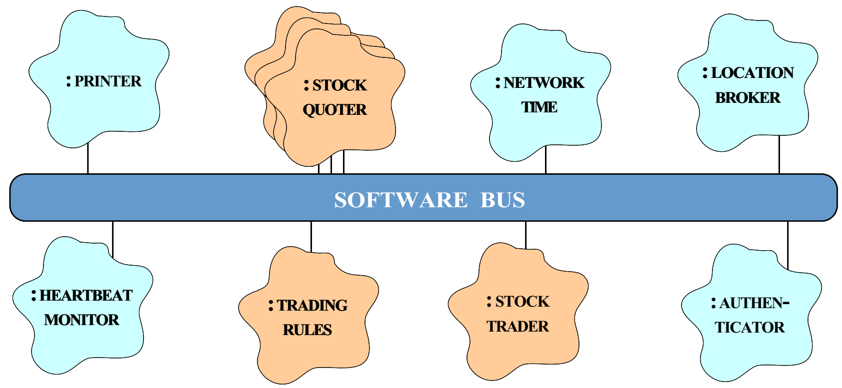
- The use of OMG IDL promotes language independence, location transparency, modularity, and robustness

OMG IDL Compiler



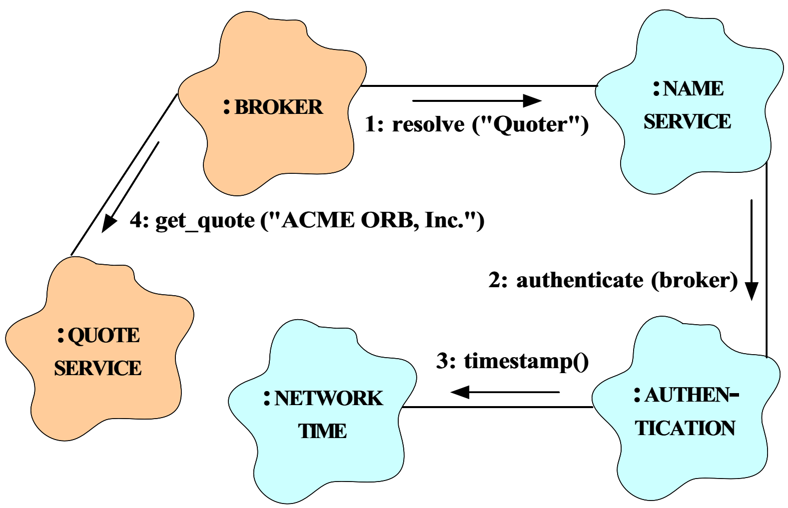
- A OMG IDL compiler generates client *stubs* and server *skeletons*

Software Bus



- CORBA provides a communication infrastructure for a heterogeneous, distributed collection of collaborating objects
- Analogy to "hardware bus"

CORBA Object Collaboration



- In theory, collaborating objects may be either local (*co-located*) or remote (*distributed*)
- In practice, beware of traps and pitfalls...

Communication Features

- Communication features of standard CORBA:
  - Supports both synchronous and "quasi-asynchronous" communication styles
    - *i.e.*, *oneway*, *twoway*, and *deferred synchronous*
  - Supports best-effort, uni-cast communication
    - Note that all of these features may be extended depending on vendor "quality of service"
- CORBA objects may collaborate in a *client/server*, *peer-to-peer*, or *publish/subscribe* manner
  - *client/server* and *peer-to-peer* are built into the standard library
  - *e.g.*, COSS Event Services defines a publish/subscribe communication paradigm

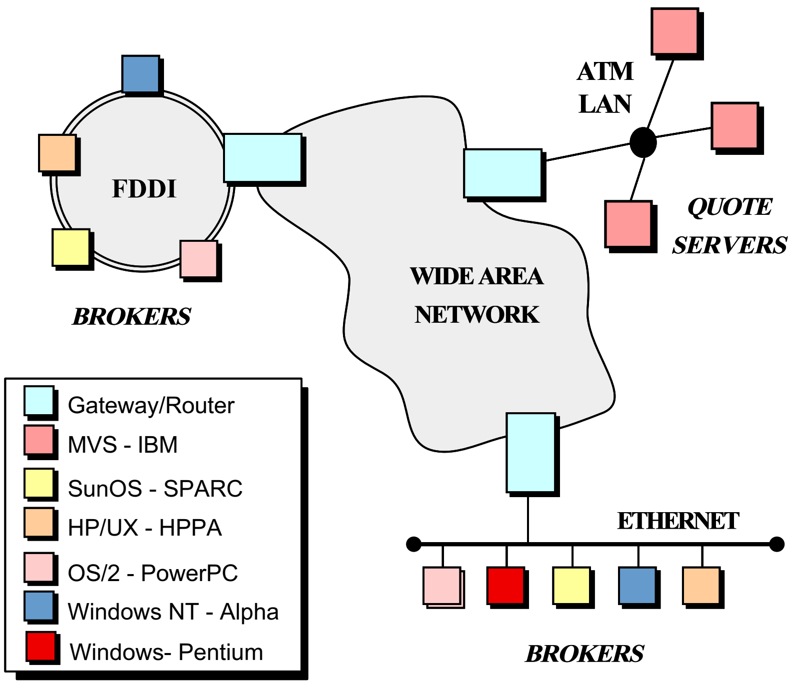
Related Work

- *Traditional RPC* (*e.g.*, DCE)
  - Provides "procedural" integration of application services
  - Doesn't provide object abstractions or message passing
  - Doesn't address inheritance of interfaces
- *Windows OLE/COM*
  - Traditionally limited to desktop applications
  - Does not address heterogeneous distributed computing (yet)
    - Distributed COM (DCOM) is now appearing on multiple platforms

Related Work (cont'd)

- *Java RMI*
  - Limited to Java only
  - Can be extended into other languages (*e.g.*, C or C++) by using a bridge across JNI
  - Well-suited for all-Java applications because of its tight integration with the Java virtual machine

CORBA Application Example



- *Stock quoter/trader application*

Stock Quoter/Trader Application

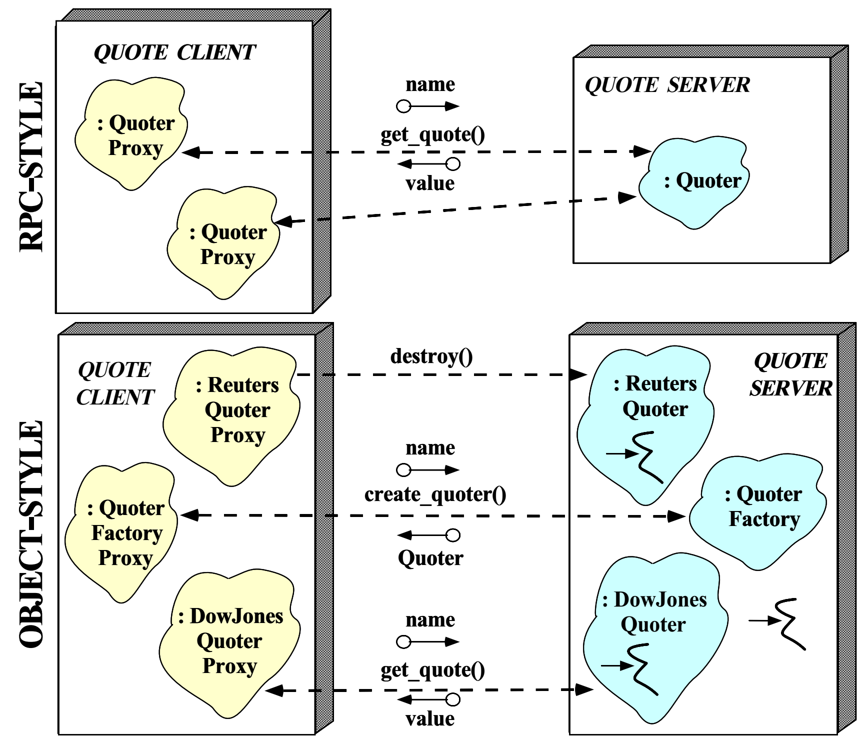
- The quote server(s) maintains the current stock prices
- Brokers access the quote server(s) via CORBA interfaces and the CORBA run-time
- Since the server(s) and the brokers are distributed, the solution must work across LAN and WAN environments

