Developing Distributed Object Computing Applications with CORBA

Douglas C. Schmidt

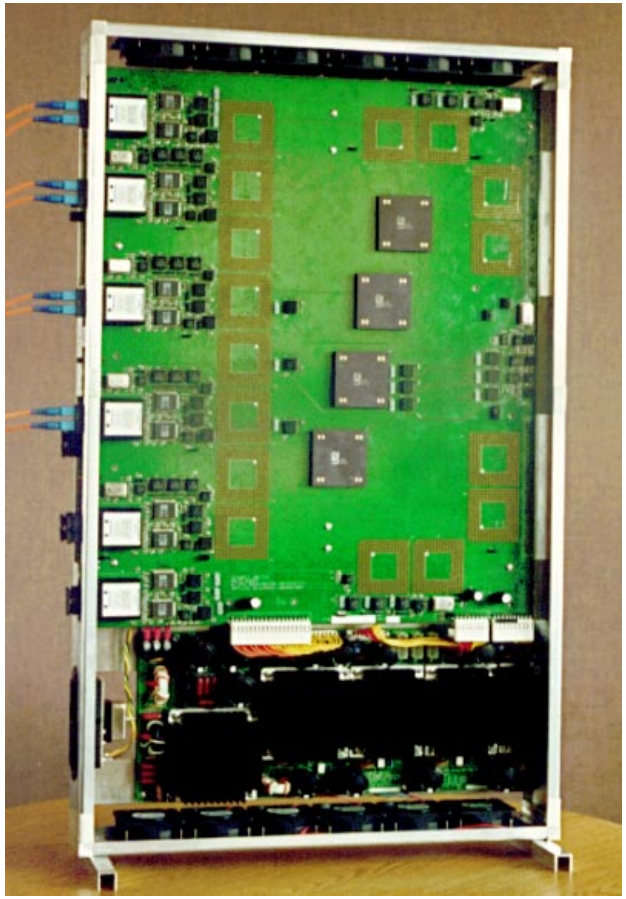
Professor
d.schmidt@vanderbilt.edu
www.cs.wustl.edu/∼schmidt/

Department of EECS
Vanderbilt University
(615) 343-8197

Sponsors

NSF, DARPA, ATD, BAE Systems, BBN, Boeing, Cisco, Comverse, GDIS, Experian, Global MT, Hughes, Kodak, Krones, Lockheed, Lucent, Microsoft, Mitre, Motorola, NASA, Nokia, Nortel, OCI, Oresis, OTI, Qualcomm, Raytheon, SAIC, SAVVIS, Siemens SCR, Siemens MED, Siemens ZT, Sprint, Telcordia, USENIX

Motivation: the Distributed Software Crisis



Symptoms

- ***Hardware*** gets smaller, faster, cheaper
- ***Software*** gets larger, slower, more expensive

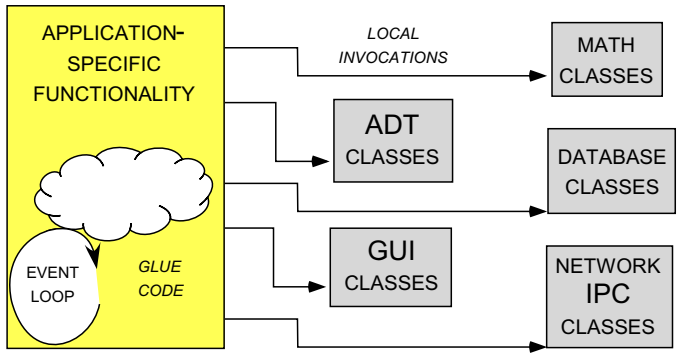
Culprits

- ***Inherent*** and ***accidental*** complexity

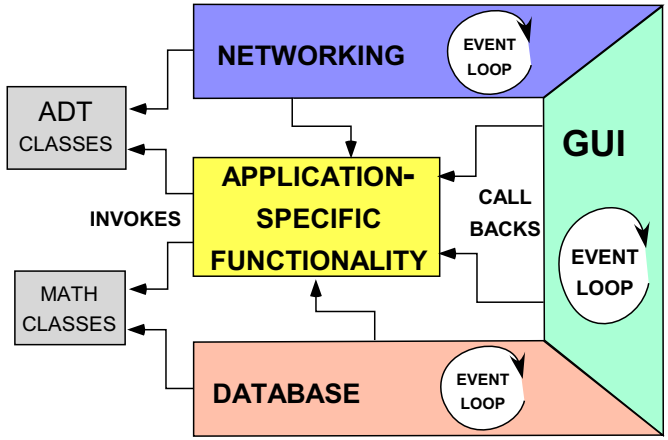
Solution Approach

- ***Components, Frameworks, Patterns, & Architecture***

Techniques for Improving Software



(A) CLASS LIBRARY ARCHITECTURE

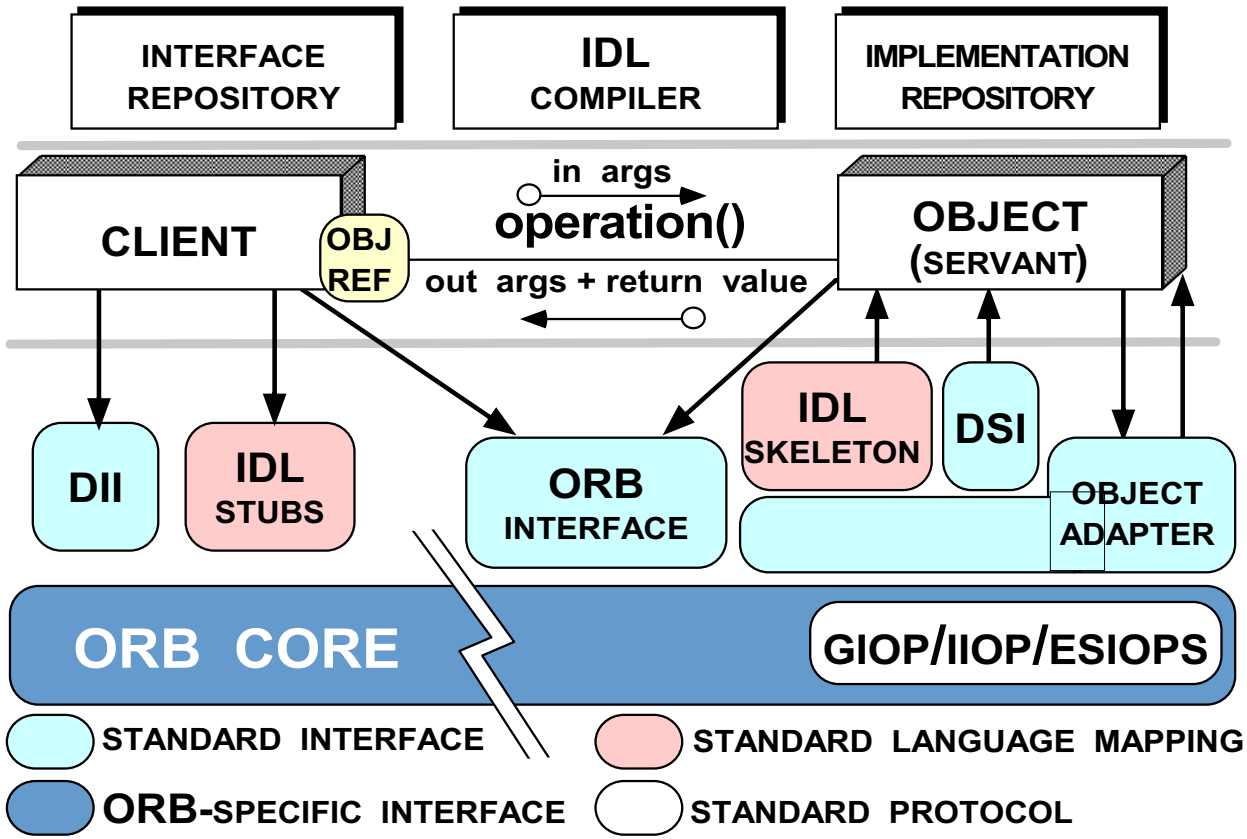


(B) FRAMEWORK ARCHITECTURE

Proven solutions →

- *Components*
  - Self-contained, "pluggable" ADTs
- *Frameworks*
  - Reusable, "semi-complete" applications
- *Patterns*
  - Problem/solution/context
- *Architecture*
  - Families of related patterns and components

Overview of CORBA Middleware Architecture



www.cs.wustl.edu/~schmidt/corba.html

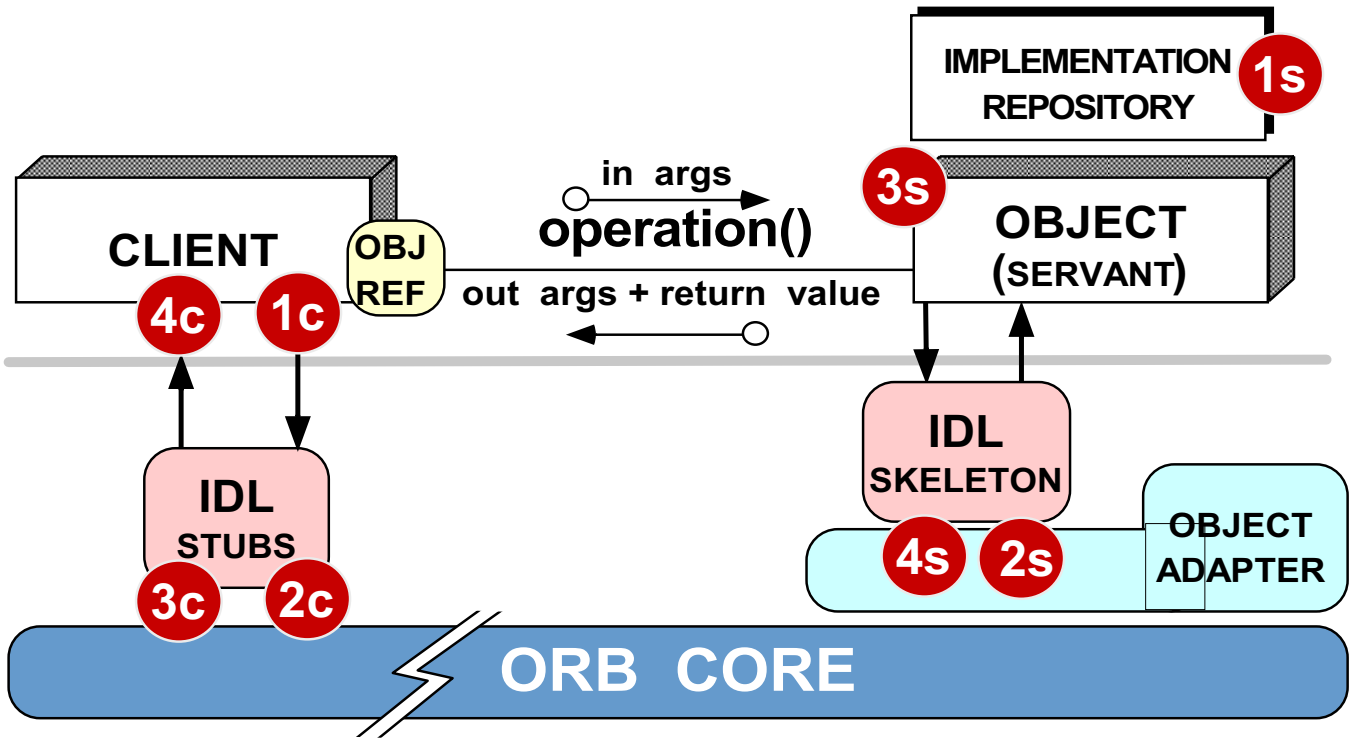
Goals of CORBA

- Simplify distribution by automating
  - Object location & activation
  - Parameter marshaling
  - Demultiplexing
  - Error handling
- Provide foundation for higher-level services

Key CORBA Concepts

- Object reference: A strongly-typed opaque handle that identifies an object's location
- Client: Makes requests on an object via one of its references
- Server: Computational context (*e.g.*, process) for objects/servants
  - Client and server are "roles" - a program can play both roles
- Stub: A proxy that converts method calls into messages
- Skeleton: An adapter that converts messages back into method calls
- Object: A CORBA programming entity with an identity, an interface, and an implementation
- Servant: A programming language entity that implements requests on one or more objects
- POA: A container for objects/servants in a server
- ORB Core: Message-passing infrastructure

CORBA Twoway Processing Steps



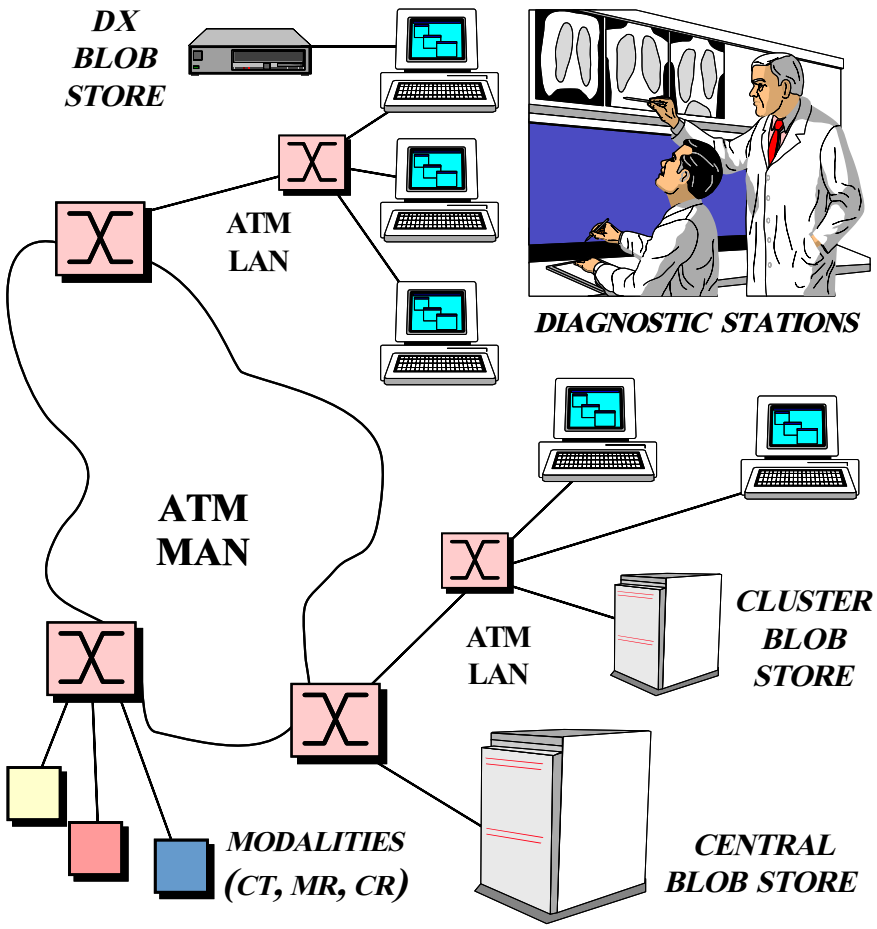
Client processing steps

1c Locate target object
2c Sent request message to server
3c Wait for request to complete
4c Return control to client

Server processing steps

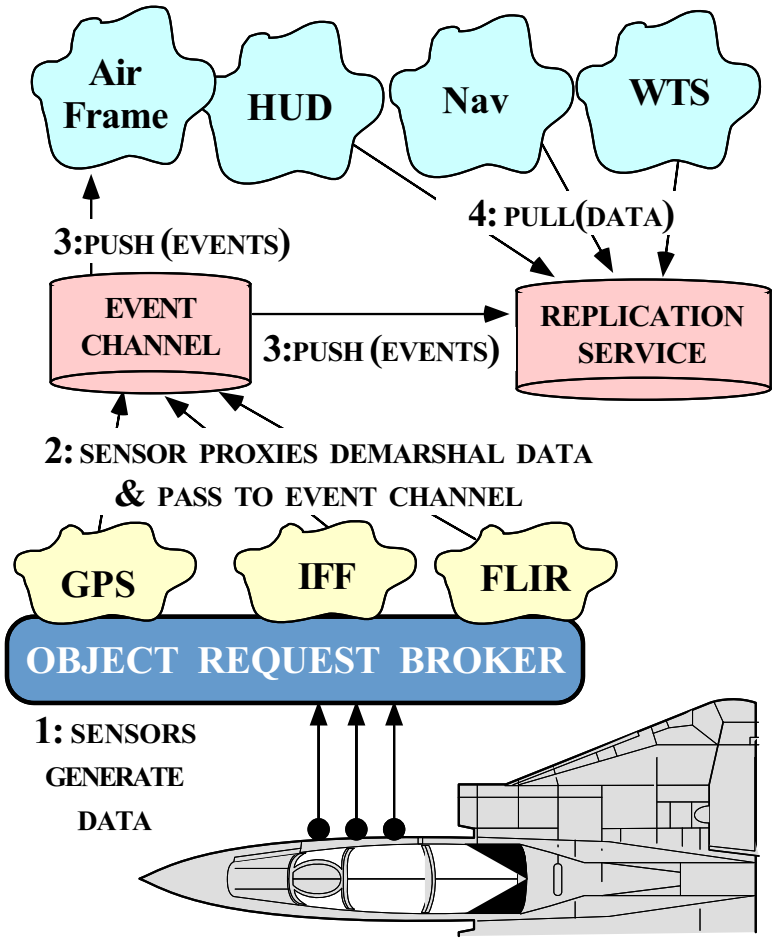
1s Activate server (if necessary)
2s Activate object's servant (if necessary)
3s Process request
4s Return result or exception

Applying CORBA to Medical Imaging



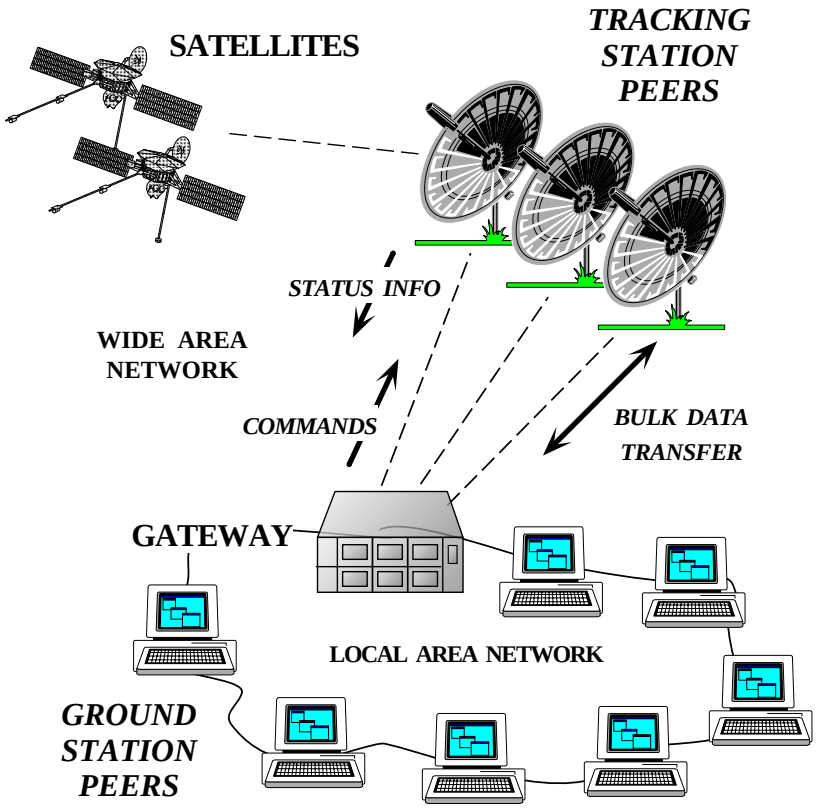
- **Domain Challenges**
  - Large volume of "Blob" data
    - *e.g.*, 10 to 40 Mbps
  - "Lossy compression" isn't viable
  - Prioritization of requests
- **URLs**
  - ~schmidt/PDF/COOTS-96.pdf
  - ~schmidt/PDF/av_chapter.pdf
  - ~schmidt/NMVC.html