Initial OMG IDL Quoter Specification

- A `module` is a high-level grouping construct

```
module Stock
{
  // An exception is a combination of a struct
  // and an event.
  exception Invalid_Stock {};
  exception Invalid_Quoter {};

  // Interface is similar to a C++ class.
  interface Quoter {
    long get_quote (in string stock_name)
      raises (Invalid_Stock);
  };

  // Manage the lifecycle of a Quoter object.
  interface Quoter_Factory {
    // Returns a new Quoter selected by name
    // e.g., "Dow Jones," "Reuters,", etc.
    Quoter create_quoter (in string name)
      raises (Invalid_Quoter);

    void destroy_quoter (in Quoter quoter);
  };
};
```


RPC-style vs. Object-style Communication



Compiling the Interface Definition

- Running the `Stock` module definition through the IDL compiler generates *client* stubs and *server* skeletons
  - The client stub is a proxy that handles *parameter marshalling* from requestor
  - The server skeleton handles *parameter demarshalling* to the target

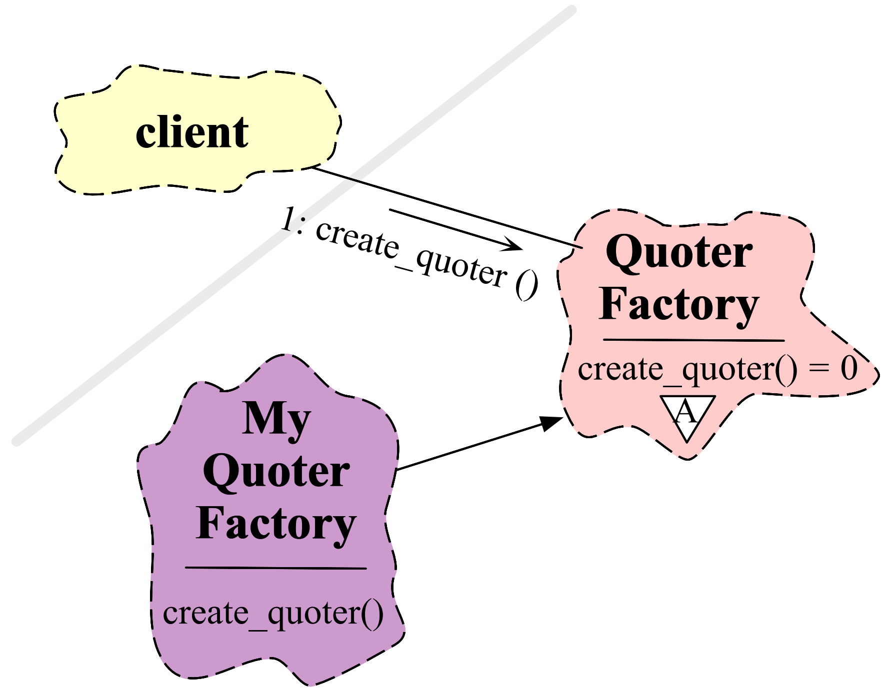
- CORBA associates a servant to a generated IDL skeleton as follows:
  1. The Class form of the Adapter pattern (inheritance)

     POA_Stock::Quoter

  2. The Object form of the Adapter pattern (object composition, *i.e.*, TIE)

     template <class Impl>
     class POA_Stock::Quoter_tie

Using the Class Form of the Adapter Pattern with POA_Stock::Quoter_Factory



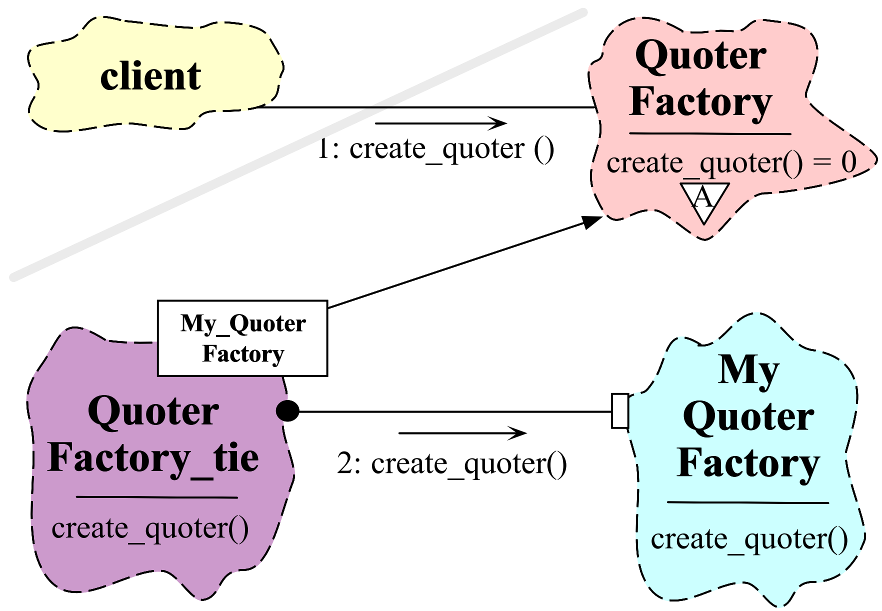
A Servant based on Inheritance

- Note inheritance from POA_Stock::Quoter_Factory

```
class My_Quoter_Factory
  : public virtual POA_Stock::Quoter_Factory
{
public:
  My_Quoter_Factory (void);

  // Factory method for creation.
  virtual Stock::Quoter_ptr create_quoter
    (const char *name);

  // Factory method for destruction.
  virtual void destroy_quoter
    (Stock::Quoter_ptr quoter);
  // ...
};
```

- The drawback is that implementations inherit from generated skeletons
  - Can create a "brittle" hierarchy
  - Hard to integrate with legacy code (*i.e.*, distributing an all-local application)

Using the Object Form of the Adapter Pattern with TIE



A Servant based on Object Composition

- Allows the distribution of classes that were developed without prior knowledge of CORBA

```
// Note, there is no use of inheritance and
// methods need not be virtual!
class My_Quoter_Factory
{
public:
  My_Quoter_Factory (void);

  // Factory method for creation.
  Stock::Quoter_ptr create_quoter (const char *name);

  // Factory method for destruction.
  void destroy_quoter (Stock::Quoter_ptr quoter);

private:
  // ...
};
```


TIE-based Implementations

- IDL compiler generates TIE adapter class

```
class POA_Stock
// Note: POA_Stock is really a namespace
{
  template <class Impl>
  class Quoter_Factory_tie : public Quoter_Factory
  {
    // ...
  };
  // ...
};
POA_Stock::Quoter_Factory_tie <My_Quoter_Factory>
factory (new My_Quoter_Factory);
```

- This scheme places an implementation pointer object within the TIE class

- All method calls via the interface class are then delegated to the implementation object

  - *i.e.*, the “object form” of the Adapter pattern!

Implementing the Methods

- A developer then writes C++ definitions for the methods in class My_Quoter_Factory:
  - This solution uses the inheritance approach

```
Stock::Quoter_ptr
My_Quoter_Factory::create_quoter (const char *name)
{
  POA_Stock::Quoter *quoter;

  // Perform Factory Method selection of
  // the subclass of Quoter.
  if (strcmp (name, "Dow Jones") == 0)
    quoter = new Dow_Jones_Quoter;
  // ...
  else if (strcmp (name, "My Quoter") == 0)
    // Dynamically allocate a new My_Quoter object.
    quoter = new My_Quoter;
  else
    throw Stock::Invalid_Quoter; // Raise exception.

  // This will create a Stock::Quoter_ptr and register
  // the servant with the default_POA.
  return quoter->_this ();
};
```


The Main Server Program

- This example uses "shared activation" mode

```
void
main (int argc, char *argv[])
{
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, 0);
  CORBA::Object_ptr pfobj =
            orb->resolve_initial_references("RootPOA");
  PortableServer::POA_var rootPOA;
  rootPOA = PortableServer::POA::_narrow(pfobj);

  My_Quoter_Factory factory;
  // Could also use the TIE approach:
  // POA_Stock::Quoter_Factory_tie <My_Quoter_Factory>
  //      factory (new My_Quoter_Factory);

  // Register servant with POA
  // Could also use factory._this () here
  rootPOA->activate_object (&factory);

  // Will block indefinitely waiting for incoming
  // invocations and dispatching method callbacks.
  orb->run ();

  // After run() returns, the ORB has shutdown.
}
```

Client-side Stubs

- Generated by an OMG IDL compiler, *e.g.*,

```
class Stock
// Note: Stock is really a namespace
{

  class Quoter // Quoter IS-A CORBA::Object.
    : public virtual CORBA::Object
  {
  public:
    // Proxy interface.
    CORBA::Long get_quote (const char *stock_name);
  };

  class Quoter_Factory // Quoter_Factory IS-A CORBA::Object
    : public virtual CORBA::Object
  {
  public:
    // Proxy Factory method for creation.
    Quoter_ptr create_quoter (const char *name);

    // Proxy Factory method for destruction.
    void destroy_quoter (Quoter_ptr quoter);
  };
// ...
};
```


OMG IDL Mapping Rules

- The CORBA specification defines mappings from CORBA IDL to various programming languages
  - *e.g.*, C++, C, Smalltalk, Java (proposed)
- Mapping OMG IDL to C++
  - Each **`module`** is mapped to a class or namespace
  - Each **`interface`** within a **`module`** is mapped to a nested C++ class
  - Each operation is mapped to a C++ method with appropriate parameters
  - Each read/write attribute is mapped to a pair of get/set methods
    - A read-only attribute is only mapped to a single get method
  - An **`Environment`** is defined to carry exceptions in languages lacking exceptions

Binding a Client to a Target Object

- Typically two steps:

  1. A CORBA client (requestor) obtains an "object reference" from a server

     – *e.g.*, May use a naming service or a locator service

  2. The client may then invoke methods on its proxy

- Recall that object references may be passed as parameters to other remote objects

Binding a Client to a Target Object (cont'd)

- Object references are represented by different generated types

  – **`_ptr`** require programmer management of reference ownership

    * Pointer to object reference

  – **`_var`** internally manages reference ownership

    * Auto pointer to object reference

  – **`_out`** eases passing out parameters between client and server

    * Never used directly by user

A Client Program

- Client binds to object and invokes method

```
int main (int argc, char *argv[])
{
  // Use a factory to bind to any quoter.
  Stock::Quoter_Factory_var qf =
    bind_service<Stock::Quoter_Factory>
      ("My_Quoter_Factory", argc, argv);
  if (CORBA::is_nil (qf)) return -1;

  Stock::Quoter_var quoter; // Manages refcounts.

  const char *stock_name = "ACME ORB Inc.";

  try {   // Bind to a quoter and make call.
    quoter = qf->create_quoter ("My_Quoter");
    CORBA::Long value = quoter->get_quote (stock_name);
    cout << stock_name << " = " << value << endl;
  } catch (Stock::Invalid_Stock &) {
    cerr << stock_name << " not a valid stock" << endl;
  } catch (...) { /* Handle exception... */ }

  qf->destroy_quoter (quoter);
  // Destructors of *_var proxies release memory.
}
```


Obtaining an Object Reference

- *e.g.*, using the COS Naming Service

```
static CosNaming::NamingContext_ptr name_context = 0;

template <class T> T *
bind_service (const char *name,
              int argc, char *argv[]) {
  CORBA::Object_var obj;
  if (name_context == 0) { // "First time in" check.
    // Get reference to name service.
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, 0);

    obj = orb->resolve_initial_references ("NameService");

    name_context =
      CosNaming::NamingContext::_narrow (obj);
    if (CORBA::is_nil (name_context)) return 0;
  }
  CosNaming::Name svc_name;
  svc_name.length (1); svc_name[0].id = name;

  // Find object reference in the name service.
  obj = name_context->resolve (svc_name);

  // Narrow to the T interface and away we go!
  return T::_narrow (obj);
}
```

Server Activation

- If the server isn't running when a client invokes a method on an object it manages, the ORB will automatically start the server

- Servers must be registered with the ORB in the "Implementation Repository"

  – *e.g.*, in Orbix

    % putit Quoter_Factory /usr/svcs/Quoter/quoter.exe

- Server(s) may be installed on any machine

- Clients may bind to an object in a server by using the Naming Service or by explicitly identifying the server

Server Activation Modes

- An idle server will be automatically launched when one of its objects is invoked

- There are four server activation modes

  1. *Shared* → one process per-object per-host
  2. *Unshared* → each individual object gets its own process
  3. *Per-method call* → each method call gets its own process
  4. *Persistent* → launched "manually"

Coping with Changing Requirements

- *New features*
  - Format changes to extend functionality
  - New interfaces and operations
- *Improving existing features*
  - Server location independence (requires smart ORB)
  - Batch requests

New Formats

- *e.g.*, percentage that stock increased or decreased since start of trading day, volume of trades, etc.

```
module Stock
{
  // ...

  interface Quoter
  {
    long get_quote (in string stock_name,
    out double percent_change,
    out long trading_volume)
      raises (Invalid_Stock);
  };
};
```

- Note that even making this simple change would involve a great deal of work for a sockets-based solution...

Adding Features Unobtrusively

- Interface inheritance allows new features to be added without breaking existing interfaces

```
module Stock
{
  // ...
  interface Quoter { /* ... */ };

  interface Stat_Quoter
    : Quoter // a Stat_Quoter IS-A Quoter
  {
    void get_stats (in string stock_name,
    out double percent_change,
    out long trading_volume)
      raises (Invalid_Stock);
  };
};
```

- Note that there are no changes to the existing `Quoter` interface

New Interfaces and Operations

- *e.g.*, adding a trading interface

```
module Stock {
  // interface Quoter_Factory and Quoter
  // Same as before.

  interface Trader {
    void buy  (in string name,
               inout long num_shares,
               in long max_value)
      raises (Invalid_Stock);

    void sell (in string name,
               inout long num_shares,
               in long min_value)
      raises (Invalid_Stock);
  };

  interface Trader_Factory { /* ... */ };
};
```

- Multiple inheritance is also useful to define a full service broker:

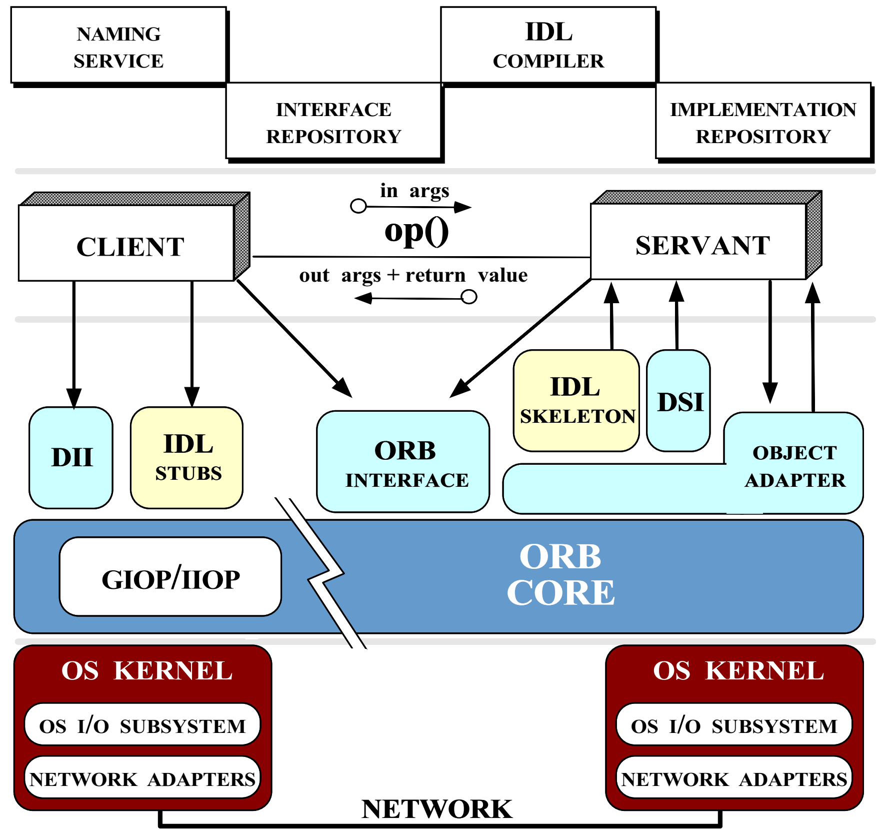
```
interface Broker : Stat_Quoter, Trader {};
```