Applying CORBA to Real-time Avionics



- **Domain Challenges**
  - Real-time periodic processing
  - Complex dependencies
  - Very low latency
- **URLs**
  - ~schmidt/PDF/JSAC-98.pdf
  - ~schmidt/TAO-boeing.html

Applying CORBA to Global PCS



- **Domain Challenges**
  - Long latency satellite links
  - High reliability
  - Prioritization
- **URL**
  - ~schmidt/PDF/TAPOS-00.pdf

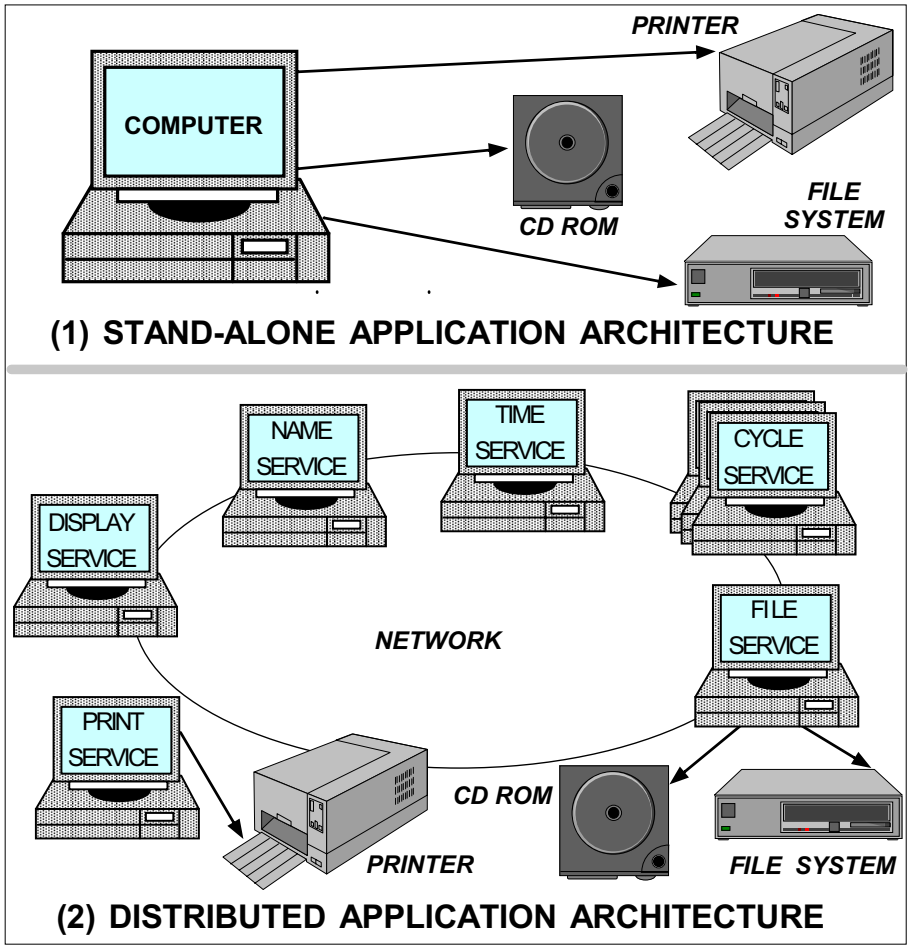
Tutorial Outline

- Motivation
- Example CORBA Applications
- Using CORBA to Cope with Changing Requirements
- Detailed Overview of CORBA Architecture and Features
- Evaluations and Recommendations

Motivation for COTS Middleware

- It is hard to develop distributed applications whose components collaborate *efficiently*, *reliably*, *transparently*, and *scalably*
- To help address this challenge, the Object Management Group (OMG) is specifying the *Common Object Request Broker Architecture* (CORBA)
- OMG is a consortium of $\sim$800 companies
  - Sun, HP, DEC, IBM, IONA, Borland, Cisco, Motorola, Boeing, etc.
- The latest version of the CORBA spec is available online
  - www.omg.org/technology/documents/formal/

Sources of Complexity for Distributed Applications



- **Inherent complexity**
  - Latency
  - Reliability
  - Partitioning
  - Ordering
  - Security
- **Accidental Complexity**
  - Low-level APIs
  - Poor debugging tools
  - Algorithmic decomposition
  - Continuous re-invention

Sources of Inherent Complexity

- *Inherent complexity* results from fundamental challenges in the distributed application domain

- Key challenges of distributed computing include
  - 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 Accidental Complexity

- *Accidental complexity* results from limitations with tools and techniques used to develop distributed applications

- In practice, key limitations of distributed computing include
  - Lack of type-safe, portable, re-entrant, and extensible system call interfaces and component libraries
  - Inadequate debugging support
  - Widespread use of *algorithmic* decomposition
  - Continuous rediscovery and reinvention of core concepts and components

Motivation for CORBA

- Simplifies application interworking
  - Higher level integration than *untyped TCP bytestreams*
- Supports heterogeneity
  - *e.g.*, middleware enables applications to be independent of transports, OS, hardware, language, location, and implementation details
- Benefits for distributed programming similar to OO languages
  - *e.g.*, encapsulation, interface inheritance, polymorphism, and exception handling
- Provides a foundation for higher-level distributed object collaboration
  - *e.g.*, CCM, J2EE, and CORBAServices

CORBA Quoter Example

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

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

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

- Ideally, a distributed service should look just like a non-distributed service
- Unfortunately, life is harder when errors occur...

CORBA Quoter Interface

```
// 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);
};
```

- We write an OMG IDL `interface` for our `Quoter`
  - Used by both clients and servers

Using OMG IDL promotes *language/platform independence*, *location transparency*, *modularity*, and *robustness*

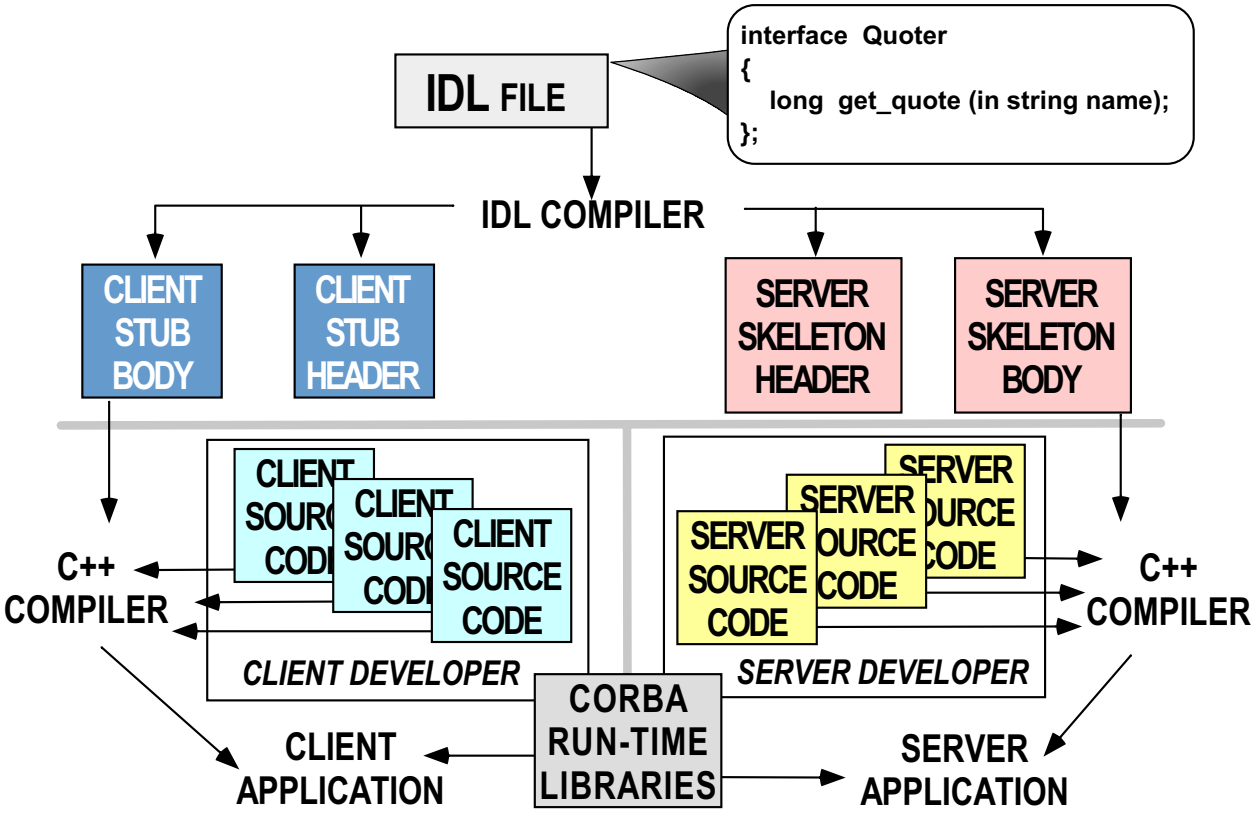
Overview of OMG Interfaces

- OMG `interfaces` are similar to C++ abstract classes or Java `interfaces`
  - They define object types
  - Can be passed as (reference) parameters
  - Can raise exceptions and
  - Can be forward declared
- There are several differences, however, since they
  - Cannot define data members
  - Cannot have private or protected access control sections
  - Must designate their parameter directions
- Only CORBA objects defined with interfaces can be accessed remotely
  - However, *locality constrained* CORBA objects can't be accessed remotely

Overview of OMG Operations

- Each operation in an OMG `interface` must have
  - A name
  - A return type (can be `void`)
  - Zero or more parameters
- An operation can optionally have
  - A `raises` clause, which indicates the exceptions(s) the operation can throw
  - A `oneway` qualifier, which indicates the caller doesn't expect any results
  - A `context` clause, which is deprecated and non-portable...
- Due to limitations with certain programming language mappings, operations cannot be overloaded in IDL interfaces

Using an OMG IDL Compiler for C++



- Different IDL compilers generate different files and provide different options

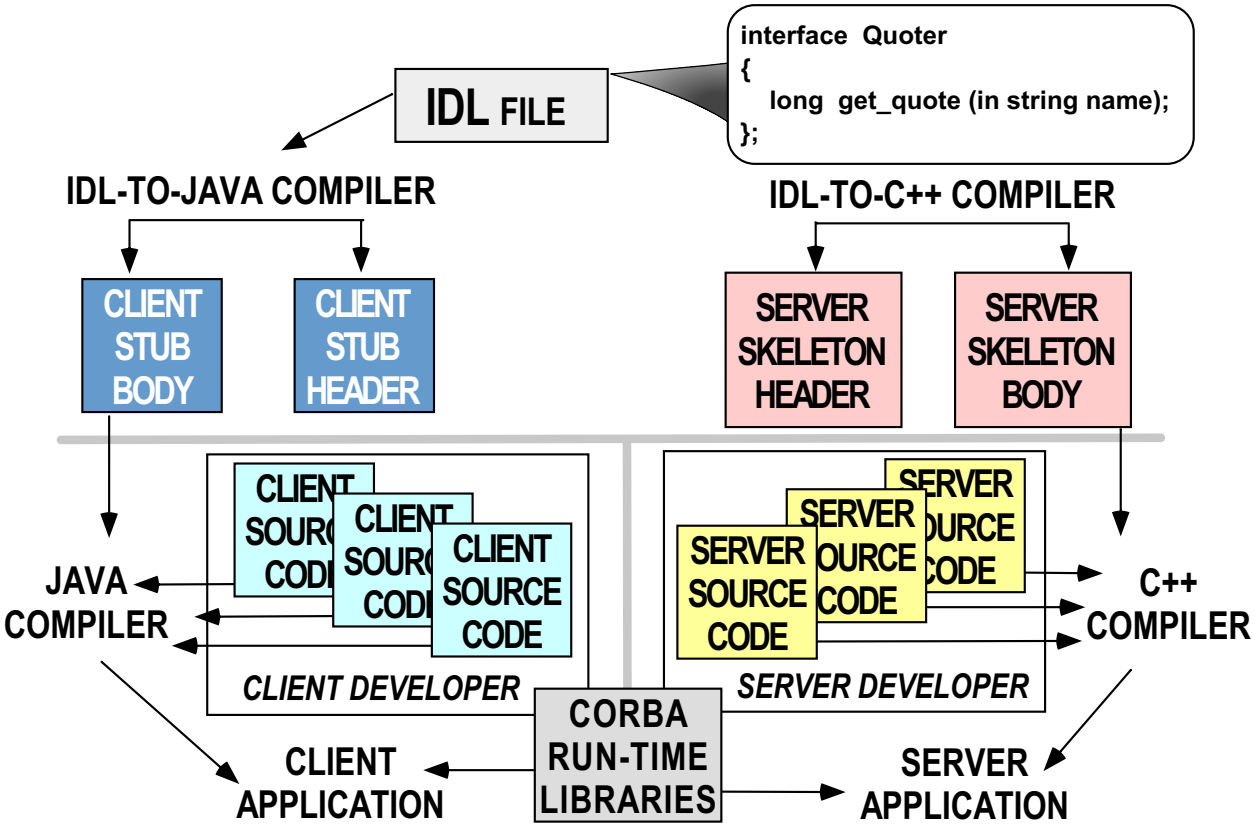
OMG IDL-to-C++ Mapping Rules (1/2)

- There are mappings from OMG IDL to various programming languages standardized by CORBA
- Mapping OMG IDL to C++ can be classified into
  - Basic C++ mapping for basic and structured types
  - Client-side mapping of IDL interfaces into C++ to support client applications
  - Server-side C++ mapping of IDL interfaces into C++ to support server developers
  - Pseudo-object C++ mapping of certain CORBA types, *e.g.*, `Object`, `ORB`, and `PortableServer::POA`
- Memory management in C++ mapping can be tricky

OMG IDL-to-C++ Mapping Rules (2/2)

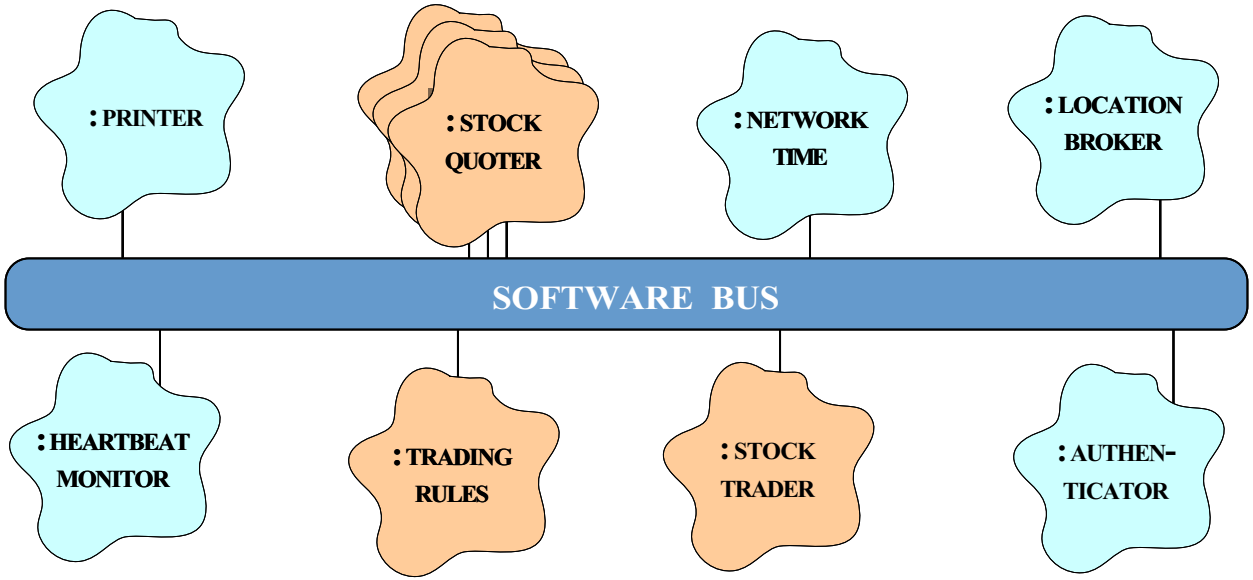
- Sample mapping OMG IDL to C++
  - Each `module` is mapped to a namespace (or class)
  - Each `interface` is mapped to a class
  - Each operation is mapped to a C++ method with appropriate parameters
  - Each read/write attribute is mapped to a pair of get/set methods
  - An `Environment` is defined to carry exceptions in languages that lack this feature
- We'll discuss the various mapping issues as we go along
- See Henning and Vinoski for all details of IDL-to-C++ mapping