Batch Requests

- Improve performance for multiple queries or trades

```
interface Batch_Quoter
  : Stat_Quoter // A Batch_Quoter IS-A Stat_Quoter
{
  typedef sequence<string> Names;
  struct Stock_Info {
    string name;
    long value;
    double change;
    long volume;
  };
  typedef sequence<Stock_Info> Info;

  exception No_Such_Stock {
    Names stock; // List of invalid stock names
  };

  void batch_quote (in Names stock_names,
                    out Info stock_info)
    raises (No_Such_Stock);
};
```


CORBA ORB Architecture



CORBA Components

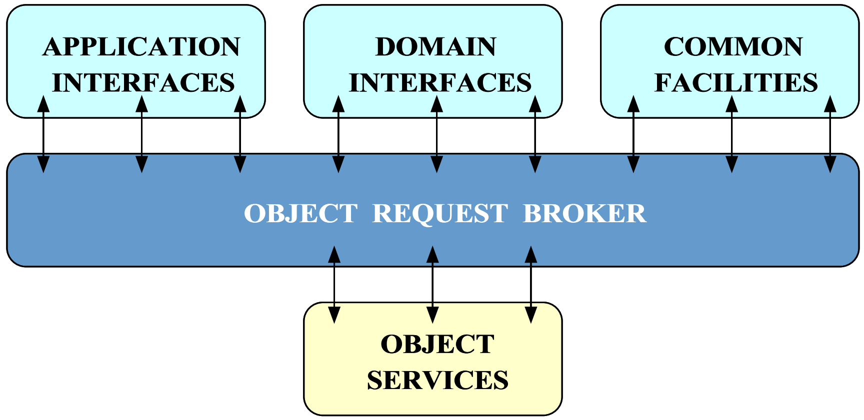
- The CORBA specification is comprised of several parts:

  1. An Object Request Broker (ORB) Core
  2. An Interoperability Spec (GIOP and IIOP)
  3. An Interface Definition Language (IDL)
  4. Programming language mappings for IDL
  5. A Static Invocation Interface (SII)
  6. A Dynamic Invocation Interface (DII)
  7. A Dynamic Skeleton Interface (DSI)
  8. Portable Object Adapter (POA)
  9. Interface and implementation repositories

- Other documents from OMG describe common object services built upon CORBA (CORBAservices)

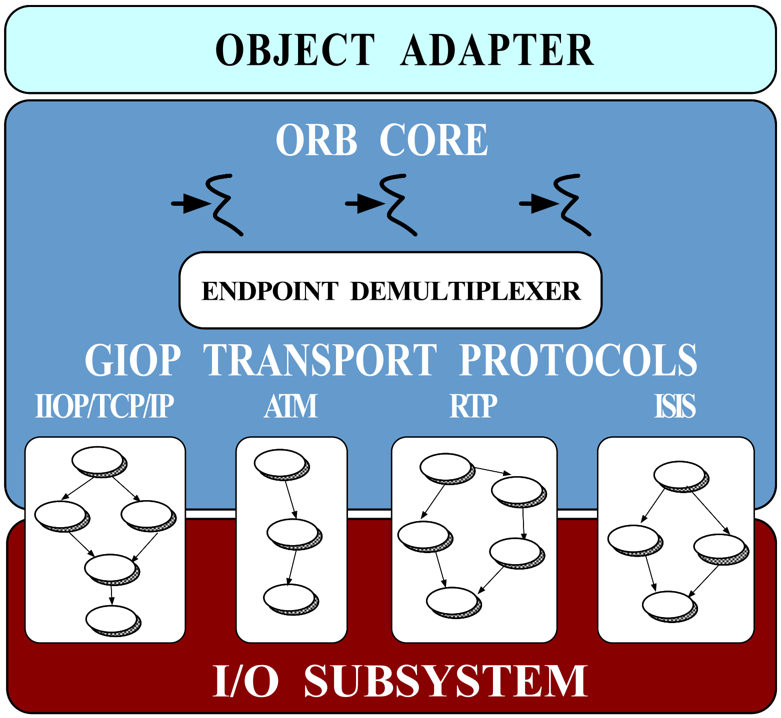
  – *e.g., Event services, Name services, Lifecycle services*

OMA Reference Model Interface Categories



- The Object Management Architecture (OMA) Reference Model describes the interactions between various CORBA components and layers

ORB Core



- Provides basic concurrency and communication mechanisms

CORBA Interoperability Protocols

- General Inter-ORB Protocol (GIOP)
  - Specifies request format and transmission protocol that enables ORB-to-ORB interoperability
- Internet Inter-ORB Protocol (IIOP)
  - Specifies a standardized interoperability protocol for the Internet
  - Works directly over TCP/IP, no RPC necessary
- Environment-specific inter-ORB protocols (ESIOPs)
  - *e.g.*, DCE

GIOP Overview

- *Common Data Representation* (CDR)
  - Transfer syntax mapping OMG-IDL data types into a bi-canonical low-level representation
    - Supports variable byte ordering and aligned primitive types
- *Message transfer*
  - Request multiplexing
    - *i.e.*, multiple clients can share a connection to an ORB
  - Ordering constraints are minimal
    - *i.e.*, can be asynchronous
- *Message formats*
  - Client: `Request`, `CancelRequest`, `LocateRequest`
  - Server: `Reply`, `LocateReply`, `CloseConnection`
  - Both: `MessageError`

GIOP Overview (cont'd)

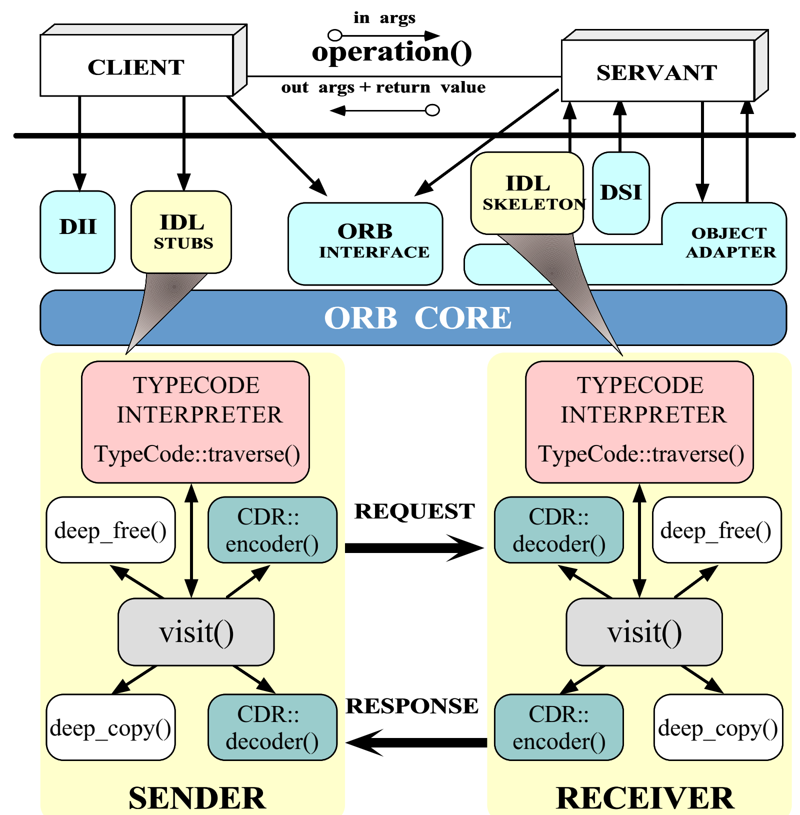
- GIOP module

```
module GIOP {
  enum MsgType {
    Request, Reply, CancelRequest, LocateRequest,
    LocateReply, CloseConnection, MessageError
  };
  struct MessageHeader {
    char magic[4];
    Version GIOP_version;
    boolean byte_order;
    octet message_type;
    unsigned long message_size;
  };

  struct requestHeader {
    IOP::ServiceContextList service_context;
    unsigned long request_id;
    boolean response_requested;
    sequence<octet> object_key;
    string operation;
    Principal requesting_principal;
  };

  // ...
```

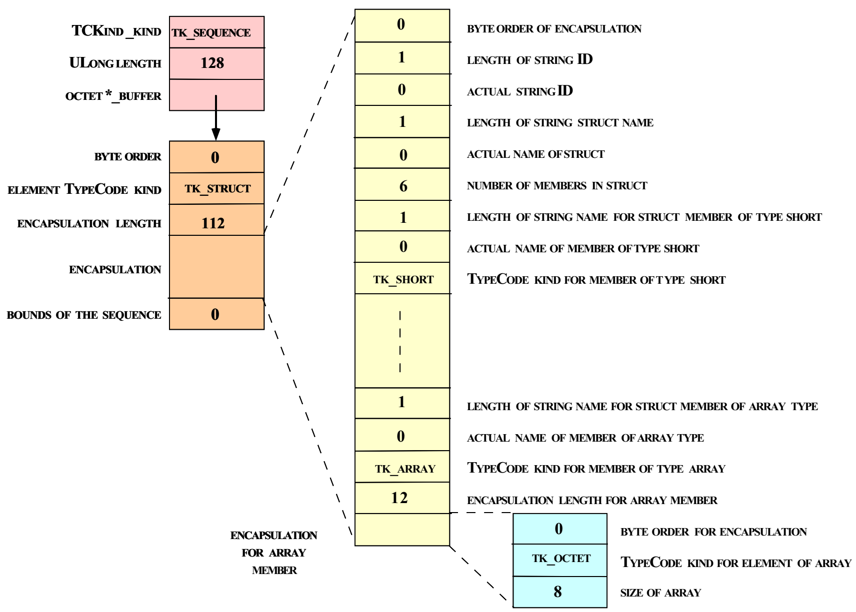
IIOP Overview

- IIOP Adds to GIOP semantics for TCP/IP connection management
- IIOP bundled with Netscape 4.0
- Inter-ORB Engine available from SunSoft
  - ftp://ftp.omg.org/pub/interop/iiop.tar.Z
- TAO is originally based on SunSoft IIOP
  - However, TAO adds *many* enhancements and optimizations

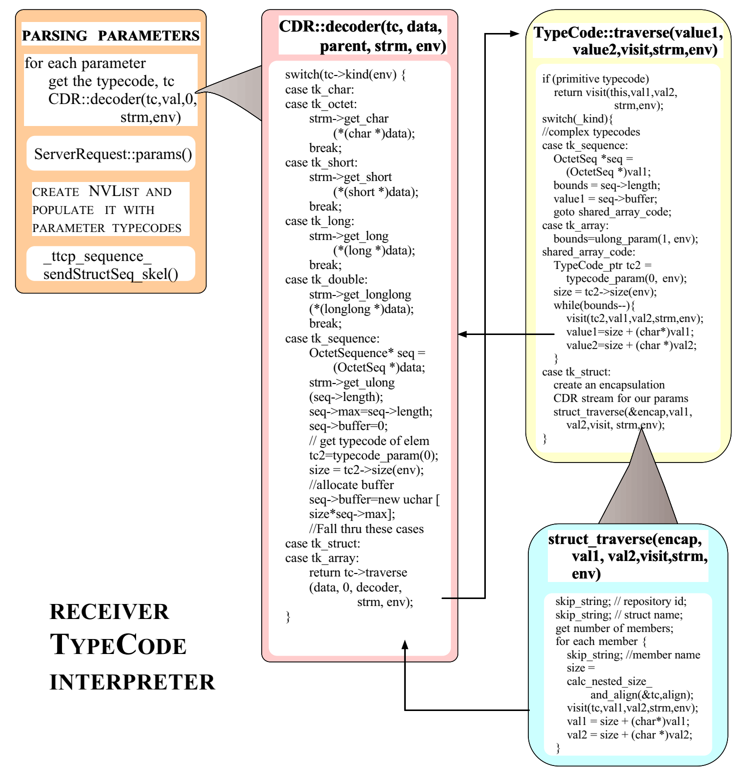
Design of TAO's IIOP Protocol Engine



TypeCode Layout for BinStruct Sequence



TypeCode Interpreter Structure



Interface Definition Language (IDL)

- Developing flexible distributed applications on heterogeneous platforms requires a strict separation of *interface* from *implementation(s)*

- Benefits of using an IDL
  - *Ensure platform independence*
    - *e.g.*, Windows NT to UNIX
  - *Enforce modularity*
    - *e.g.*, must separate concerns
  - *Increase robustness*
    - *e.g.*, reduce opportunities for network programming errors
  - *Enable language independence*
    - *e.g.*, COBOL to C++

Related IDLs

- Many IDLs are currently available, *e.g.*,
  - OSI ASN.1
  - OSI GDMO
  - SNMP SMI
  - DCE IDL
  - Microsoft's IDL (MIDL)
  - OMG IDL
  - ONC's XDR

- However, many of these are *procedural* (rather than *object-based* or *object-oriented*) IDLs
  - Procedural IDLs are more complicated to extend and reuse since they don't support inheritance

CORBA Interface Definition Language (IDL)

- OMG IDL is an object-oriented interface definition language
  - Used to specify interfaces containing *operations* and *attributes*
  - OMG IDL support interface inheritance (both single and multiple inheritance)
- OMG IDL is designed to map onto multiple programming languages
  - *e.g.*, C, C++, Smalltalk, COBOL, Modula 3, DCE, Java, etc.
- OMG IDL is similar to Java `interfaces` and C++ "abstract classes"

Application Interfaces

- Interfaces described using OMG IDL may be application-specific, *e.g.*,
  - *Databases*
  - *Spreadsheets*
  - *Spell checker*
  - *Network manager*
  - *Air traffic control*
  - *Documents*
  - *Medical imaging systems*
- Objects may be defined at any level of granularity
  - *e.g.*, from fine-grained GUI objects to multi-megabyte multimedia "Blobs"

OMG IDL Features

- OMG IDL is similar to Java `interfaces` or C++ abstract base classes
  - Note, it is not a complete programming language, it only defines interfaces

- OMG IDL supports the following features:

  * `modules`
  * **`interfaces`**
  * Operations
  * Attributes
  * Inheritance
  * Basic types (*e.g.*, **double**, **long**, **char**, etc).
  * Arrays
  * **`sequence`**
  * **`struct, enum, union, typedef`**
  * **`consts`**
  * **`exceptions`**

OMG IDL vs. C++

- Differences from C++

  * No data members
  * No pointers
  * No constructors or destructors
  * No overloaded methods
  * No **int** data type
  * Contains parameter passing modes
  * Unions require a tag
  * Different String type
  * Different Sequence type
  * Different exception interface
  * No templates
  * No control constructs
  * **oneway** call semantics
  * **readonly** keyword
  * Can pass "contexts" in operations