Using an IDL Compiler for C++ & Java



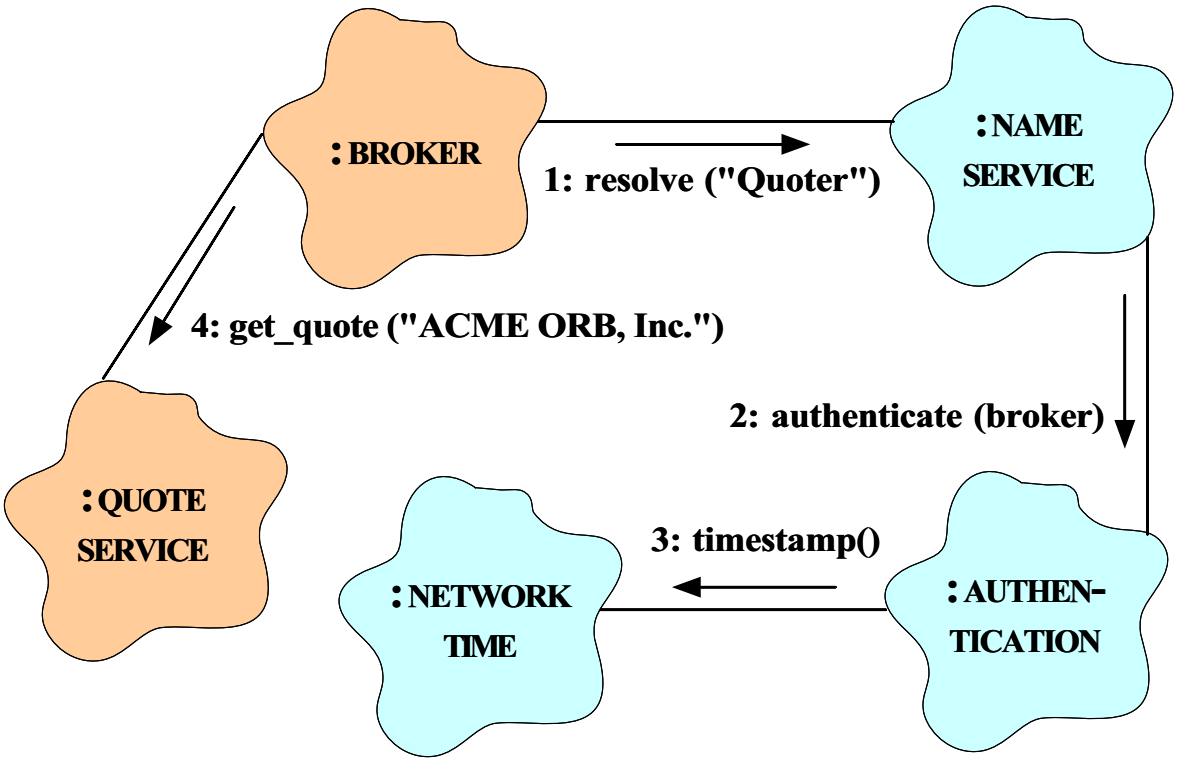
- CORBA makes it straightforward to exchange data between different programming languages *in different address spaces*

Software Bus



- CORBA provides a communication infrastructure for a heterogeneous, distributed collection of collaborating objects
- Analogous to “hardware bus”

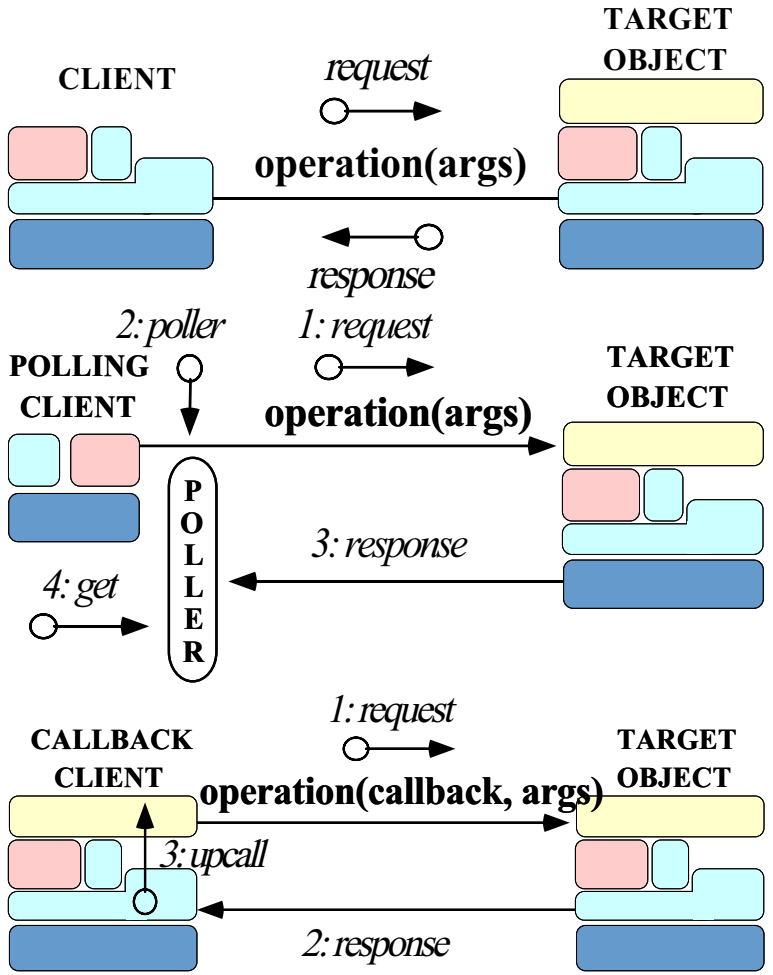
CORBA Object Collaboration



- Collaborating objects can be either remote or local
  - *i.e.*, distributed or collocated
- For this to work transparently the ORB should support *nested upcalls* and *collocation optimizations*

Communication Features of CORBA

- CORBA supports reliable, uni-cast communication
  - *i.e.*, *oneway*, *twoway*, *deferred synchronous*, and *asynchronous*
- CORBA objects can also collaborate in a *client/server*, *peer-to-peer*, or *publish/subscribe* manner
  - *e.g.*, COS Event & Notification Services define a publish & subscribe communication paradigm



Fundamental CORBA Design Principles

- Separation of interface and implementation
  - Clients depend on interfaces, not implementations
- Location transparency
  - Service use is orthogonal to service location
- Access transparency
  - Invoke operations on objects
- Typed interfaces
  - Object references are typed by interfaces
- Support of multiple inheritance of interfaces
  - Inheritance extends, evolves, and specializes behavior

Related Work (1/4)

- Traditional Client/Server RPC (*e.g.*, DCE)
  - Servers offer a service and wait for clients to invoke remote procedure calls (RPCs)
  - When a client invokes an RPC the server performs the requested procedure and returns a result
- Problems with Client/Server RPC
  - Only supports “procedural” integration of application services
  - Doesn’t provide object abstractions, *e.g.*, polymorphism, inheritance of interfaces, etc.
  - Doesn’t support async message passing, or dynamic invocation

Related Work (2/4)

- Windows COM/DCOM/COM+
  - A component model for Windows that support binary-level integration and reuse of components
- Problems with Windows COM/DCOM/COM+
  - Largely limited to desktop applications
  - Does not address heterogeneous distributed computing

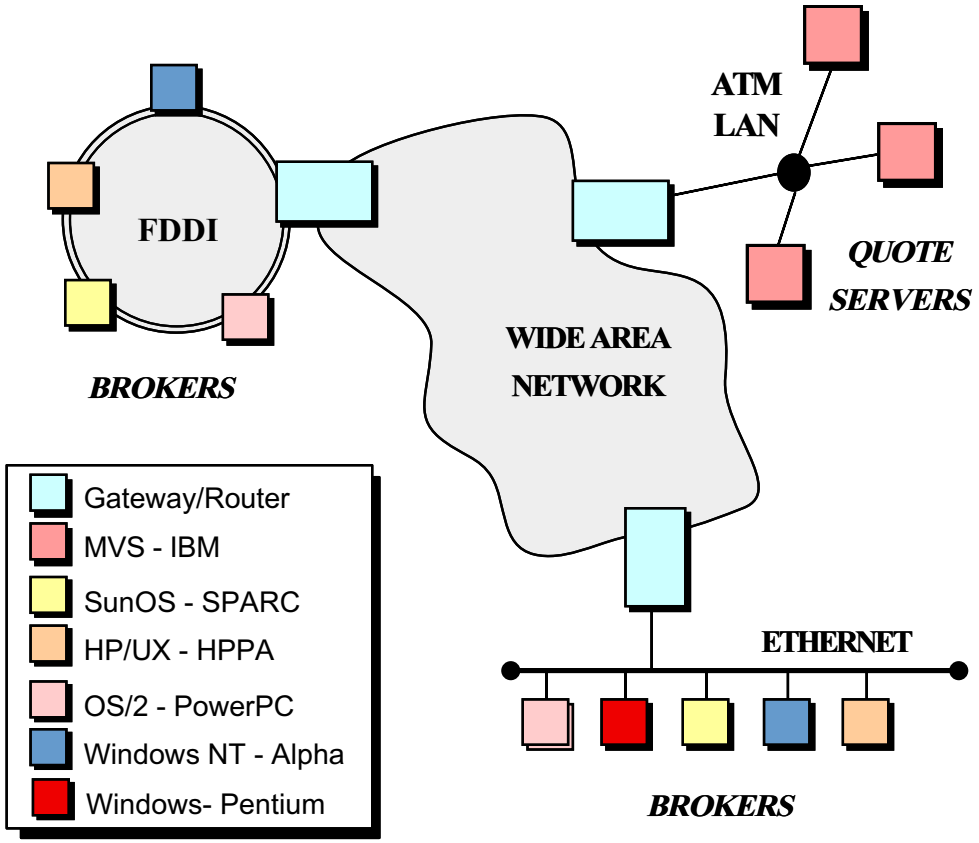
Related Work (3/4)

- SOAP
  - A simple XML-based protocol that allows applications to exchange structured and typed information on the Web using HTTP and MIME
  - Widely implemented
- Problems with SOAP
  - Considerable time/space overhead

Related Work (4/4)

- Java RMI
  - Limited to Java only
    - Can be extended into other languages, such as C or C++, by using a bridge across the Java Native Interface (JNI)
  - Well-suited for all-Java applications because of its tight integration with the Java virtual machine
    - *e.g.*, can pass both object data and code by value
  - However, many challenging issues remain unresolved
    - *e.g.*, security, robustness, and versioning
- J2EE and .NET
  - Higher-level distributed component frameworks
  - Widely used in business/enterprise domains

CORBA Stock Quoter Application Example



- The quote server(s) maintains the current stock prices
- Brokers access the quote server(s) via CORBA
- Note all the heterogeneity!
- We use this example to explore many features of CORBA

Simple OMG IDL Quoter Definition

```
module Stock {
  // Exceptions are similar to structs.
  exception Invalid_Stock {};
  exception Invalid_Factory {};

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

  // A factory that creates Quoter objects.
  interface Quoter_Factory
  {
    // Factory Method that returns a new Quoter
    // selected by name e.g., "Dow Jones,"
    // "Reuters,", etc.
    Quoter create_quoter (in string quoter_service)
      raises (Invalid_Factory);
  };
};
```

Note the use of the Factory Method pattern

Overview of IDL Parameter Passing (1/2)

- Operation parameters in OMG IDL must be designated to have one of the following *directions*:
  - `in`, which means that the parameter is passed from the client to the server
  - `out`, which means that the parameter is returned from the server to the client
  - `inout`, which means that the parameter is passed from the client to the server and then returned from the server to the client, overwriting the original value
- Parameter passing modes are used in CORBA to optimize the data exchanged between client and server

Overview of IDL Parameter Passing (2/2)

- The C++ mapping for parameter passing depend on both the type and the direction
  - Built-in `in` params (*e.g.*, `char` and `long`) passed by value
  - User defined `in` params (*e.g.*, `structs`) passed by const reference
  - Strings are passed as pointers (*e.g.*, const char *)
  - `inout` params are passed by reference
  - Fixed-size `out` params are passed by reference
  - Variable-size `out` params are allocated dynamically
  - Fixed-size return values are passed by value
  - Variable-size return values are allocated dynamically
  - Object reference `out` params and return values are duplicated
- As usual, applications can be shielded from most of these details by using `_var` types

Overview of Object References (1/3)

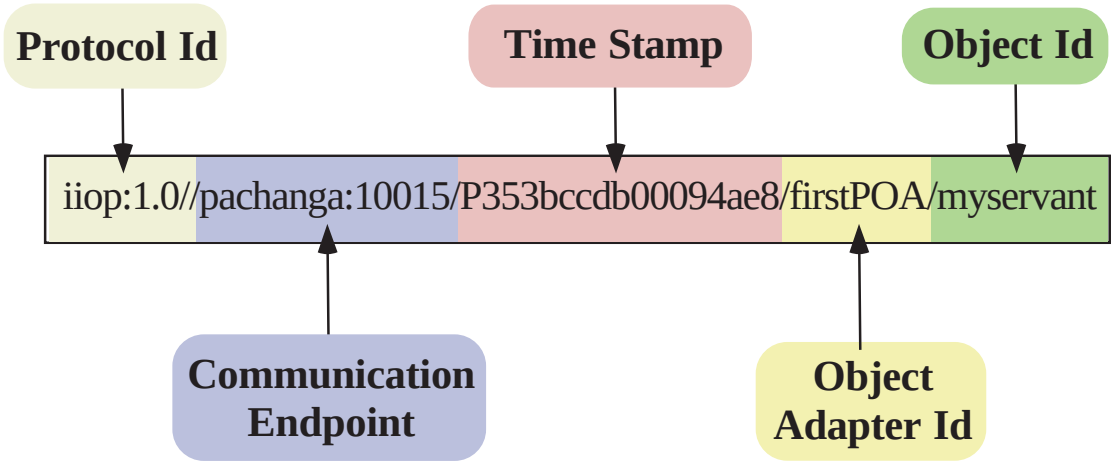
- An object reference is a strongly-typed opaque handle to one instance of an interface that identifies the object's location
- An object reference is an ORB-specific entity that can contain
  - A *repository ID*, which identifies its interface type
  - *Transport address information*, *e.g.*, a server's TCP/IP host/port address(es)
  - An *object key* that identifies which object in the server the request is destined for
- An object reference similar to a C++ "pointer on steriods" that's been enhanced to identify objects in remote address spaces
  - *e.g.*, it can be NULL and it can reference non-existent objects

Overview of Object References (2/3)

- Object references can 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 object stays where it is, however, and its reference is passed by value
- The ORB provides the receiver with a pointer to a proxy in its own address space
  - This proxy refers to the remote object implementation
- Object references are a powerful feature of CORBA
  - *e.g.*, they support *peer-to-peer* interactions and *distributed callbacks*

Overview of Object References (3/3)

- The following is a *transient* object reference
  - The timestamp helps ensure uniqueness across process lifetimes



- *Persistent* object references omit the timestamp to help ensure consistency across process lifetimes
  - There's also a requirement to keep port numbers and IP addresses consistent...

Overview of OMG Modules

- OMG `modules` are similar to C++ namespaces or Java packages
  - *i.e.*, they define scopes and can be nested
- OMG `modules` can be reopened to enable incremental definitions, *e.g.*:

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

- Reopening of modules is particular useful for mutually dependent interfaces that require *forward definitions*

Overview of OMG Exceptions

- Two types of exceptions in OMG IDL inherit from `CORBA::Exception`:
  - **System exceptions** (*e.g.*, `CORBA::OBJECT_NOT_EXIST`), which are predefined by the CORBA spec and must not appear in a `raises` clause
  - **User exceptions** (*e.g.*, `Stock::Invalid_Stock`), which can be defined by user applications and can appear in a `raises` clause
- There are various restrictions on exceptions in CORBA
  - *e.g.*, they can't be nested or inherited and can't be members of other data types

Revised OMG IDL Quoter Definition

Apply the CORBA Lifecycle Service

```
module Stock {
  exception Invalid_Stock {};

  interface Quoter : CosLifeCycle::LifeCycleObject
  {
    long get_quote (in string stock_name)
      raises (Invalid_Stock);
    // Inherits:
    // void remove () raises (NotRemovable);
  };

  // Manage the lifecycle of a Quoter object.
  interface Quoter_Factory :
    CosLifeCycle::GenericFactory
  {
    // Returns a new Quoter selected by name
    // e.g., "Dow Jones," "Reuters,", etc.
    // Inherits:
    // Object create_object (in Key k,
    //                       in Criteria criteria)
    // raises (NoFactory, InvalidCriteria,
    //         CannotMeetCriteria);
  };
};
```

Overview of OMG Object

- The `CosLifeCycle::GenericFactory::create_object()` factory method returns an object reference to an instance that's derived from the `CORBA::Object` interface
- Since all objects implicitly inherit from `CORBA::Object`, all object references support the following operations:

```
interface Object {
  // Reference counting methods
  Object duplicate ();
  void release ();
  // Checks for existence and reference identity & relationships
  boolean is_nil ();
  boolean non_existent ();
  boolean is_equivalent (in Object another_object);
  boolean is_a (in string repository_id);
  // ...
```

Overview of Fixed- and Variable-size Types

Certain types are variable-size:

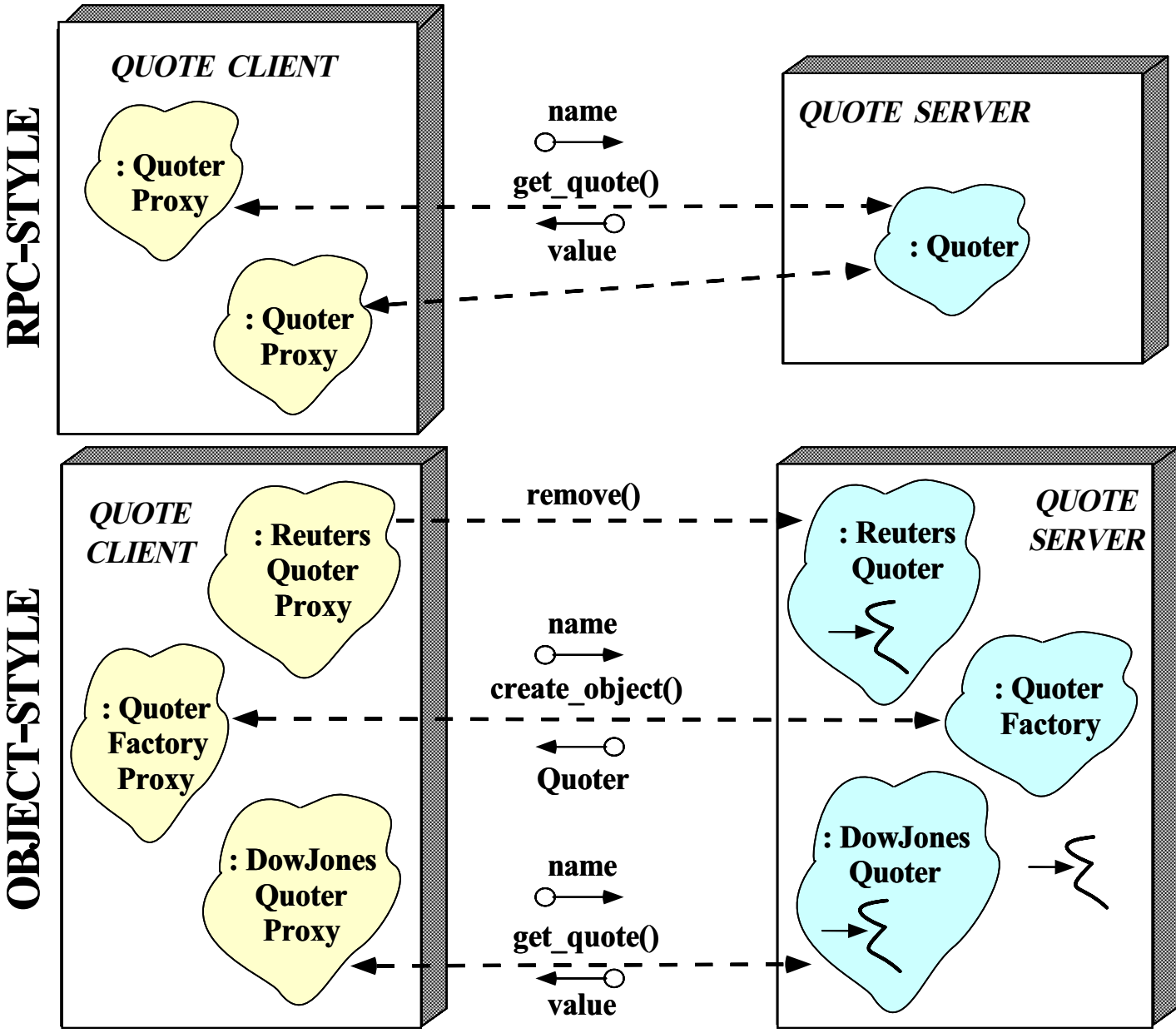
- Bounded or unbounded strings (as shown in the `Stock::Quoter` example)
- Bounded or unbounded sequences
- Object references
- Type `any`