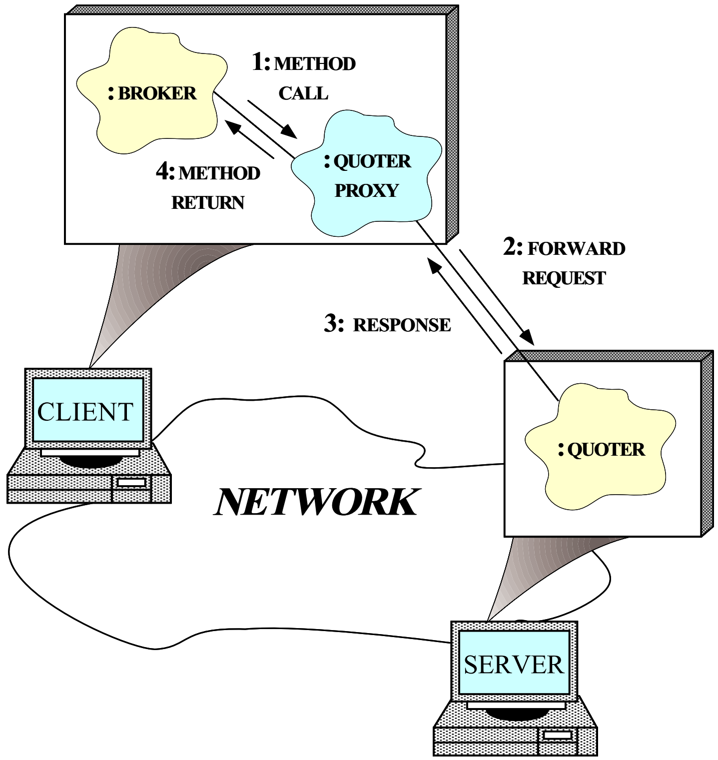
Static Invocation Interface (SII)

- The most common way of using OMG IDL involves the "Static Invocation Interface" (SII)

- In this case, all the methods are specified in advance and are known to the client and the server via *proxies*
  - Proxies are also known as *surrogates*

- The primary advantage of the SII is its simplicity, typesafety, and efficiency

Proxy Pattern



- *Intent*: provide a surrogate for another object that controls access to it

Dynamic Invocation Interface (DII)

- A less common programming API is the "Dynamic Invocation Interface" (DII)
  - Enables clients to invoke methods on objects that aren't known until run-time
    - *e.g.*, MIB browsers
  - Allows clients to "push" arguments onto a request stack and identify operations via ASCII name
  - Type-checking via meta-info in "Interface Repository"
- The DII is more flexible than the SII
  - *e.g.*, it supports *deferred synchronous* invocation
- However, the DII is also more complicated, less typesafe, and inefficient

Dynamic Skeleton Interface (DSI)

- The "Dynamic Skeleton Interface" (DSI) provides analogous functionality for the server-side that the DII provides on the client-side
- It is defined in CORBA 2.x primarily for using building ORB "Bridges"
- The DSI lets server code handle arbitrary invocations on CORBA objects

Object References

- An "object reference" is an opaque handle to an object

- Object references may be passed among processes on separate hosts

  – The underlying CORBA ORB will correctly convert object references into a form that can be transmitted over the network

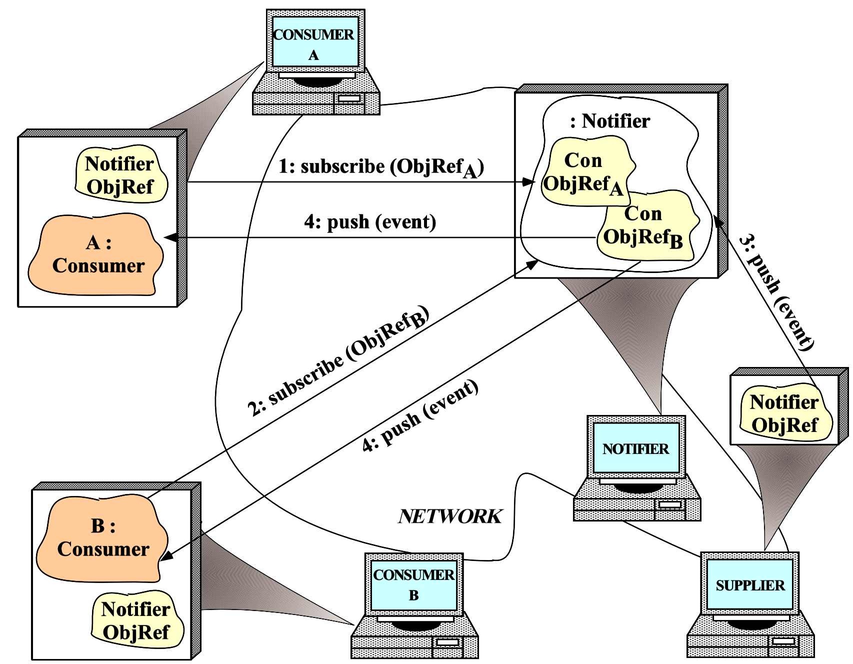
  – The ORB passes the receiver's implementation a pointer to a proxy in its own address space

    * This proxy refers to the object's implementation

- Object references are a powerful feature of CORBA

  – *e.g.*, supports *peer-to-peer* interactions and *distributed callbacks*

Using Object References



- Passing object references is useful to implement a distributed event notification mechanism

Event Receiver Interface

- An `Consumer` is called back by the `Notifier`
  - Note that all operations are **oneway** to avoid blocking

```
struct Event {
  string tag_; // Used for filtering.
  any value_; // Event contents.
};

interface Consumer
{
  // Inform the Consumer
  // event has occurred.
  oneway void push (in Event event);

  // Disconnect the Consumer from the
  // Notifier, giving it the <reason>.
  oneway void disconnect (in string reason);
};
```


Notifier Interface

- A `Notifier` publishes `Events`

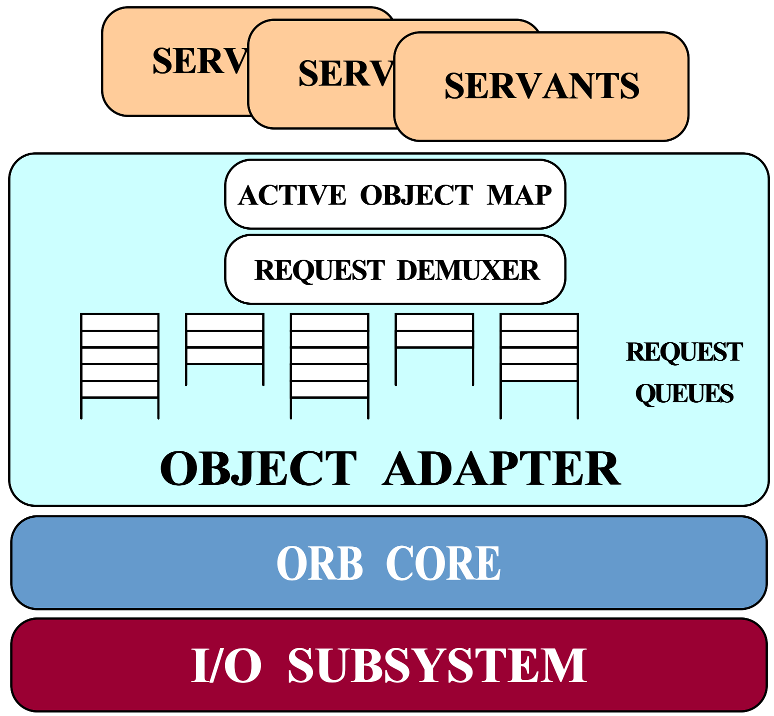
```
interface Notifier
{
  // Subscribe the Consumer to
  // receive events that match filtering_criteria
  // applied by the Notifier.
  oneway void subscribe (in Consumer consumer,
 in string filtering_criteria);

  // Unsubscribe the Consumer.
 oneway void unsubscribe (in Consumer consumer);

  // Push the Event to all the consumers
  // who have subscribed and who match the
  // filtering criteria.
  oneway void push (in Event event);
};
```