Other types can be variable- or fixed-size:

- `structs`, `unions`, and arrays are fixed-size if they contain only fixed-size fields (recursively)
- `structs`, `unions`, and arrays are variable-size if they contain *any* variable-size fields (recursively)

- Variable-size types require the sender to dynamically allocate instances and the receiver to deallocate instances
- Again, use `_var` types to simplify programming

RPC- vs. Object-style Designs



Results of Compiling the Stock.idl File

Running the `Stock` module through the IDL compiler generates *stubs* and *skeletons*

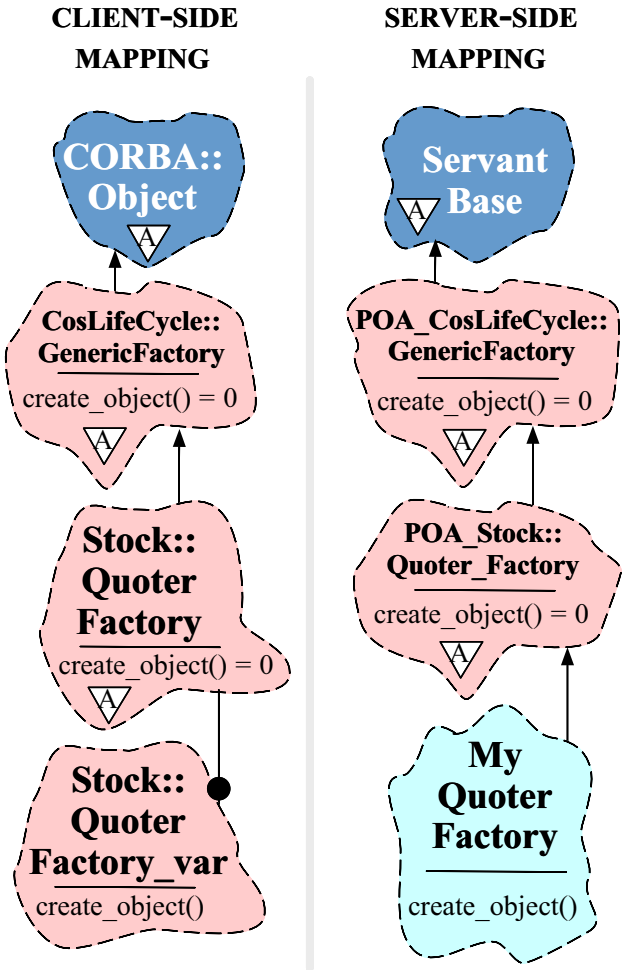
- Each (twoway) stub is a *proxy* that
  1. Ensures a connection to the server is established
  2. Marshals the request parameters
  3. Sends the request
  4. Waits to get the reply
  5. Demarshals the reply parameters
  6. Returns to the client caller
- Each skeleton is an *adapter* that
  1. Demarshals the request parameters
  2. Performs an upcall on the designated servant method
  3. Marshals the reply parameters
  4. Sends the reply back to the client

Overview of Generated Client Stubs

```
// Note C++ mapping of IDL module type
namespace Stock {
  // Note C++ mapping of IDL interface type
  class Quoter // Quoter also IS-A CORBA::Object.
    : public virtual CosLifeCycle::LifeCycleObject {
  public:
    // Note C++ mapping of IDL long and string types
    CORBA::Long get_quote (const char *stock_name);
    // ...
  };

  class Quoter_Factory
    : public virtual CosLifeCycle::GenericFactory {
  public:
    // Factory method for creation that inherits:
    // CORBA::Object_ptr create_object
    //   (const CosLifeCycle::Key &factory_key,
    //    const CosLifeCycle::Criteria &criteria)
    // Note C++ mapping of Key and Criteria structs.
    // ...
  };
};
```

Note that you never instantiate a stub class directly, but always via a *factory*

OMG Object and POA IDL Mappings

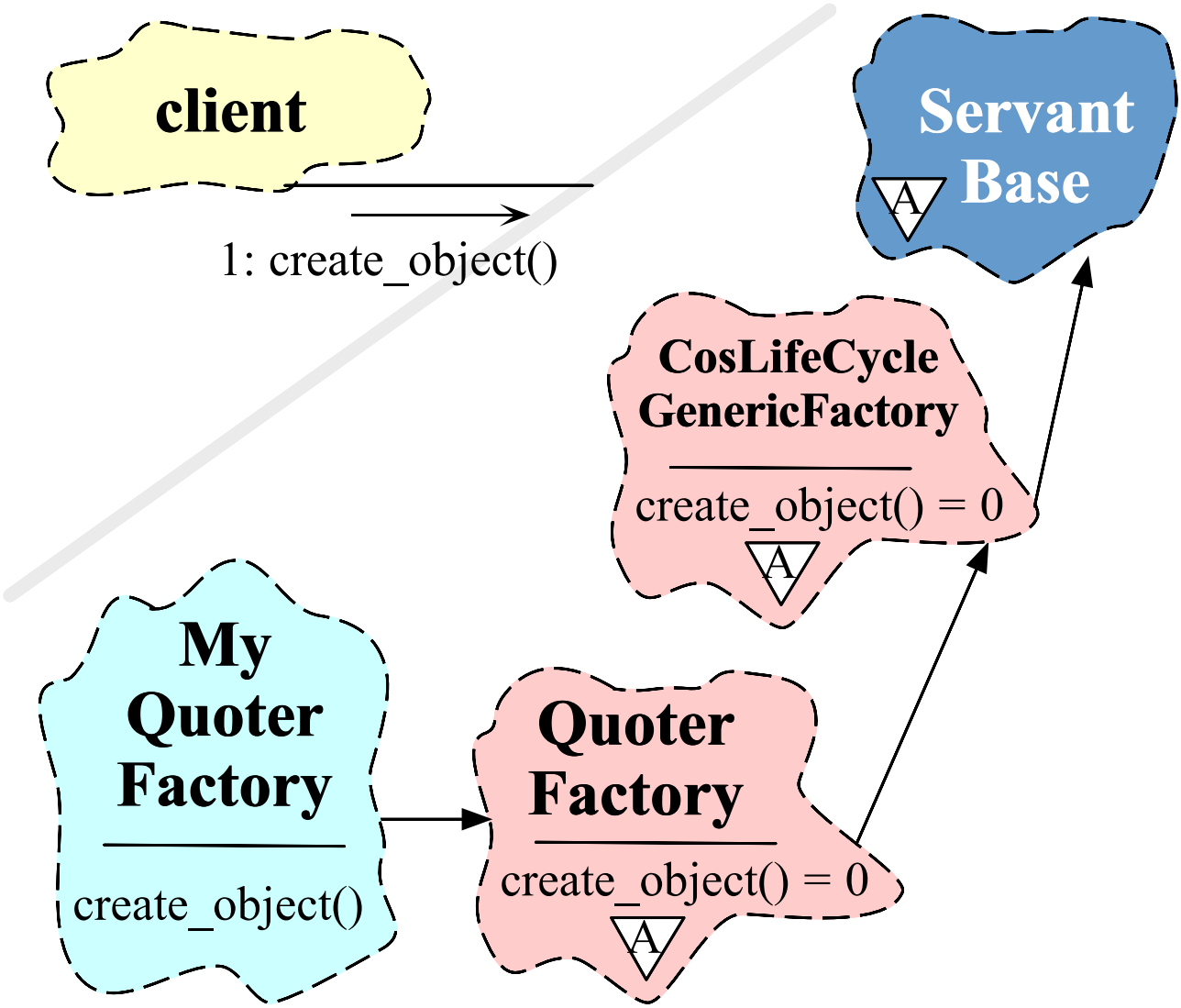


- The OMG client mapping inherits all proxy interfaces from the `Object` interface
  - Moreover, proxy classes mirror the IDL inheritance hierarchy, so references to derived interfaces are compatible with references to base interfaces via *widening* and *polymorphism*
- The IDL server C++ mapping inherits all Servants from `ServantBase`

Overview of Generated Server Skeletons

- Skeleton classes provide the server counterpart to the client stub class proxies
  - There's a C++ virtual method in the skeleton class for each operation defined in the IDL interface
- CORBA associates a user-defined servant class to a generated IDL skeleton class using either
  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`

The Class Form of the Adapter Pattern



Defining a Servant Using Inheritance

- Servant classes can inherit from their skeleton directly:

```
class My_Quoter_Factory : public virtual POA_Stock::Quoter_Factory
public:
  My_Quoter_Factory (const char *factory = "my quoter factory");
  virtual CORBA::Object_ptr // Factory method for creation.
    create_object (const CosLifeCycle::Key &factory_key,
                   const CosLifeCycle::Criteria &the_criteria)
        throw (CORBA::SystemException, QuoterFactory::NoFactory);
private:
  std::map<std::string, CORBA::Object_var> objref_list_;
};
```

- However, this approach can create a “brittle” hierarchy and make it hard to integrate with legacy code (*e.g.*, when distributing a stand-alone application)
- Moreover, virtual inheritance is sometimes implemented poorly by C++ compilers

PortableServer::ServantBase

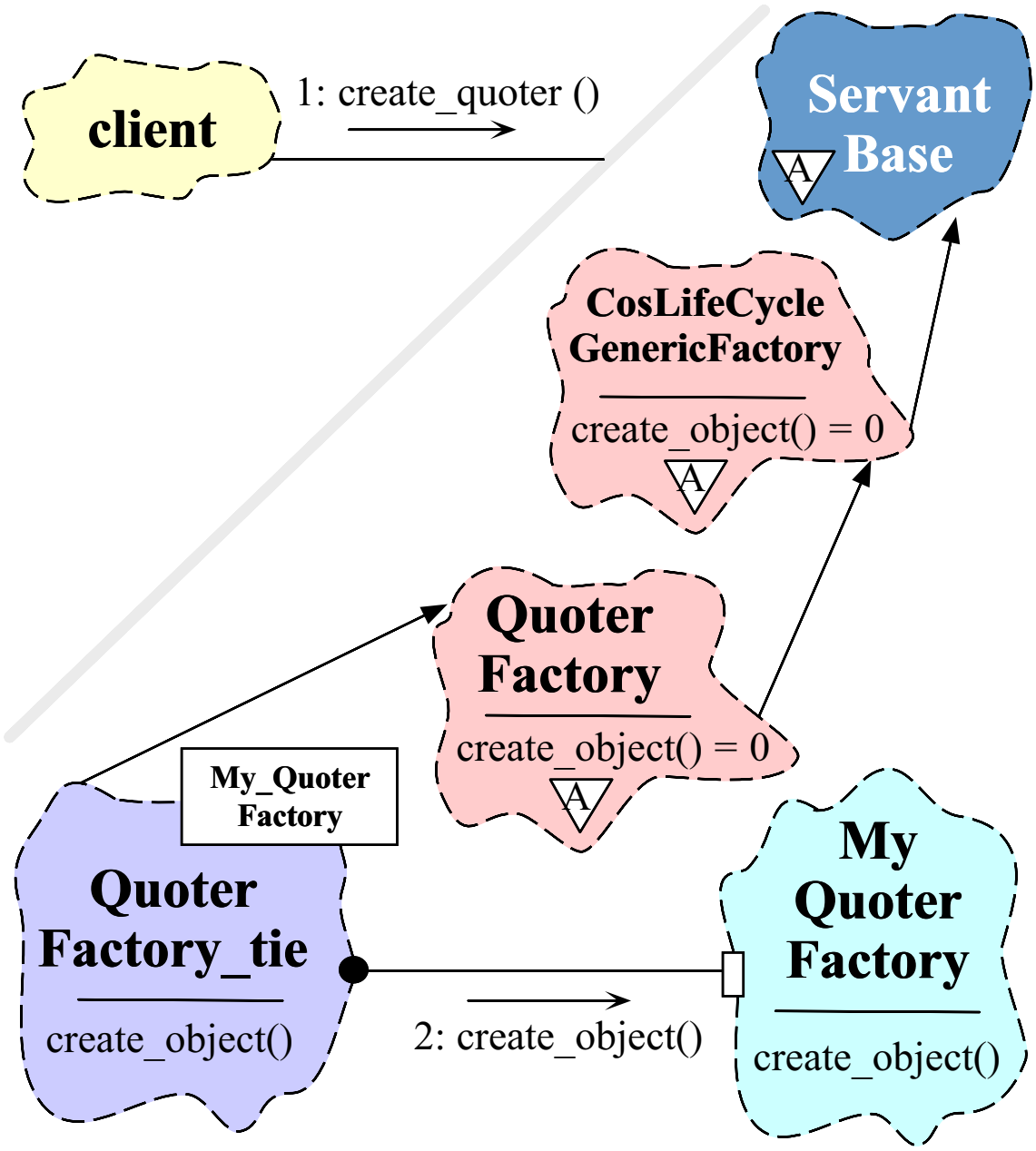
- `PortableServer::ServantBase` implements reference counting for servant classes via two methods:
  - `_add_ref()` increments reference count (initial count is 1)
  - `_remove_ref()` decrements reference count by 1 and deletes servant when value is 0

- Servant classes implicitly inherit from `PortableServer::ServantBase`

- Developers should create the servant using operator `new`

- When ever any method is called that returns a servant object (*e.g.*, `id_to_servant()`, `reference_to_servant()`, `_this()`), `_add_ref()` is called automatically to increment the ref count by 1

The Object Form of the Adapter Pattern



A TIE-based Implementation

```
class My_Quoter_Factory {
public:
  My_Quoter_Factory (const char *name = "my quoter factory");
  // Factory method for creation.
  CORBA::Object_ptr create_object
    (const CosLifeCycle::Key &factory_key,
     const CosLifeCycle::Criteria &the_criteria)
    throw (CORBA::SystemException, QuoterFactory::NoFactory);
private: // ...
};
```

TIE allows classes to become distributed even if they weren't developed with prior knowledge of CORBA

- There is no use of inheritance and operations need not be virtual!
- However, lifecycle issues for "tie" and "tied" objects are tricky...

Defining a Servant Using TIE

```
namespace POA_Stock {
  template <class Impl>
  class Quoter_Factory_tie : public Quoter_Factory { /* ... */ };
  // ...
```

We generate a typedef and a servant that places an implementation pointer object within the TIE class:

```
typedef POA_Stock::Quoter_Factory_tie<My_Quoter_Factory>
        MY_QUOTER_FACTORY;

MY_QUOTER_FACTORY *factory =
  new MY_QUOTER_FACTORY (new My_Quoter_Factory);
```

All operation calls via the TIE class are then delegated to the implementation object

Implementing My_Quoter_Factory

The following code is identical regardless of which form of Adapter pattern is used for servant classes

```
CORBA::Object_ptr
My_Quoter_Factory::create_object
  (const CosLifeCycle::Key &factory_key,
   const CosLifeCycle::Criteria &the_criteria)
{
  POA_Stock::Quoter *quoter;
  PortableServer::ServantBase_var xfer;

  // Factory method selects quoter.
  if (strcmp (factory_key.id,
                    "my quoter") == 0) {
    xfer = quoter = new My_Quoter;
  }
  else if (strcmp (factory_key.id,
                    "dow jones") == 0) {
    xfer = quoter = new Dow_Jones_Quoter;
  } else // Raise an exception.
    throw Quoter_Factory::NoFactory ();

  // Create a Stock::Quoter_ptr, register
  // the servant with the default_POA, and
  // return the new Object Reference.
  return quoter->_this ();
};
```

Another `create_object()` Implementation

Preventing multiple servant activations for the same key

```
CORBA::Object_ptr
My_Quoter_Factory::create_object
  (/* args omitted */)
{
  CORBA::Object_var objref;
  if (objref_list_.find (factory_key.id,
                         objref) == 0)
    return objref._retn ();
  // Declarations...
  // Factory method selects quoter.
  if (strcmp (factory_key.id,
                    "my quoter") == 0) {
    xfer = quoter = new My_Quoter;
  }
  // Create a Stock::Quoter_ptr, register
  // the servant with the default_POA, and
  // return the new Object Reference.
  objref = quoter->_this ();

  // operator=() defined in CORBA::Object_var
  // duplicates references.
  objref_list_.bind (factory_key.id,
                     objref);
  return objref._retn ();
};
```

Overview of Implicit Activation and _this()

- Each generated skeleton class contains a `_this()` method, *e.g.*:

```
class POA_Stock::Quoter
  : public virtual CosLifeCycle::LifeCycleObject {
public:
  Quoter_ptr _this ();
};
```

- Depending on the *POA policy*, the `_this()` method can be used to activate a servant and return the corresponding object reference
- Internally, `_this()` duplicates the object reference, so it must be decremented at some point to avoid memory leaks
- If you use `_this()` for a servant in a non-Root POA, make sure to override the servant's `_default_POA()` method...

Reference Counting Servants Properly

- When a servant that inherits from `PortableServer::ServantBase` is created its ref count is set to 1
- When it's activited with the POA its ref count is incremented to 2
- When `PortableServer::POA::deactivate_object()` is called later the ref count is deremented by 1
  - But the servant is only destroyed when its ref count is 0
- To ensure the servant is destroyed properly, use `PortableServer::ServantBase_var` to hold the newly allocated servant pointer since its destructor calls `_remove_ref()` automatically
  - This approach is also exception-safe

Implementing the `My_Quoter` Interface

Implementation of the `Quoter` IDL interface

```
class My_Quoter : virtual public POA_Stock::Quoter
{
public:
  My_Quoter (void *state); // Constructor.

  // Returns the current stock value.
  long get_quote (const char *stock_name)
    throw (CORBA::SystemException, Quoter::InvalidStock);

  // Deactivate quoter instance.
  void remove (void)
    throw (CORBA::SystemException,
           CosLifeCycle::LifeCycleObject::NotRemovable);
private:
  // ...
};
```

Overview of Throwing Exceptions

- To throw an exception, simply instantiate the exception class and throw it, *i.e.*, `throw Quoter_Factory::NoFactory()`
  - The process is slightly more baroque using emulated exceptions
- Servant code should generally try to throw user exceptions
  - Avoid throwing system exceptions since they convey less information to clients
  - When you do throw a system exception, set its *completion status* to indicate what state the operation was in
- Use C++ `try` blocks to protect scopes where operations may throw exceptions and always use `_var` and/or `std::auto_ptr<>` types appropriately

Memory Management Tips (1/3)

- Memory management is straightforward for basic/fixed types, but more complicated for variable-sized types
- Rule of thumb: Caller owns all storage
- Use `_var` to manage memory automatically
  - But *never* declare method signatures using `_var`; use `_ptr` instead...

```
Quoter_ptr factory (CORBA::Object_ptr arg); // Ok
Quoter_var factory (CORBA::Object_var arg); // Wrong
```

- Remember `_var` owns the memory
  - Unless `_retn()` is used
- Not obeying the rules can cause crashes (if you're lucky) or memory leaks/corruption (if you're not)

Memory Management Tips (2/3)

- Server-side implementations of operations should
  - Object references must be duplicated before being stored to prevent premature deletion
  - Operations receiving variable-sized data types should perform a deep copy of the incoming data to safely use them later
    - Caching pointers to the parameter will *not* help
  - Allocate memory for *out* and *return* parameters of variable-sized types
    - Clients handle this by using `_var` types
  - Servants automatically give up ownership of memory allocated for *out* and *return* parameters.
    - Call `_duplicate()` or equivalent operation if servants need to retain ownership.
  - Use `PortableServer::ServantBase_var` and

`std::auto_ptr<>` to prevent memory leaks when exceptions occur

Memory Management Tips (3/3)

- Frequently Made Mistakes (FMM's)

  1. Storing strings within sequences and structs
  2. Not handling the return reference from an operation, but passing it to another operation
  3. Not activating the POA manager
  4. Not setting `length()` of sequence properly
  5. Not duplicating object references properly
  6. Not using `ServantBase_var` properly

- We'll show how to avoid these mistakes in subsequent slides

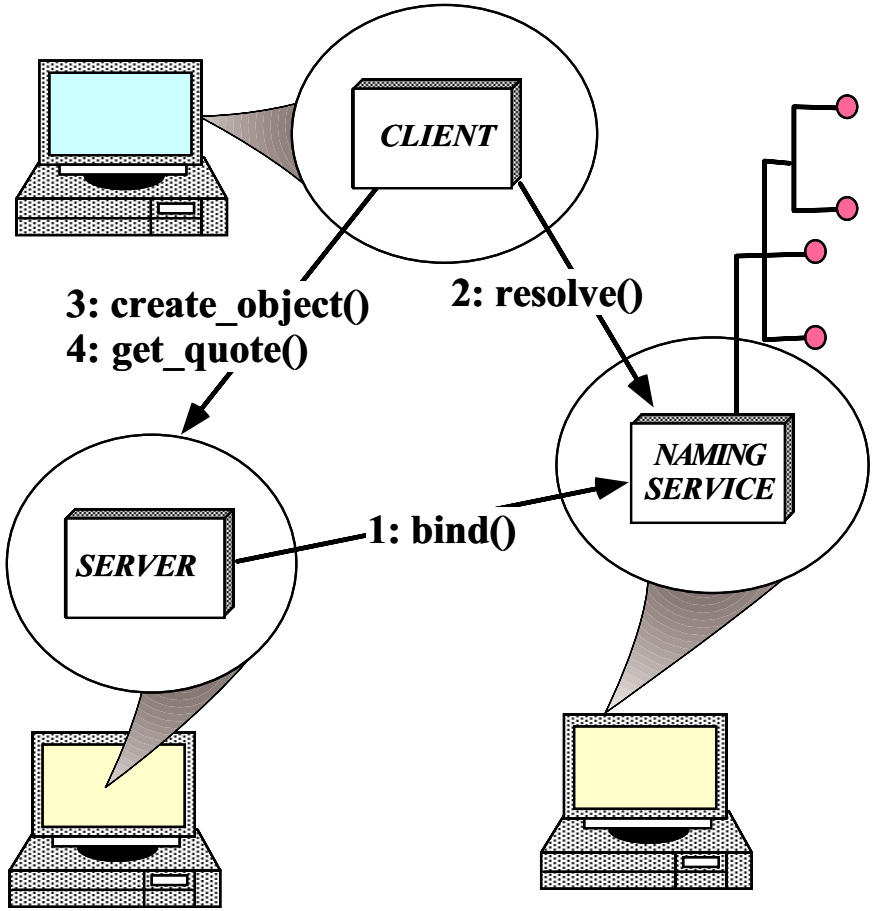
Motivation for the CORBA Naming Service

- Clients access `Quoter` objects returned by `My_Quoter_Factory`
  - But how do clients find `My_Quoter_Factory`?!
- One approach is to use `CORBA::ORB` helper operations to convert an object reference to a string and vice versa

```
interface ORB {
  // ...
  string object_to_string (in Object o);
  Object string_to_object (in string s);
};
```

- Stringified object references can be written to and read from a file, passed between ORBs, and/or stored in a database
- A more effective and scalable approach, however, is often to use the CORBA Naming Service

Overview of the CORBA Naming Service



- **Purpose**
  - Maps sequences of strings to object references
- **Capabilities**
  - A `Naming Context` can be a hierarchically nested graph
  - `Naming Contexts` can also be federated

Registering `My_Quoter_Factory` with the Naming Service

```
extern CosNaming::NamingContextExt_ptr
       name_context;

My_Quoter_Factory::My_Quoter_Factory
  (const char *factory_name) {
  char tmp[] = "object impl";
  CORBA::StringSeq sseq (2); sseq.length (2);
  sseq[0] = factory_name; sseq[1] = tmp; // FMM 1
  // FMM: assignment from const char * duplicates
  // the string but a non-const char doesn't.
  CosNaming::Name name;
  name.length (1);
  name[0].id = sseq[0]; name[0].kind = sseq[1];

  // Obtain objref and register with POA.
  Quoter_Factory_var qf = this->_this ();

  // Export objref to naming context.
  name_context->bind (name, qf.in ());

  // FMM 2 is to do the following.
  // name_context->bind (name, this->_this ());
};
```

Real code should handle exceptions...

Programming with Object References (1/3)

- An IDL compiler generates two different object reference types for each `interface`:
  - `<interface>_ptr` → C++ pointer to object reference
    - An "unmanaged type" that requires programmers to manipulate reference ownership via `<proxy>::_duplicate()` and `CORBA::release()`
  - `<interface>_var` → "Smart pointer" to object reference
    - Manages reference lifetime by assuming ownership of dynamically allocated memory and deallocating it when the `_var` goes out of scope
    - `operator->()` delegates to the underlying pointer value
    - `_var` types are essential for writing exception-safe code

Programming with Object References (2/3)

- You should use `_var` types as often as possible since they automate most of the error-prone reference counting, *e.g.*:

```
// When ORB returns object reference its proxy has
// a reference count of 1
Quoter_ptr quoter = resolve_quoter_service ();
CORBA::Long value = quoter->get_quote ("ACME ORB Inc.");
CORBA::release (quoter);
// release() decrements the reference count by one,
// which causes deallocate when the count reaches 0
```

  versus

```
Quoter_var quoter = resolve_quoter_service ();
CORBA::Long value = quoter->get_quote ("ACME ORB Inc.");
// quoter automatically releases object reference.
```

- Calls to `_duplicate()` and `CORBA::release()` only affect the local proxy, *not* the remote object!!!

Programming with Object References (3/3)

- To handle broken C++ compilers, you may need to use special helper methods generated by the IDL compiler to workaround problems with implicit type conversions from `_var` to the underlying pointer
  - `in()` passes the `_var` as an `in` parameter
  - `inout()` passes the `_var` as an `inout` parameter
  - `out()` passes the `_var` as an `out` parameter
- Variable-size `_var` types have a `_retn()` method that transfers ownership of the returned pointer
  - This method is important for writing exception-safe code

The Main Server Program

Uses *persistent activation* mode

```
int main (int argc, char *argv[])
{
  ORB_Manager orb_manager (argc, argv);

  const char *factory_name = "my quoter factory";

  // Create the servant, which registers with
  // the rootPOA and Naming Service implicitly.
  My_Quoter_Factory *factory =
    new My_Quoter_Factory (factory_name);

  // Transfer ownership to smart pointer.
  PortableServer::ServantBase_var xfer (factory);

  // Block indefinitely waiting for incoming
  // invocations and dispatch upcalls.
  return orb_manager.run ();
  // After run() returns, the ORB has shutdown.
}
```

Motivation for ORB_Manager

- Like many CORBA servers, our stock quoter server is initialized via the following steps:

  1. We call `CORBA::ORB_init()` to obtain the locality constrained object reference to the ORB pseudo-object
  2. We use the ORB object reference to obtain the Root POA
  3. We then instantiate the quoter factory servant and activate it to obtain its object reference
  4. We next make the object reference for the quoter factory available to clients via the Naming Service
  5. Finally, we activate the Root POA's manager and run the ORB's event loop

- To automate many of these steps, we define the `ORB_Manager` wrapper facade class

Overview of ORB_Manager

```
class ORB_Manager {
public:
  // Initialize the ORB manager.
  ORB_Manager (int argc, char *argv[]) {
    orb_ = CORBA::ORB_init (argc, argv, 0);
    CORBA::Object_var obj =
      orb_->resolve_initial_references ("RootPOA");
    poa_ =
      PortableServer::POA::_narrow (obj.in ());
    poa_manager_ = poa_->the_POAManager ();
  }
  // ORB Accessor.
  CORBA::ORB_ptr orb (void) { return orb_; }

  // Run the main ORB event loop.
  int run (void) {
    poa_manager_->activate (); // FMM 3
    return orb_->run ();
  }

  // Cleanup the ORB and POA.
  ~ORB_Manager () { orb_->destroy (); }
private:
  CORBA::ORB_var orb_;
  PortableServer::POA_var poa_;
  PortableServer::POA_Manager_var poa_manager_;
};
```

Overview of Pseudo-objects and Locality Constraints

- The `CORBA::ORB` and `PortableServer::POA` interfaces define "pseudo-objects," *i.e.*:

```
orb_ = CORBA::ORB_init (argc, argv, 0);
CORBA::Object_var obj =
  orb_->resolve_initial_references ("RootPOA");
poa_ =
  PortableServer::POA::_narrow (obj.in ());
```

- Pseudo-objects have IDL interfaces but are implemented in the ORB's runtime library, rather than by using generated stubs/skeletons
- Pseudo-objects are "locality constrained," which means that their object references can't be passed to remote address spaces

Overview of `_narrow()` Conversion Operators

- The IDL compiler generates static method `_narrow()` for each proxy that behaves like the C++ `dynamic_cast` operator
  - *i.e.*, it returns a non-nil reference if the argument to the method is the right type, else nil
- Note that `_narrow()` implicitly calls `_duplicate()`, which increments the reference count

```
class Quoter : public virtual CosLifeCycle::LifeCycleObject {
public:
  static Quoter_ptr _narrow (CORBA::Object_ptr arg);
  // ...

class Stat_Quoter : public virtual Quoter {
public:
  static Stat_Quoter_ptr _narrow (CORBA::Object_ptr arg);
  // ...
```

Overview of ORB Shutdown

- The following two operations shutdown the ORB gracefully:

```
interface ORB {
  void shutdown (in boolean wait_for_completion);
  void destroy ();
};
```

- These operations do the following:
  - Stop the ORB from accepting new requests
  - Allow existing requests to complete and
  - Destroy all POAs associated with the ORB
- The `wait_for_completion` boolean allows the caller to decide whether to wait for the ORB to finish shutting down before returning
  - This is important for multi-threaded ORB implementations...

Recap of the Stock Quoter Server

- In our stock quoter server, we (*i.e.*, the application developers) simply write
  1. The IDL interfaces
  2. The servant classes
  3. Code to initialize the server event loop
- The ORB and associated tools (*e.g.*, IDL compiler) provides the rest:
  1. Generated skeleton classes that connect the ORB with the application-defined servant classes
  2. (De)marshaling code
  3. Management of object references
  4. The ORB runtime libraries that handle connection management, GIOP data transfer, endpoint and request demuxing, and concurrency control

How a Client Accesses a CORBA Object

- Several steps:
  1. Client uses `resolve_initial_references()` and "Interoperable Naming Service" to obtain a `NamingContext`
     - This is the standard ORB "bootstrapping" mechanism
  2. Client then uses `NamingContext` to obtain desired object reference
  3. The client then invokes operations via object reference

- Object references can be passed as parameters to other remote objects
  - This design supports various types of "factory" patterns

Stock Quoter Client Program (1/3)

```
int main (int argc, char *argv[]) {
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, 0);

  CORBA::Object_var obj =
    orb->resolve_initial_references ("NameService");

  CosNaming::NamingContextExt_var name_context =
      CosNaming::NamingContextExt::_narrow (obj.in ());

  Stock::Quoter_var quoter; // Manages refcounts.
  char *stock_name = 0;
```

Stock Quoter Client Program (2/3)

```
try { // Use a factory to resolve any quoter.
  Stock::Quoter_Factory_var qf =
    resolve_service<Stock::Quoter_Factory>
      ("my quoter factory", name_context.in ());
  if (CORBA::is_nil (qf.in ())) return 0;

  CosLifeCycle::Key key; key.length (1);
  key[0].id = "my quoter";

  // Find a quoter and invoke the call.
  CORBA::Object_var obj = qf->create_object (key);
  quoter = Stock::Quoter::_narrow (obj);

  stock_name = CORBA::string_dup ("ACME ORB Inc.");
  CORBA::Long value = quoter->get_quote (stock_name);
```

Stock Quoter Client Program (3/3)

```
    cout << stock_name << " = " << value << endl;
    // Destructors of *_var release memory.
  }
  catch (Stock::Invalid_Stock &)
  {
    cerr << stock_name << " not valid" << endl;
  } catch (...) { /* Handle exception... */ }

  CORBA::string_free (const_cast <char *> (stock_name));

  quoter->remove (); // Shut down server object
}
```

Overview of Memory Management for OMG Strings

- CORBA provides the following methods that must be used to manage the memory of dynamically allocated strings

```
namespace CORBA {
  char *string_dup (const char *ostr);
  void string_free (char *nstr);
  char *string_alloc (ULong len); // Allocates len + 1 chars
  // ... Similar methods for wstrings ...
}
```

- These methods are necessary for platforms such as Windows that have constraints on heap allocation/deallocation
- In the Stock Quoter client example above we could have avoided the use of dynamic string allocations by simply using the following

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

Obtaining an Object Reference

```
template <class T>
typename T::_ptr_type /* trait */
resolve_service (const char *n,
          CosNaming::NamingContextExt_ptr name_context) {
  CosNaming::Name svc_name;
  svc_name.length (1); svc_name[0].id = n;
  svc_name[0].kind = "object impl";

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

  // Can also use
  // obj = name_context->resolve_str (n);

  // Narrow to the T interface and away we go!
```

```
  return T::_narrow (obj);
}
```

Coping with Changing Requirements

- New Quoter features
  - Format changes to extend functionality
  - New interfaces and operations
- Improving existing Quoter features
  - Batch requests
- Leveraging other ORB features
  - Asynchronous Method Invocations (AMI)
  - Passing object references to implement a publisher/subscriber architecture

New Formats

For example, 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 {
  // No change to Quoter interface!!
  interface Quoter { /* ... */ };

  interface Stat_Quoter : Quoter // a Stat_Quoter IS-A Quoter {
    // Note OMG IDL's inability to support overloading!
    long get_stats (in string stock_name,
                    out double percent_change,
                    out long volume) raises (Invalid_Stock);
  // ...
```

- Applications can pass a `Stat_Quoter` whereever a `Quoter` is expected
  - Clients can use `_narrow()` to determine actual type

New Interfaces and Operations

For example, 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);
    // sell() operation is similar...
  };
  interface Trader_Factory { /* ... */ };
};
```

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

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

Note that you can't inherit the same operation from more than one interface

Batch Requests

Improve performance for multiple queries or trades

```
interface Batch_Quoter : Stat_Quoter
{ // Batch_Quoter IS-A Stat_Quoter
  typedef sequence<string> Names;
  struct Stock_Info {
    // Acts like String_var initialized to empty string.
    string name;
    long value;
    double change;
    long volume;
  };
  typedef sequence<Stock_Info> Info;
  exception No_Such_Stock { Names stock; };

  // Note problems with exception design...
  void batch_quote (in Names stock_names,
                    out Info stock_info) raises (No_Such_Stock);
};
```

Overview of OMG Structs

- IDL `structs` are similar to C++ `structs`
  - *i.e.*, they contain one or more fields of arbitrary types
- However, IDL `structs` must be named and have one or more fields
- The C++ mapping rules are different for fixed- and variable-size `structs`
  - *i.e.*, variable-size `structs` must be dynamically allocated by sender and deallocated by receiver
- Using the IDL-generated `_var` helper types minimize the differences between fixed- and variable-sized `structs` in C++ mapping

Overview of OMG Sequences (1/2)

- IDL sequences are variable-length vectors of size >= 0
- They can be *bounded* or *unbounded*
  - Bounded sequences have a max number of elements
  - Unbounded sequences can grow to any (reasonable) length
- Sequences can contain other sequences

```
typedef sequence<octet> Octet_Sequence;
typedef sequence<Octet_Sequence> Octet_Argv;
```

- Sequences can be also used to define recursive data structures for `structs` and `unions`

```
struct Node {
  sequence<Node> Children;
  // ...
};
```

Overview of OMG Sequences (2/2)

- Each IDL sequence type maps to a distinct C++ class
- The `length()` accessor returns # elements in sequence
- The `length()` mutator can change # elements in sequence
- Each C++ class defines pair of overloaded subscript operators (`operator[]`)
- Although it's illegal to access beyond the current length, you can use the `length()` mutator to grow the sequence length at its tail
- FMM 4: Using the sequence to store data without setting the `length()` can cause undefined behaviors
- The copying semantics of sequences depend on the types of its elements

Motivation for Asynchronous Method Invocations (AMI)

- Early versions of CORBA lacked support for asynchronous two-way invocations

- This omission yielded the following drawbacks

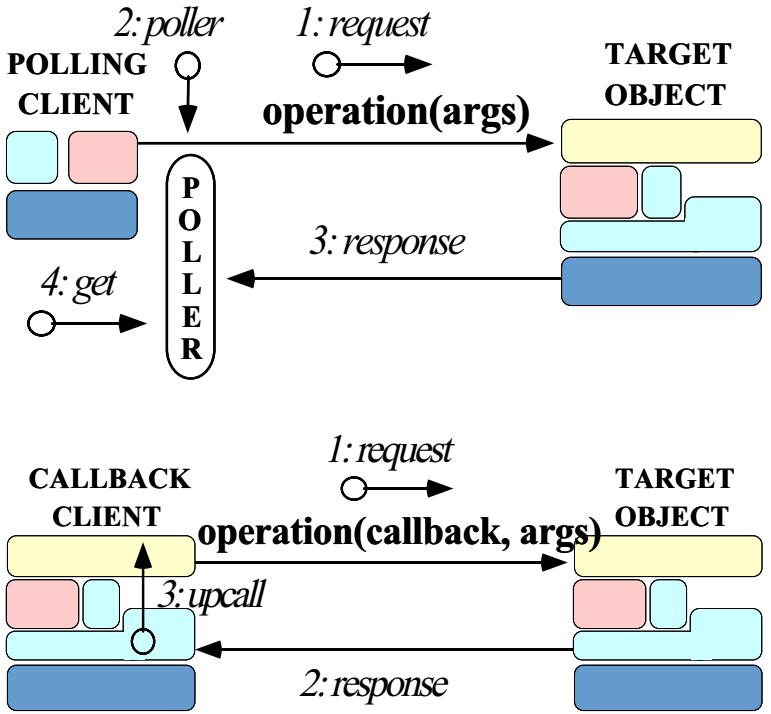
  1. Increase the number of client threads
     - *e.g.*, due to synchronous two-way communication
  2. Increase the end-to-end latency for multiple requests
     - *e.g.*, due to blocking on certain long-delay operations
  3. Decrease OS/network resource utilization
     - *e.g.*, inefficient support for bulk data transfers