Notifier Implementation

- The `Notifier` maintains a table of object references to `Consumers`

```
class My_Notifier // C++ pseudo-code
{
public:
  void subscribe (Consumer *consumer,
  const char *fc) {
    insert <consumer> into <consumer_set_> with <fc>.
  }

  void unsubscribe (Consumer *consumer) {
    remove <consumer> from <consumer_set_>.
  }

  void push (const Event &event) {
    foreach <consumer> in <consumer_set_> loop
      if (event.tag_ matches <consumer>.filter_criteria)
<consumer>.push (event);
    end loop;
  }

private:
  // e.g., use an STL set.
  set <Consumer *> consumer_set_;
};
```


Object Adapter



- Provide services that map object references and requests to servants

Portable Object Adapter (cont'd)

- Design goals
  - Object implementations are portable between ORBs
  - Objects with persistent identities
    - ∗ Object implementations span multiple server lifetimes
  - Transparent activation of objects
  - Single servant can support multiple object identities
  - Multiple instances of the POA in a server
  - Transient objects with minimal programming effort and overhead
  - Implicit activation of servants with POA-allocated Object Ids
  - POA behavior is dictated by creation policies
  - Object implementations can inherit from static skeleton classes, or a DSI implementation

POA Architecture

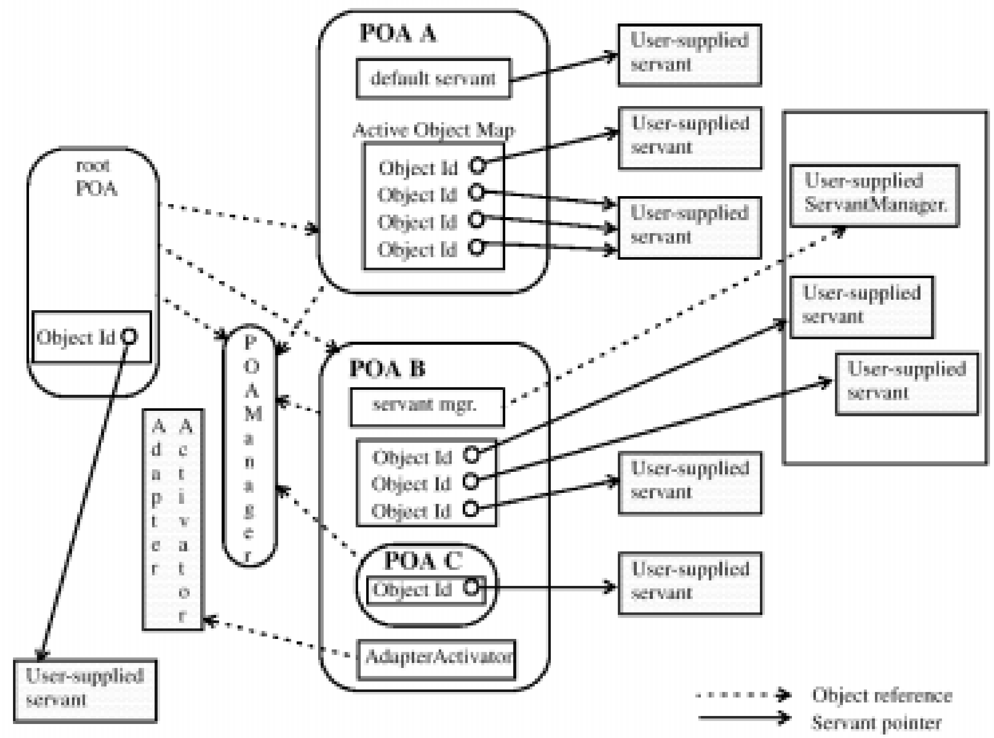


Figure 3-2 POA Architecture

POA Components

- *Client*: Makes requests on an object through one of its references

- *Server*: Computational context for an object implementation
  - Generally, a server corresponds to a process
  - Client and server are roles – a program maybe be both

- *Object*: A programming entity with an identity, an interface, and an implementation

- *Servant*: A programming language entity that implements requests on one or more objects

POA Components (cont'd)

- *Object Id*: A value that is used by the POA and by the implementation to identify a particular CORBA object
  - Object Id values may be assigned by the POA, or by the implementation
  - Object Id values are hidden from clients, encapsulated by references
  - Object Ids have no standard form; they are managed by the POA as uninterpreted octet sequences

- *Object Reference*: Encapsulates an Object Id and a POA identity

- *POA*: A namespace for Object Ids and a namespace for child POAs
  - Nested POAs form a hierarchical name space for objects within a server

POA Components (cont'd)

- *Policy*: Specifics the characteristics of the POA

- *POA Manager*: Encapsulates the processing state of associated POAs.
  - Used to queue or discard requests for the associated POAs
  - Also used to deactivate the POAs

- *Servant Manager*: Callback object used to activate and deactivate servants on demand
  - Two kinds: `ServantActivator` and `ServantLocator`

- *Adapter Activator*: Callback object used when a request is received for a child POA that does not currently exist
  - The adapter activator can then create the required POA on demand

POA Examples

- Getting the root POA

```
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
CORBA::Object_ptr pfobj =
  orb->resolve_initial_references ("RootPOA");
PortableServer::POA_var rootPOA;
rootPOA = PortableServer::POA::_narrow (pfobj);
```

- Creating a POA

```
CORBA::PolicyList policies (2);

policies[0] = rootPOA->create_thread_policy
  (PortableServer::ThreadPolicy::ORB_CTRL_MODEL);
policies[1] = rootPOA->create_lifespan_policy
  (PortableServer::LifespanPolicy::TRANSIENT);

PortableServer::POA_ptr poa =
  rootPOA->create_POA
             ("my_poa",
              PortableServer::POAManager::_nil (),
              policies);
```

POA Examples (cont'd)

- Explicit activation with POA-assigned Object Ids

```
// IDL
interface Foo
{
  long doit ();
};

class POA_Foo : public ServantBase
{
  virtual CORBA::Long doit (void) = 0;
};

class My_Foo_Servant : public POA_Foo
{
  virtual Long doit (void) { return 42; }
};

My_Foo_Servant *afoo = new My_Foo_Servant;
PortableServer::ObjectId_var oid =
  poa->activate_object (afoo);

poa->the_POAManager ()->activate ();
orb->run ();
```


POA Examples (cont'd)

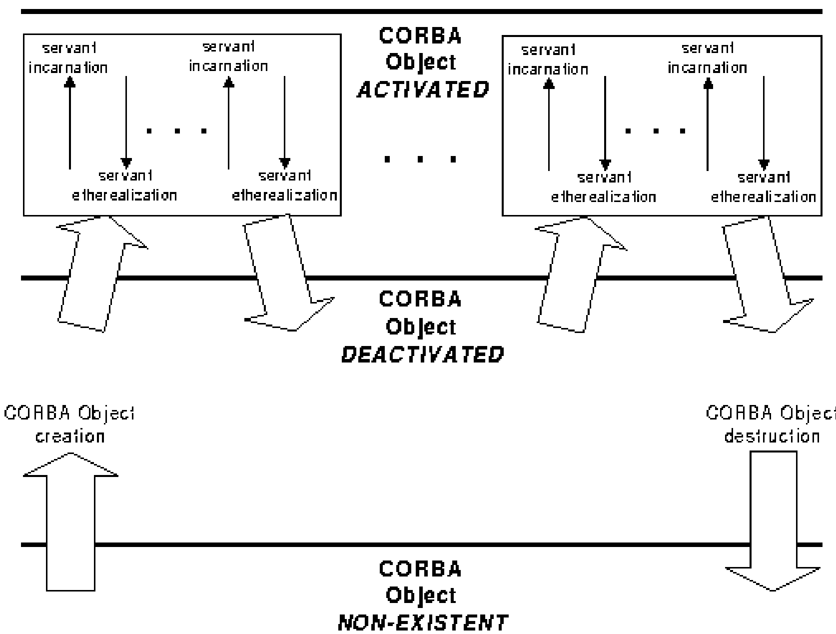
- Explicit activation with user-assigned Object Ids

```
My_Foo_Servant *afoo = new My_Foo_Servant;

PortableServer::ObjectId_var oid =
  PortableServer::string_to_ObjectId ("myFoo");
poa->activate_object_with_id (oid.in (), afoo);
```

- Creating references before activation

```
PortableServer::ObjectId_var oid =
  PortableServer::string_to_ObjectId ("myFoo");
CORBA::Object_var obj =
  poa->create_reference_with_id (oid.in (),
                                 "IDL:Foo:1.0");

Foo_var foo = Foo::_narrow (obj);

// ...later...

My_Foo_Servant *afoo = new My_Foo_Servant;
poa->activate_object_with_id (oid.in (),
                              afoo);
```

Request Lifecycle for POA



Examples (cont'd)

- Servant Manager Definition and Creation

```
// Skeleton class
namespace POA_PortableServer
{
  class ServantActivator : public virtual ServantManager
  {
    virtual ~ServantActivator ();
    virtual Servant incarnate (const ObjectId &,
                               POA_ptr poa) = 0;
    virtual void etherealize
      (const ObjectId&,
       POA_ptr poa,
       Servant,
       Boolean remaining_activations) = 0;
  };
}
```

Examples (cont'd)

- Servant Manager Definition and Creation

```
// Implementation class
class My_Foo_Servant_Activator
  : public POA_PortableServer::ServantActivator
{
  Servant incarnate (const ObjectId &oid,
                     POA_ptr poa)
  {
    String_var s =
      PortableServer::ObjectId_to_string (oid);

    if (strcmp (s, "myFoo") == 0)
      return new My_Foo_Servant;
    else
      throw CORBA::OBJECT_NOT_EXIST;
  }

  void etherealize (const ObjectId &oid,
                    POA_ptr poa,
                    Servant servant,
                    Boolean remaining_activations)
  {
    if (remaining_activations == 0)
      delete servant;
  }
};
```


Examples (cont'd)

- Object activation on demand

```
PortableServer::ObjectId_var oid =
  PortableServer::string_to_ObjectId ("myFoo");
CORBA::Object_var obj =
  poa->create_reference_with_id (oid, "IDL:foo:1.0");

My_Foo_Servant_Activator *foo_im =
  new My_Foo_Servant_Activator;

ServantActivator_var im_ref = foo_im->_this ();
poa->set_servant_manager (im_ref);
poa->the_POAmanager ()->activate ();

orb->run ();
```

- See POA specification for more examples on:
  - One Servant for all Objects
  - Single Servant, many objects and types, using DSI

Evaluating CORBA

- *Learning curve*
- *Interoperability*
- *Portability*
- *Feature Limitations*
- *Performance*

Learning Curve

- CORBA introduces the following:
  1. *New concepts*
     - *e.g.*, object references, proxies, and object adapters
  2. *New components and tools*
     - *e.g.*, interface definition languages, IDL compilers, and object-request brokers
  3. *New features*
     - *e.g.*, exception handling and interface inheritance
- Time spent learning this must be amortized over many projects

Interoperability

- The first CORBA 1 spec was woefully incomplete with respect to interoperability
  - The solution was to use ORBs provided by a single developer...
- CORBA 2.x defines a useful interoperability specification
  - Later extensions deal with portability issues for server-side
    - *i.e.*, the POA spec
  - Most ORB implementations now support IIOP or GIOP robustly...
- However, higher-level CORBA services aren't covered by ORB interoperability spec...

Portability

- To improve portability, OMG CORBA 2.1 + the POA spec. standardizes
  - IDL-to-C++ language mapping
  - Naming service, event service, lifecycle service
  - ORB initialization service
  - Portable Object Adapter API
  - Servant mapping
- Vendors are increasingly supporting these features
- Porting applications from ORB to ORB will be limited, however, until conformance tests become common-place
  - There is an RFP for this.

Feature Limitations

- Standard CORBA doesn't really address key "inherent" complexities of distributed computing, *e.g.*,
  - *Latency*
  - *Fault tolerance*
  - *Causal ordering*
  - *Deadlock*
- To some extent, it does address *service partitioning*
  - But you must be very careful in practice...

Feature Limitations (cont'd)

- CORBA does not allow objects to be passed by value
  - However, the OMG is working on an RFP
- Instead, it only supports the following semantics:
  - Object references are passed-by-reference
    * However, all method invocations on the remote host are routed back to the originator host
  - C-style structures and discriminated unions may be passed-by-value
    * However, these structures and unions do *not* contain any methods
- Support for passing objects by value must be hand-crafted on top of CORBA using "factories"

Feature Limitations (cont'd)

- CORBA doesn't yet define support for asynchronous or non-blocking operations
  - This is left as "quality of service" of the implementation
  - OMG is working on an async messaging RFP
- Versioning is supported in IDL via "pragmas"
  - Unlike Sun RPC or DCE, which include in language
- Current implementations of CORBA lack efficient support for bulk data transfer

Performance Limitations

- Performance may not be as good as handcrafted code for some applications
  - Due to the following
    * Additional remote invocations for naming
    * Marshalling/demarshalling overhead
    * Data copying
    * Memory management
    * Demultiplexing
- Typical trade-off between extensibility, robustness, maintainability → *micro-level efficiency*
- Note that a well-crafted ORB may be able to automatically optimize *macro-level efficiency*

CORBA Implementations

- Many ORBs are now available
  - Orbix from IONA
  - Sun/Chorus COOL
  - Visibroker from Visigenic/Borland
  - ORB Plus from HP
  - PowerBroker/CORBAPlus from Expersoft
  - TAO from Washington University
- In theory, CORBA facilitates vendor-independent and platform-independent application collaboration
  - In practice, heterogeneous ORB interoperability and portability still an issue...

CORBA Services

- Other OMG documents (*e.g.*, COSS) specify higher level services
  - *Naming service*
    - Mapping of convenient object names to object references
  - *Event service*
    - Enables decoupled, asynchronous communication between objects
  - *Lifecycle service*
    - Enables flexible creation, copy, move, and deletion operations via factories
- Other CORBA services include transactions, trading, relations, security, etc.

Summary of CORBA Features

- CORBA specifies the following functions to support an Object Request Broker (ORB)
  - Interface Definition Language (IDL)
  - A mapping from IDL onto C and C++
  - A Static Invocation Interface, used to compose method requests via proxies
  - A Dynamic Invocation Interface, used to compose method requests at run-time
  - Interface and Implementation Repositories containing meta-data queried at run-time
  - The Portable Object Adapter (POA), allows service programmers to interface their code with an ORB

Concluding Remarks

- Additional information about CORBA is available on-line at the following WWW URLs (prefix http:// before each of these)
  - Doug Schmidt's CORBA page (contains many articles on CORBA)
    * www.cs.wustl.edu/~schmidt/corba.html
  - OMG's WWW Page
    * www.omg.org/
  - DSTC's OMG Page
    * www.dstc.edu.au/AU/research_news/omg/corba.html
  - LANL's OMG Page
    * www.acl.lanl.gov/CORBA

TAO Development and CM

- Two models for source code sharing
  - Access to our CVS repository
    - RiverAce uses this model successfully
  - Use CVS's **`export`** feature
    - May be better suited to one-way flow
  - Use dual, synchronized repositories
    - Can be very complicated

Tuning TAO's Behavior

- Command line arguments
  - *e.g.*, **`-ORBhost`**, **`-ORBpreconnect`**, etc.
  - typically alters *external* behaviors
- Service Configurator's **`svc.conf`** service arguments
  - *e.g.*, **`-ORBresources`**, **`-ORBconcurrency`**, etc.
  - typically alters *internal* behaviors, *i.e.*,,

    behavior of internal ORB components

Tweaking the Internals

- `TAO_Resource_Factory`
  - Controls how key ORB resources are shared across threads
  - *e.g.*,, `-ORBresources` with `tss` or `global`
- `TAO_Default_Server_Strategy_Factory`
  - Controls demultiplexing strategy (`-ORBdemuxstrategy`), request/connection service concurrency (`-ORBconcurrenc`
    etc.
- `TAO_Default_Client_Strategy_Factory`
  - This is currently empty and may go away
- Complete documentation in `$TAO_ROOT/docs/Options.`