Limitations with Workarounds for CORBA's Lack of Asynchrony

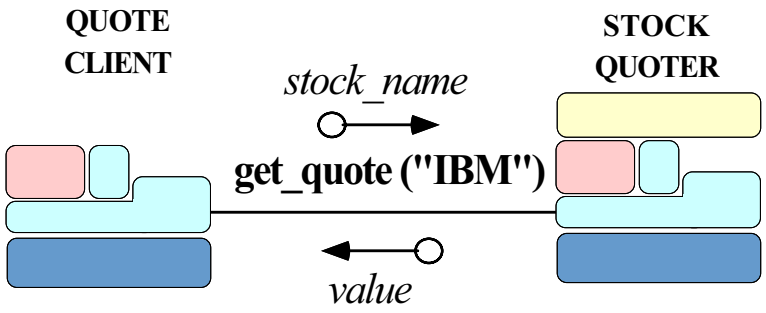
- *Synchronous method invocation* (SMI) multi-threading
  - Often non-portable, non-scalable, and inefficient
- Oneway operations
  - Best-effort semantics are unreliable
  - Requires *callback* objects
  - Applications must match callbacks with requests
- Deferred synchronous
  - Uses DII, thus *very* hard to program
  - Not type-safe

OMG Solution → CORBA Messaging Specification

- Defines *QoS Policies* for the ORB
  - Timeouts
  - Priority
  - Reliable one-ways
- Specifies two *asynchronous method invocation* (AMI) models
  1. Poller model
  2. Callback model
- Standardizes *time-independent invocation* (TII) model
  - Used for store/forward routers



AMI Callback Overview



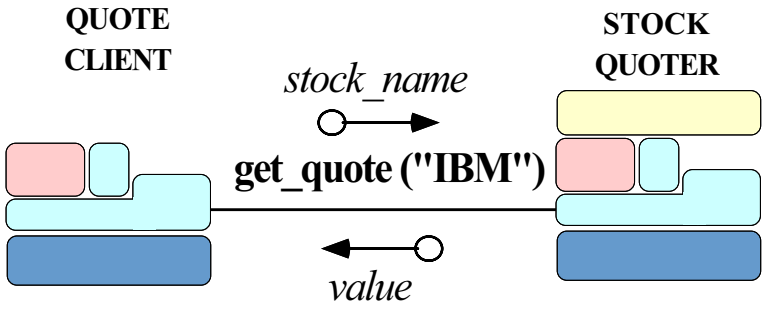
Quoter IDL Interface:

```
module Stock {
 interface Quoter {
   // Two-way operation to
   // get current stock value.
   long get_quote
     (in string stock_name);
 };
 // ...
}
```

Implied-IDL for client:

```
module Stock {
  // ReplyHandler.
  interface AMI_QuoterHandler
    : Messaging::ReplyHandler {
     // Callback method.
     void get_quote (in long return_value);
   };

 interface Quoter {
   // Two-way synchronous operation.
   long get_quote (in string stock_name);

   // Two-way asynchronous operation.
   void sendc_get_quote
      (in AMI_QuoterHandler handler,
       in string stock);
  };
};
```

Example: Synchronous Client



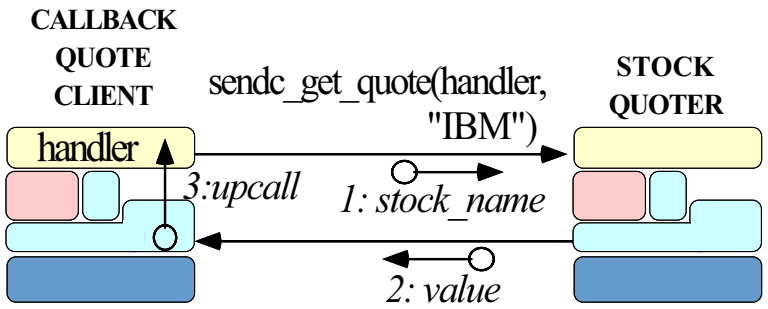
IDL-generated stub:

```
CORBA::ULong
Stock::Quoter::get_quote
  (const char *name)
{
  // 1. Setup connection
  // 2. Marshal
  // 3. Send request & wait
  // 4. Get reply
  // 5. Demarshal
  // 6. Return
};
```

Application:

```
// NASDAQ abbreviations for ORB vendors.
static const char *stocks[] =
{
  "IONA" // IONA Orbix
  "BEAS" // BEA Systems WLE
  "IBM"  // IBM Component Broker
}
// Set the max number of ORB stocks.
static const int MAX_STOCKS = 3;

// Make synchronous two-way calls.
for (int i = 0; i < MAX_STOCKS; i++) {
  CORBA::Long value =
    quoter_ref->get_quote (stocks[i]);
  cout << "Current value of "
       << stocks[i] << " stock: "
       << value << endl;
}
```

Example: AMI Callback Client (1/2)



Asynchronous stub:

```
void
Stock::Quoter::sendc_get_quote
       (AMI_QuoterHandler_ptr,
        const char *name)
{
 // 1. Setup connection
 // 2. Store reply handler
 //    in ORB
 // 3. Marshal
 // 4. Send request
 // 5. Return
};
```

Reply Handler Servant:

```
class My_Async_Stock_Handler
 : public POA_Stock::AMI_QuoterHandler {
public:
 My_Async_Stock_Handler (const char *s)
   : stock_ (s)
 { }
 ~My_Async_Stock_Handler (void) { }

 // Callback method.
 virtual void get_quote
   (CORBA::Long ami_return_val) {
   cout << stock_.in () << " stock: "
        << ami_return_val << endl;
    // Decrement global reply count.
    reply_count--;
  }
private:
  CORBA::String_var stock_;
};
```

Example: AMI Callback Client (2/2)

```
// Global reply count
int reply_count = MAX_STOCKS;

// Servants.
My_Async_Stock_Handler *
  handlers[MAX_STOCKS];

// Objrefs.
Stock::AMI_QuoterHandler_var
  handler_refs[MAX_STOCKS];

int i;

// Initialize ReplyHandler
// servants.
for (i = 0; i < MAX_STOCKS; i++)
 handlers[i] = new
 My_Async_Stock_Handler (stocks[i]);
```

```
// Initialize ReplyHandler object refs.
for (i = 0; i < MAX_STOCKS; i++)
  handler_refs[i] =
    handlers[i]->_this ();

// Make asynchronous two-way calls
// using the callback model.
for (i = 0; i < MAX_STOCKS; i++)
  quoter_ref->sendc_get_quote
    (handler_refs[i].in (),
     stocks[i]);

// ... activate POA manager ...

// Event loop to receive all replies.
while (reply_count > 0)
  if (orb->work_pending ())
    orb->perform_work ();
  else
    ...
```

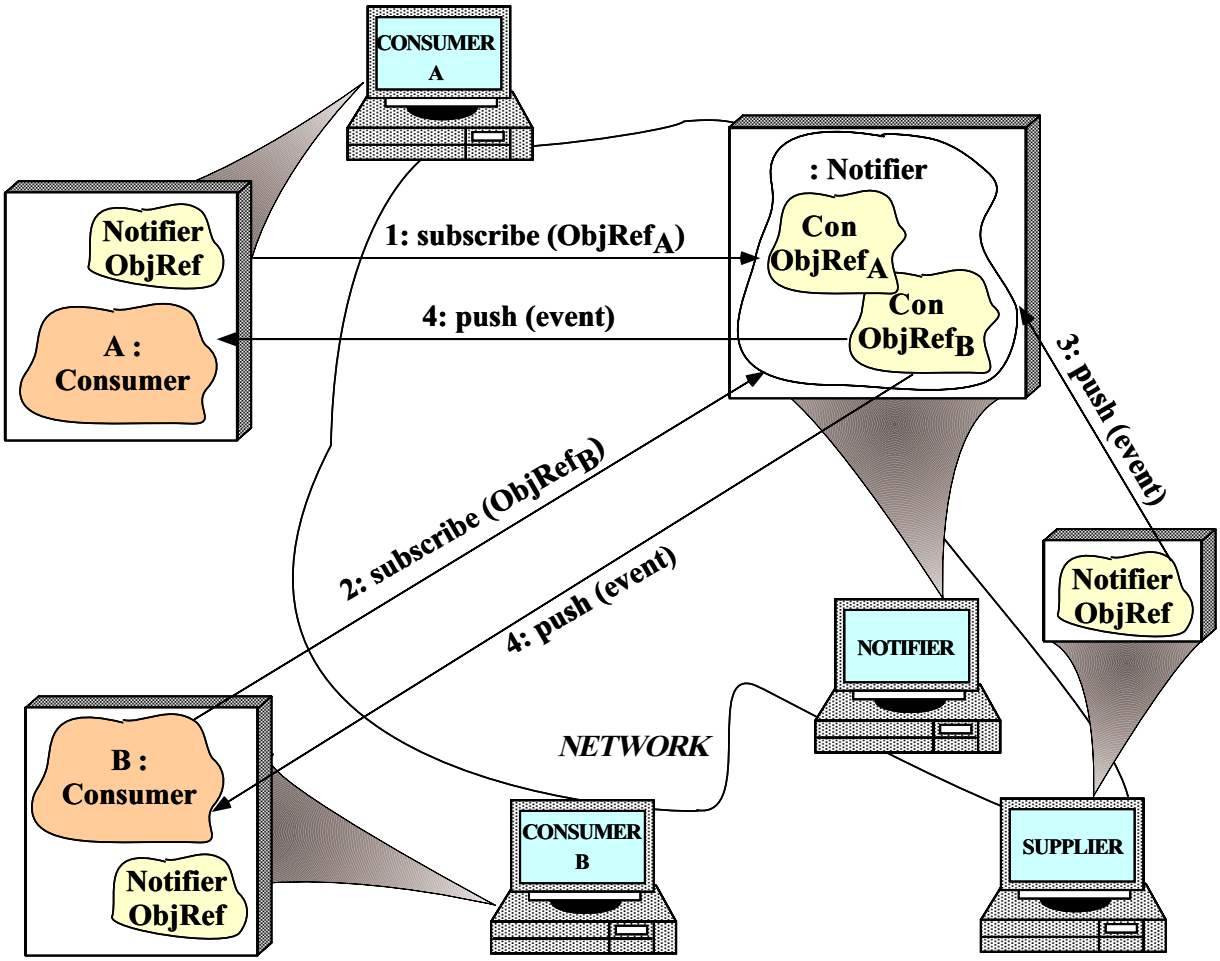
Additional Information on AMI

- Messaging specification is integrated into CORBA spec.
  - www.omg.org
- See Vinoski's CACM article on CORBA 3.0 for more info.
  - www.cs.wustl.edu/∼schmidt/vinoski-98.pdf
- See our papers on AMI
  - www.cs.wustl.edu/∼schmidt/report-doc.html
  - www.cs.wustl.edu/∼schmidt/PDF/ami1.pdf
  - www.cs.wustl.edu/∼schmidt/PDF/ami2.pdf
- See TAO release to experiment with working AMI examples
  - `$TAO_ROOT/tests/AMI/`

Motivation for a Publisher/Subscriber Architecture

- To this point, our stock quoter service has required the client to “poll” the server periodically to receive the latest quote value
  - However, this design is inefficient since the client keeps contacting the server, even if nothing has changed!
- A more scalable approach may be to use the *Publisher/Subscriber architectural pattern*
  - This pattern decouples the publishers who produce quote events from subscribers who consume them
- We’ll redesign our stock quoter application to implement the Publisher/Subscriber pattern using object references

A Publisher/Subscriber Stock Quoter Architecture



Note the use of the Publisher/Subscriber pattern

Event Type

- We define an `Event struct` that contains a `string` and an `any`:

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

- This maps to the following C++ class

```
struct Event {
  TAO::String_mgr topic_;
  CORBA::Any value_;
};
```

- The `TAO::String_mgr` behaves like a `String_var` that's initialized to the empty string
  - Do *not* use the `TAO::String_mgr` in your application code since it's explicitly designed to be ORB-specific!!!

Overview of the CORBA Any Type (1/2)

- OMG IDL defines type any for use with applications that can only determine the types of their data at runtime
- This type contains the following pair of fields:
  - The TypeCode that describes the type of the value in the any in order to ensure typesafety
  - The current value of the any
- The client ORB stores the TypeCode before the value so that the server ORB can properly decode and interpret the value

Overview of the CORBA Any Type (2/2)

- IDL any maps to the C++ class CORBA::Any:

```
class Any {
public:
  Any (); // Constructs an Any that contains no value.
  Any (const Any &); // Deep copy semantics
  Any &operator= (const Any &); // Deep copy semantics
  // ...
```

- Built-in types are inserted and extracted using overloaded operator<<= and operator>>=, respectively
  - The insertion operators copies the value and sets the typecode
  - The extract operators return true *iff* the extraction succeeds, *i.e.*, if the typecodes match!
- The IDL compiler generates these overloaded operators for user-defined types, as shown later in a DII example

Event Receiver Interface

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

  // Disconnect the Consumer
  // from the Notifier.
  void disconnect (in string reason);
};
```

A `Consumer` is called back by the Notifier

Overview of Oneway Operations

- The `push()` operations in `Consumer` and `Notifier` interfaces are *oneway*
  - They must therefore have `void` return type, only `in` parameters, and no `raises` clause
- By default, oneway operations have "best effort" semantics
  - *i.e.*, there is no guarantee they will be delivered in the order sent or that they'll even be delivered at all!
- Later versions of CORBA define so-called "reliable oneways," which address some of these issues via the `SyncScope` policy
  - *e.g.*, SYNC_NONE, SYNC_WITH_TRANSPORT, SYNC_WITH_SERVER, and SYNC_WITH_TARGET

Notifier Interface

```
interface Notifier {
  // = For Consumers.
  // Subscribe a consumer to receive
  // events that match filtering_criteria
  // applied by notifier.  Returns consumer
  long subscribe
    (in Consumer consumer,
     in string filtering_criteria);
  // Remove a particular consumer.
  void unsubscribe (in long consumer_id);

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

A Notifier publishes `Events` to subscribed Consumers

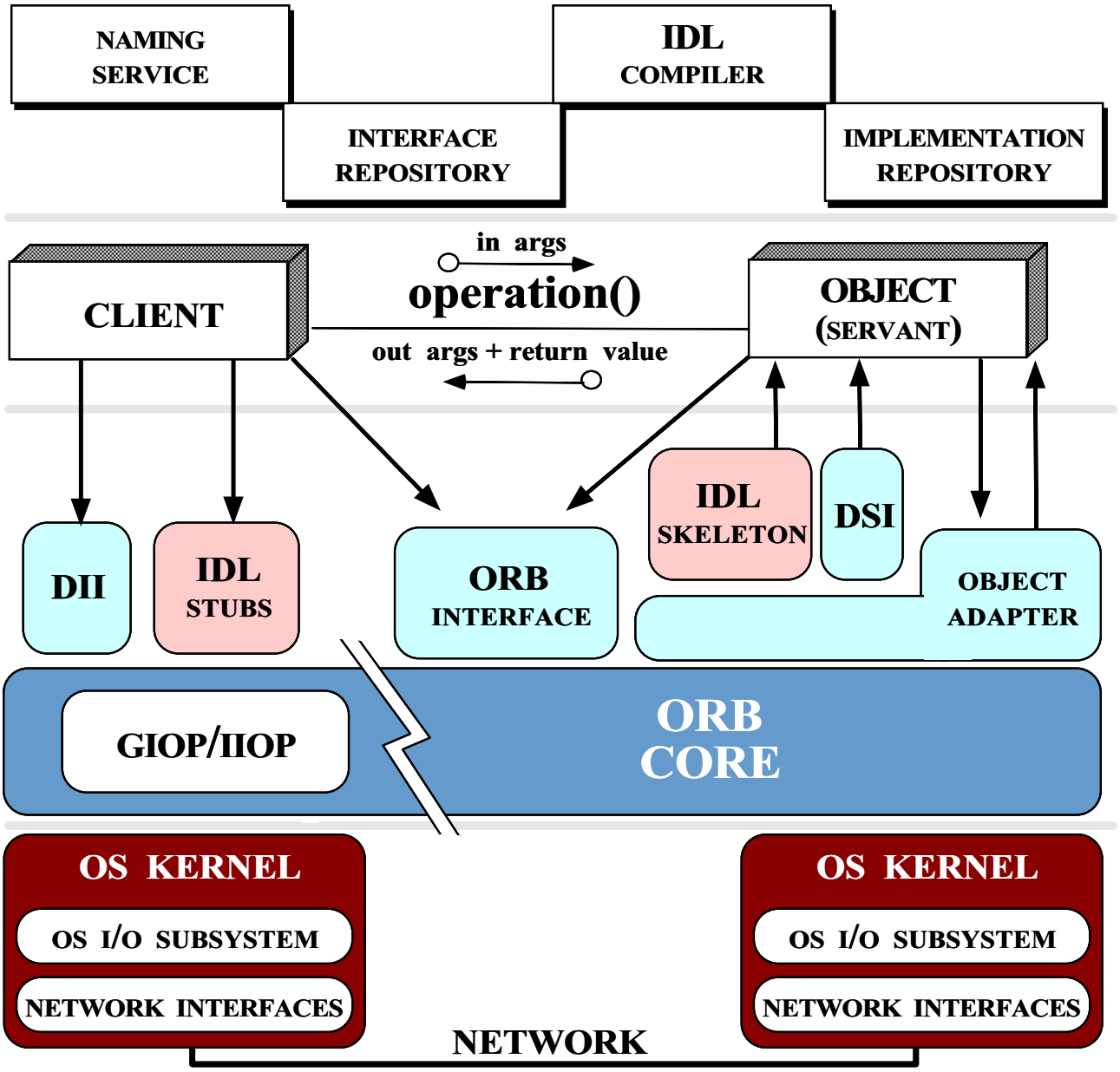
Limitations of Object References

- Note that the `Notifier::subscribe()` operation returns a *consumer ID* that the `unsubscribe()` operation uses to remove the subscription of a particular consumer
- We need this ID since it's invalid to compare objects for equality directly using object references, *i.e.*:
  - Object references only indicate location, *not* object identity
  - `Object::is_equivalent()` is a *local* operation that tests object *reference* identity, not *object* identity!!
- Other invalid operations on object references include
  - Using C++ `dynamic_cast` rather than `_narrow()`
  - Testing for NULL rather than using `CORBA::is_nil()`

Notifier Implementation

```
class My_Notifier { // C++ pseudo-code, error checking omitted...
public:
  CORBA::Long subscribe (Consumer_ptr con, const char *fc) {
    // Not using _duplicate is FMM 5
    consumer_set_.bind (fc, Consumer::_duplicate (con));
    return consumer_id;
  }
  void unsubscribe (CORBA::Long con_id) {
    Consumer_var con;
    // FMM 5 is to use _ptr; _var needed since _ptr's in map.
    consumer_set_.unbind (fc, con);
    remove <con_id> from <consumer_set_>.
  }
  void push (const Event &event) {
    foreach <consumer> in <consumer_set_>
      if (event.topic_ matches <consumer>.filter_criteria)
        <consumer>.push (event);
  }
private: // e.g., use an STL map.
  std::map <string, Consumer_ptr> consumer_set_;
};
```

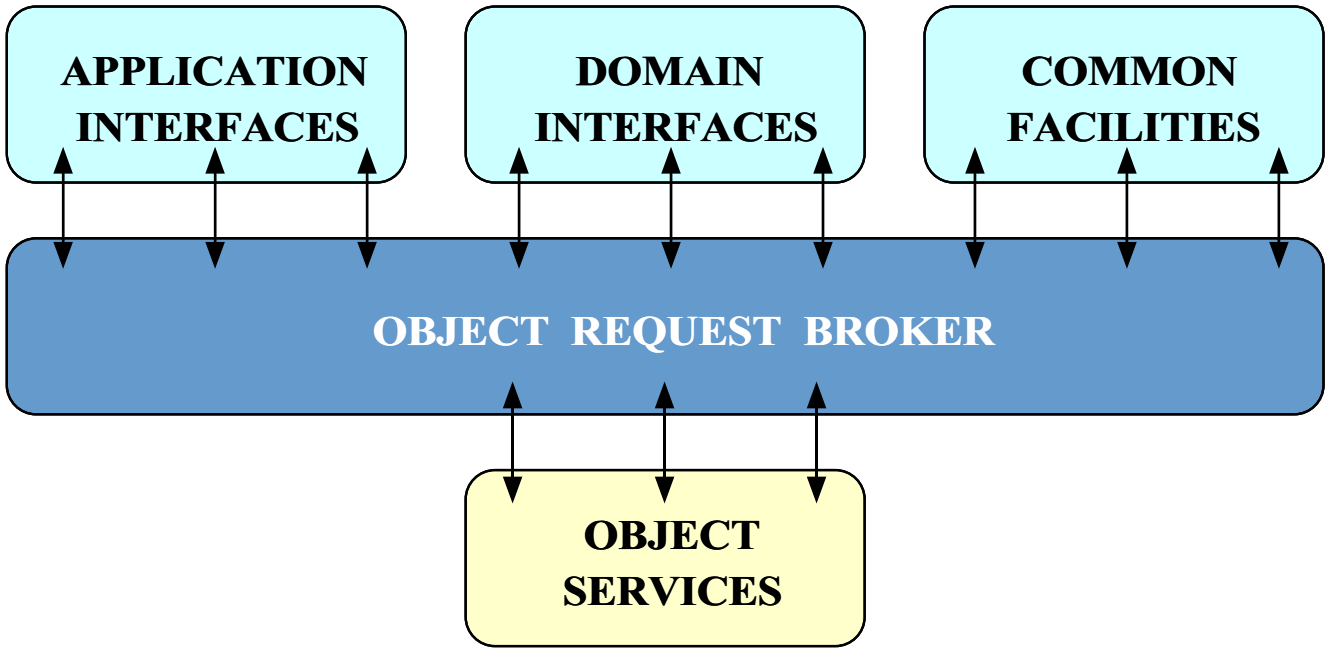
CORBA ORB Architecture



Overview of CORBA Components

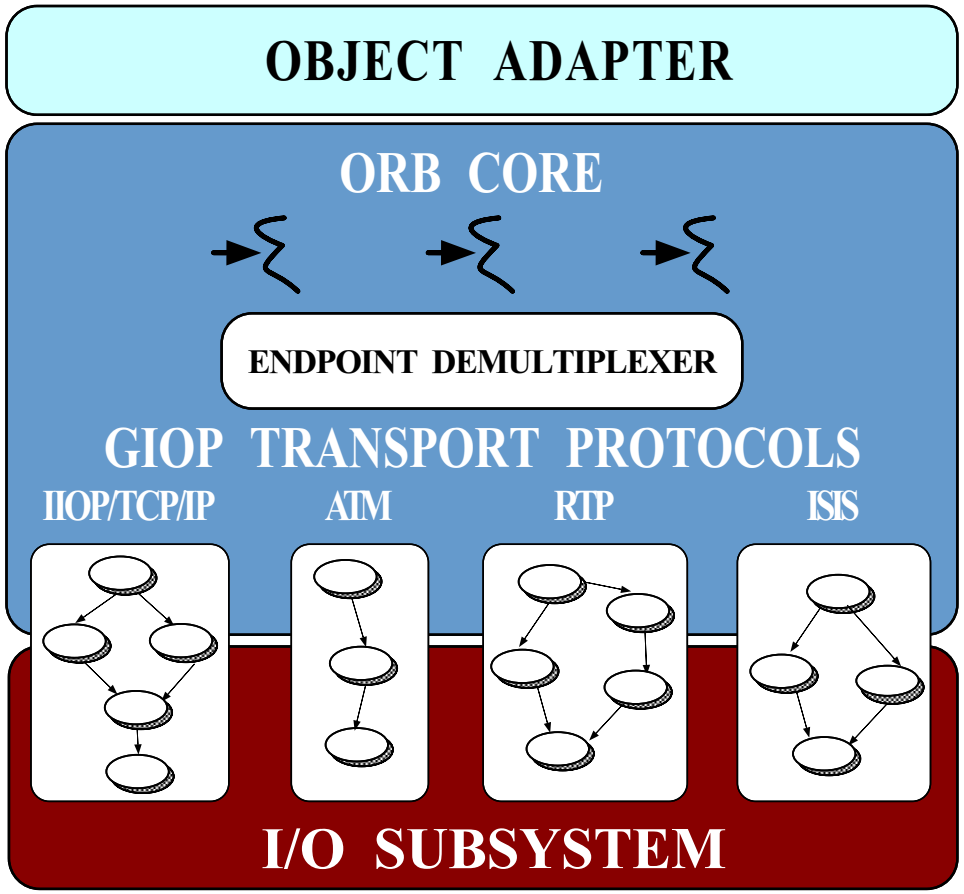
- The CORBA specification contains the following major components:
  - Object Request Broker (ORB) Core
  - Interoperability Spec (GIOP and IIOP)
  - Interface Definition Language (IDL)
  - Programming language mappings for IDL
  - Static Invocation Interface (SII)
  - Dynamic Invocation Interface (DII)
  - Static Skeleton Interface (SSI)
  - Dynamic Skeleton Interface (DSI)
  - Portable Object Adapter (POA)
  - Implementation Repository
  - Interface Repository

OMA Reference Model Interface Categories



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

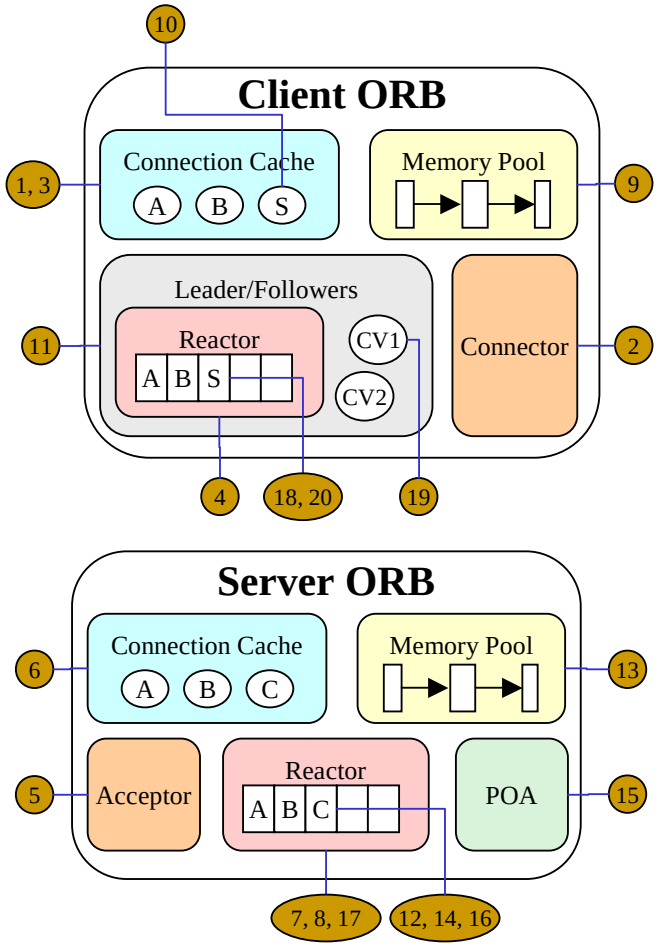
Overview of the ORB Core



Features

- Connection/memory management
- Request transfer
- Endpoint demuxing
- Concurrency control

Tracing a Request Through a CORBA ORB



Request invocation phases

1. Client ORB connection management
2. Server ORB connection management
3. Client invocation for twoway calls
4. Server processing for twoway calls
5. Client reply handling for twoway calls

Client ORB Connection Management

The following are the activities a client ORB performs to create a connection actively when a client application invokes an operation on an object reference to a target server object:

1. Query the client ORB's connection cache for an existing connection to the server designated in the object reference on which the operation is invoked
2. If the cache doesn't contain a connection to the server, use a connector factory to create a new connection S
3. Add the newly established connection S to the connection cache
4. Also add connection S to the client ORB's reactor since S is bi-directional and the server may send requests to the client using S

Server ORB Connection Management

The server ORB activities for accepting a connection passively include:

5. Use an acceptor factory to accept the new connection C from the client
6. Add C to the server ORB's connection cache since C is bi-directional and the server can use it to send requests to the client
7. Also add connection C to the server ORB's reactor so the server is notified when a request arrives from the client
8. Wait in the reactor's event loop for new connection and data events

Client Invocation of Synchronous Twoway Operation

We now describe the steps involved when a client invokes a synchronous two-way request to a server:

9. Allocate a buffer from a memory pool to marshal the parameters in the operation invocation
10. Send the marshaled data to the server using connection S. Connection S is locked for the duration of the transfer
11. Use the leader/followers manager to wait for a reply from the server. Assuming that a leader thread is already available, the client thread waits as a follower on a condition variable or semaphore.[1]

[1] The leader thread may actually be a server thread waiting for incoming requests or another client thread waiting for its reply

Server Processing for Twoway Operation

The server ORB activities for processing a request are described below:

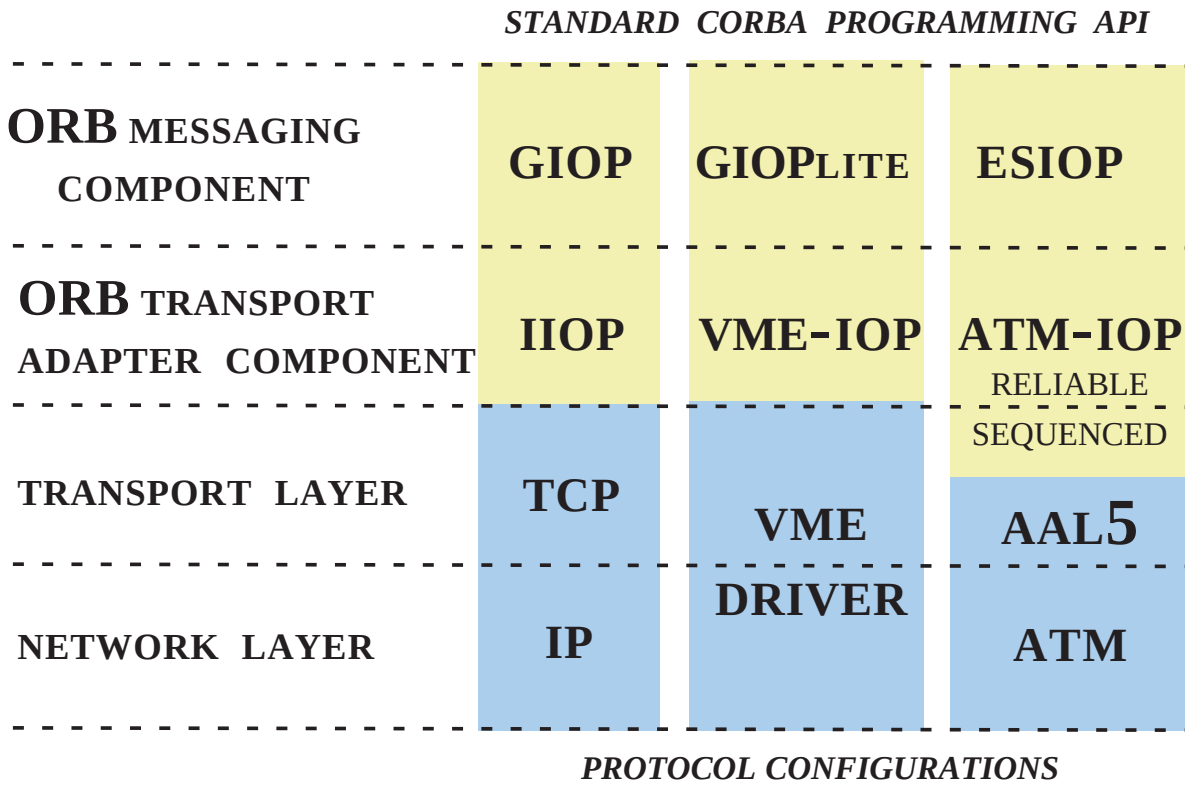
12. Read the header of the request arriving on connection C to determine the size of the request
13. Allocate a buffer from a memory pool to hold the request
14. Read the request data into the buffer
15. Demultiplex the request to find the target portable object adapter (POA), servant, and skeleton – then dispatch the designated upcall to the servant after demarshaling the request parameters
16. Send the reply (if any) to the client on connection C, connection C is locked for the duration of the transfer
17. Wait in the reactor's event loop for new connection and data events

Client Reply Handling for Twoway Operation

Finally, the client ORB performs the following activities to process a reply from the server:

18. The leader thread reads the reply from the server on connection S
19. After identifying that the reply belongs to the follower thread, the leader thread hands off the reply to the follower thread by signaling the condition variable used by the follower thread
20. The follower thread demarshals the parameters and returns control to the client application, which processes the reply

CORBA Interoperability Protocols



- GIOP
  - Enables ORB-to-ORB interoperability
- IIOP
  - Works directly over TCP/IP, no RPC
- ESIOPs
  - *e.g.*, DCE, DCOM, wireless, etc.

Overview of GIOP and IIOP

- 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.*, shared connections
  - Ordering constraints are minimal, *i.e.*, can be asynchronous
- Message formats
  - Client: `Request`, `CancelRequest`, `LocateRequest`
  - Server: `Reply`, `LocateReply`, `CloseConnection`
  - Both: `MessageError`
- IIOP is a mapping of GIOP over TCP/IP

Example GIOP Format

```
module GIOP {
  enum MsgType {
    Request, Reply, CancelRequest,
    LocateRequest, LocateReply,
    CloseConnection, MessageError
  };

  struct MessageHeader {
    char magic[4];
    Version GIOP_version;
    octet byte_order; // Fragment bit in 1.1.
    octet message_type;
    unsigned long message_size;
  };

  struct RequestHeader {
    IOP::ServiceContextList service_context;
    unsigned long request_id;
    // Reliable one-way bits in 1.2
    boolean response_requested;
    sequence<octet> object_key;
    string operation;
    Principal requesting_principal;
  };
  // ...
```

Overview of Interface Definition Languages (IDLs)

- Motivation
  - Developing flexible distributed applications on heterogeneous platforms requires
    - An interface contract between client and server that defines permissible operations and types
    - Strict separation of *interface* from *implementation(s)*
- Benefits of using an IDL
  - Ensure platform independence $\rightarrow$ *e.g.*, Windows NT to UNIX
  - Enforce modularity $\rightarrow$ *e.g.*, separate concerns
  - Increase robustness $\rightarrow$ *e.g.*, eliminate common network programming errors
  - Enable language independence $\rightarrow$ *e.g.*, COBOL, C, C++, Java, etc.

Example 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* IDLs
  - These are more complicated to extend and reuse since they don't support inheritance

Overview of OMG IDL (1/2)

- 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++, C#, Java, Smalltalk, COBOL, Perl, etc.
- All data exchanged between clients and servers must be defined using OMG IDL

Overview of OMG IDL (2/2)

- OMG IDL is similar to Java `interfaces` or C++ abstract classes
  - *i.e.*, it defines the interface and type "contracts" that clients and servers must agree upon to exchange data correctly and efficiently
- OMG IDL is *not* a complete programming language, however
  - *e.g.*, it is purely declarative and can not be used to define object state or perform computations
- IDL source files must end with the `.idl` suffix

OMG IDL Features

- OMG IDL is similar to C++ and Java
  - *e.g.*, comment styles, identifiers, built-in types, etc.
- OMG IDL supports the following features:
  - `modules` and `interfaces`
  - Operations and Attributes
  - Single and multiple inheritance
  - Fixed-size basic types (*e.g.*, `double`, `long`, `char`, `octet`, etc).
  - Arrays and `sequence`
  - `struct, enum, union, typedef`
  - `consts`
  - `exceptions`

OMG IDL Differences from C++ and Java

- Case-insensitive
- No control constructs
- No data members (*cf* valuetypes)
- No pointers (*cf* valuetypes)
- No constructors or destructors
- No overloaded operations
- No `int` data type
- Contains parameter passing modes
- Unions require a tag
- Different String type
- Different Sequence type
- Different exception interface
- No templates
- `oneway` call semantics
- `readonly` keyword
- `any` type

Using OMG IDL Interfaces Effectively

- The CORBA specification and services are defined using IDL
- Interfaces described using OMG IDL can also be application-specific
  - *e.g.*, databases, spreadsheets, spell checkers, network managers, air traffic control systems, documents, medical imaging systems, etc.
- Objects may be defined at any level of granularity
  - *e.g.*, from fine-grained GUI objects to multi-megabyte multimedia "Blobs"
- It's essential to remember that *distributed* objects will incur higher latency than *collocated* objects
  - Interfaces designed for purely stand-alone applications may therefore require reengineering

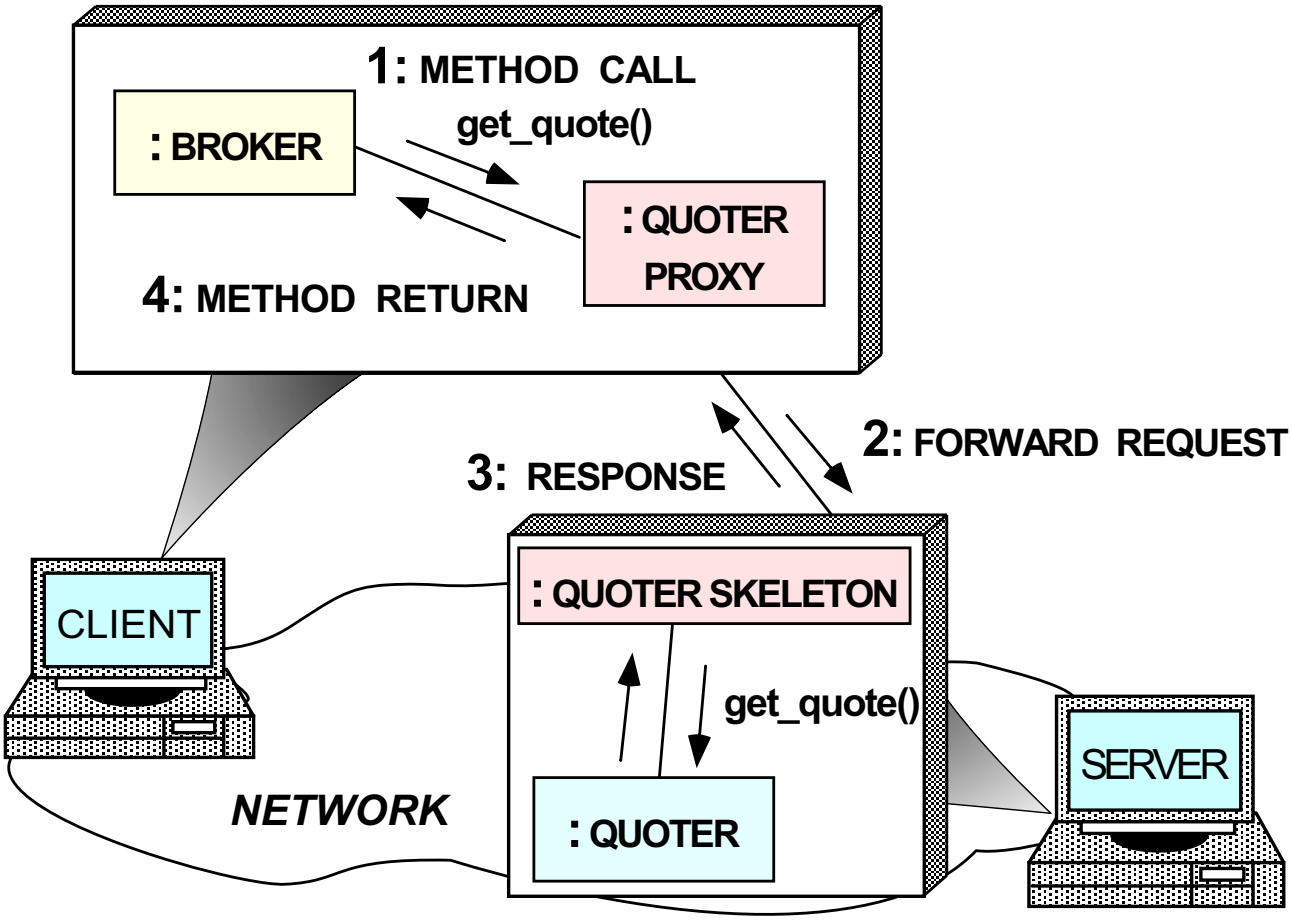
Static Invocation Interface (SII)

```
// Get object reference.
Quoter_var quoter = // ...

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

CORBA::Long value =
  quoter->get_quote (name);
cout << name << " = "
     << value << endl;
```

- The common way to use OMG IDL is the "Static Invocation Interface" (SII)
- All operations are specified in advance and are known to client via *stubs*
  - Stubs marshal operation calls into request messages

- The advantages of SII are *simplicity*, *typesafety*, and *efficiency*
- The disadvantage of SII is its *inflexibility* (and potentially its footprint)

SII Stubs use the 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 operations 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 an ASCII name
    - Type-checking via meta-info in “Interface Repository”
- The DII is more flexible than the SMI SII
  - *e.g.*, it supports *deferred synchronous* invocation and enables dynamic dispatching of operations
- However, the DII is also more complicated, less typesafe, and inefficient

An Example DII Client

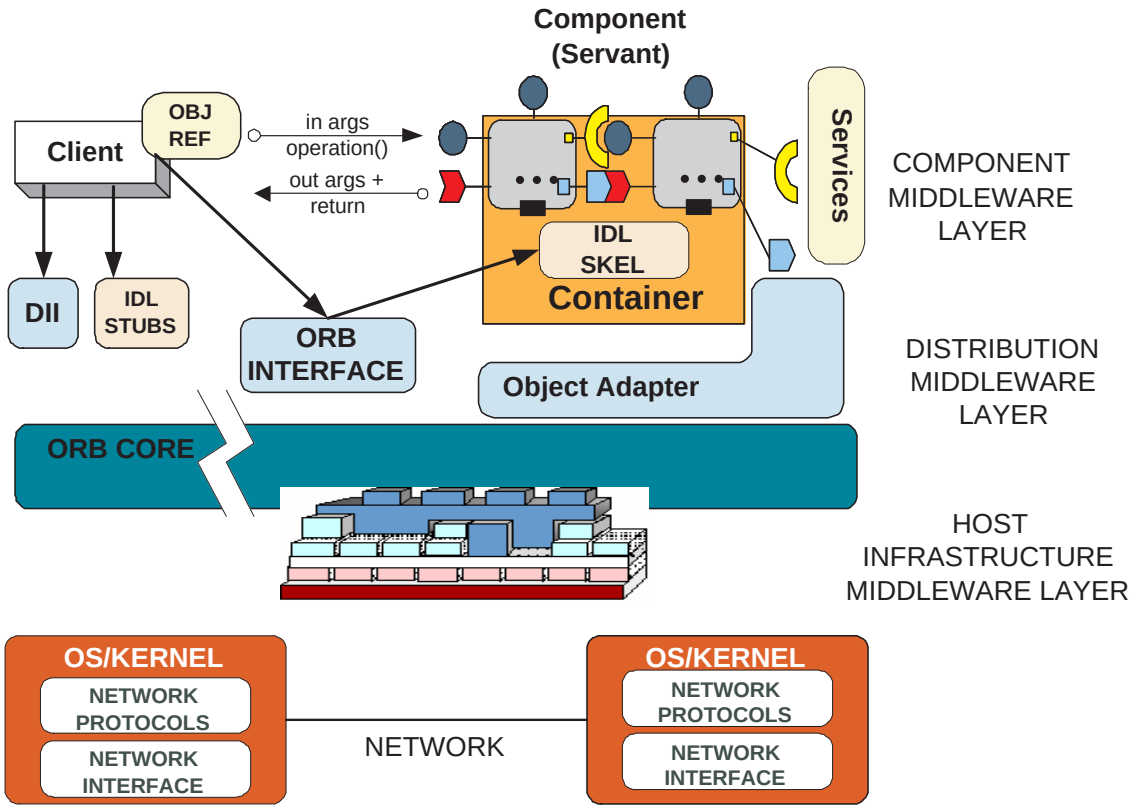
```
// Get Quoter reference.
Stock::Quoter_var quoter_ref = // ...
CORBA::Long value;

// Create request object.
CORBA::Request_var request =
  quoter_ref->_request ("get_quote");

// Add parameter using insertion operation,
// which makes a ‘‘deep copy’’ and sets
// typecode to ‘‘unbounded string.’’
request->add_in_arg () <<= "IONAY";
request->set_return_type (CORBA::_tc_long);

request->invoke (); // Call method.

// Retrieve/print value using extraction
// operator, which makes a ‘‘shallow copy.’’
if (request->return_value () >>= value)
  cout << "Current value of IONA stock: "
       << value << endl;
```

- The DII example above is more complicated and inefficient than the earlier SII example
- www.cs.wustl.edu/∼schmidt/report-doc.html has more information on DII

Static and Dynamic Skeleton Interface

- The Static Skeleton Interface (SSI) is generated automatically by the IDL compiler
  - The SII performs the operation demuxing/dispatching and parameter demarshaling
- The Dynamic Skeleton Interface (DSI) provides analogous functionality for the server that the DII provides on the client
  - It is defined primarily to build ORB “Bridges”
  - The DSI lets server code handle arbitrary invocations on CORBA objects
  - The DSI requires the use of certain POA features

Advanced CORBA Features

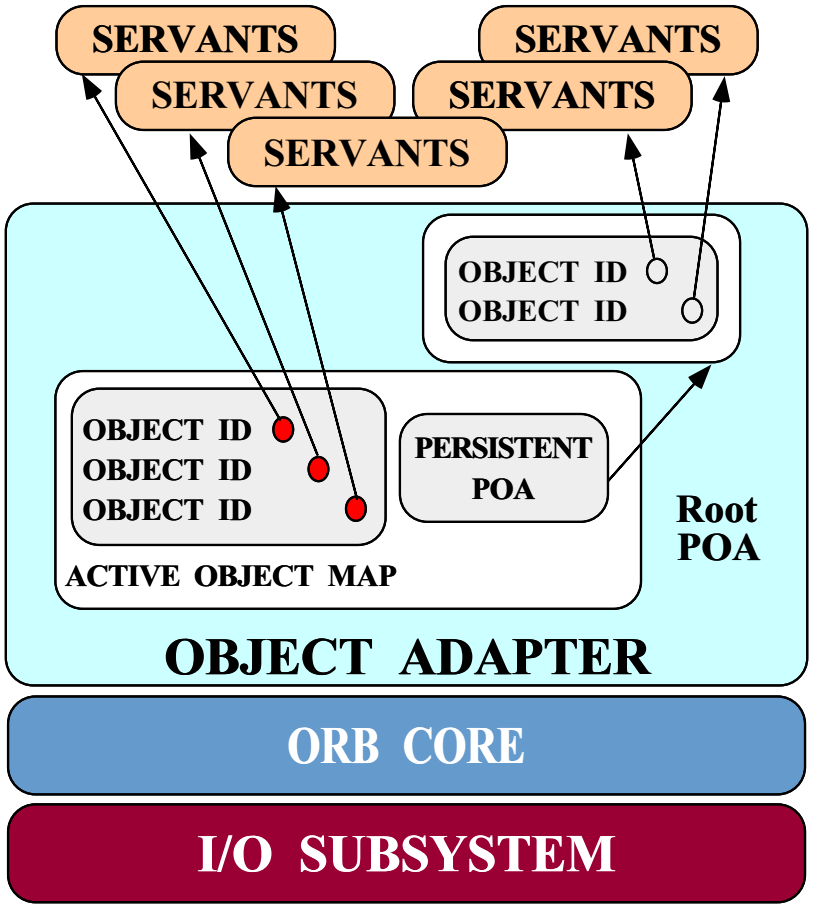


Features

- Portable Object Adapter
- Multi-threading
- Implementation Repository
- CORBA Component Model

www.cs.wustl.edu/~schmidt/corba.html

Overview of the Portable Object Adapter (POA)



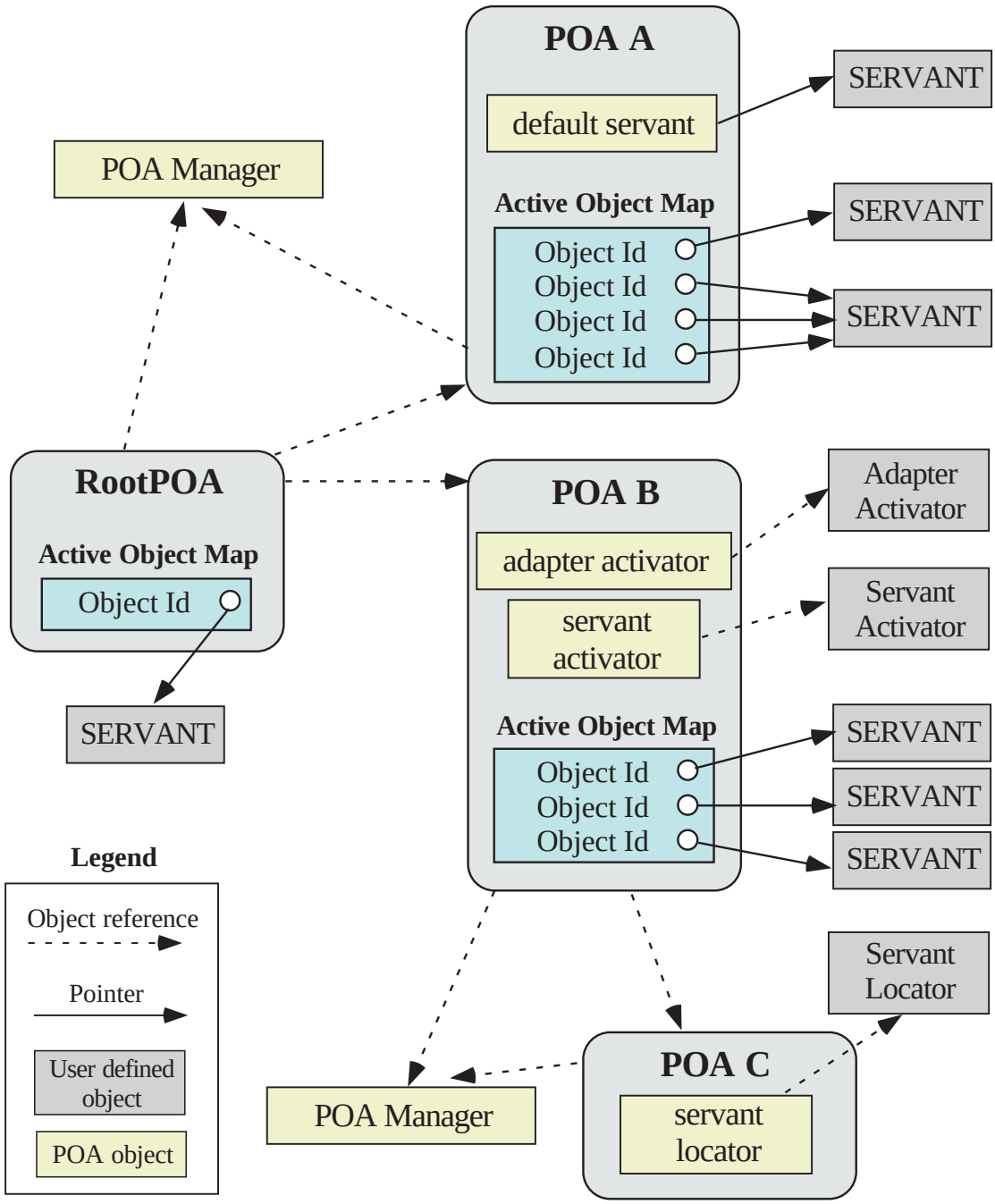
`PortableServer` interfaces

- `POA`
- `POAManager`
- `Servant`
- POA Polices
- Servant activators and servant locators
- `POACurrent`
- `AdapterActivator`

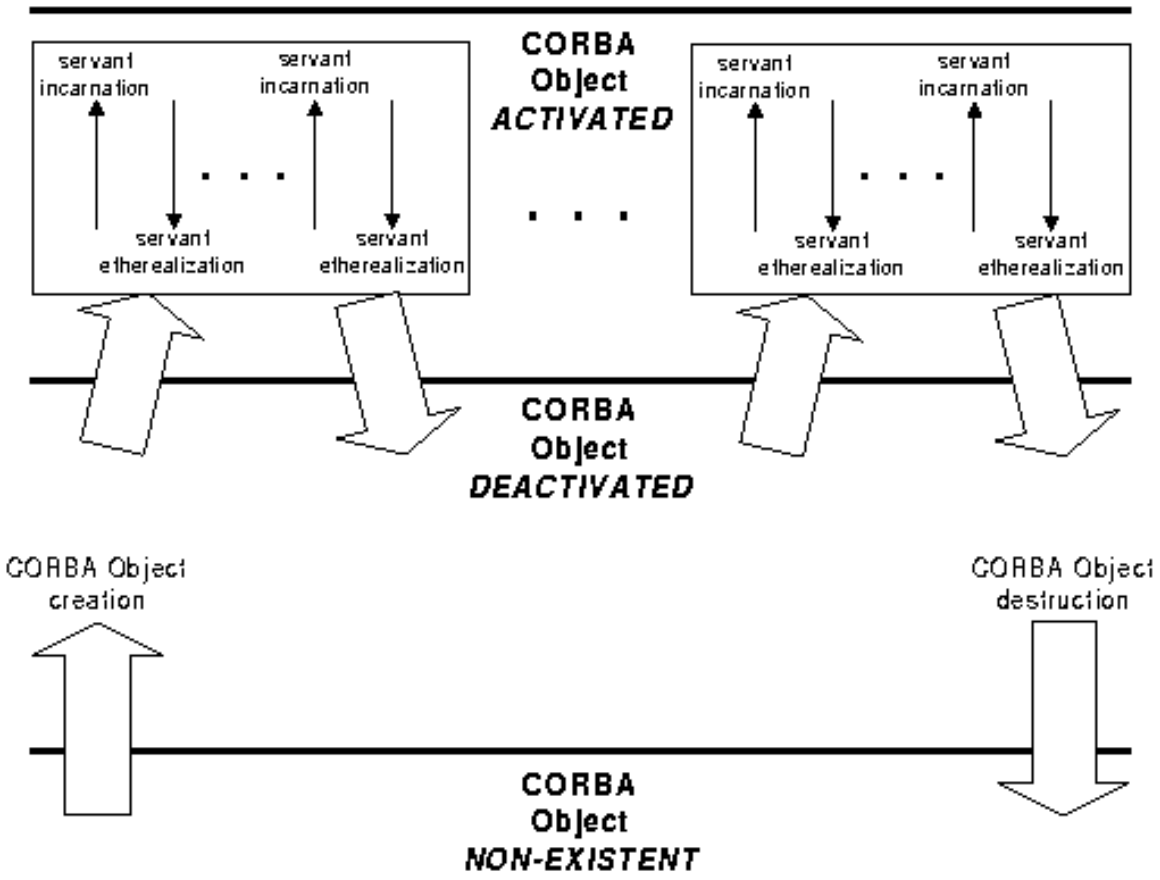
Design Goals of the Portable Object Adapter

- Servants that are portable between ORBs
- Objects with persistent & transient identities
- Transient objects with minimal programming effort and overhead
- Transparent activation & deactivation of servants
- Implicit and explicit servant activation
- A single servant can support multiple object identities
- Multiple (nested) instances of the POA in a server process
- POA behavior is dictated by creation policies
- Servants can inherit from skeletons or use DSI

The POA Architecture



Object Lifecycle for a POA



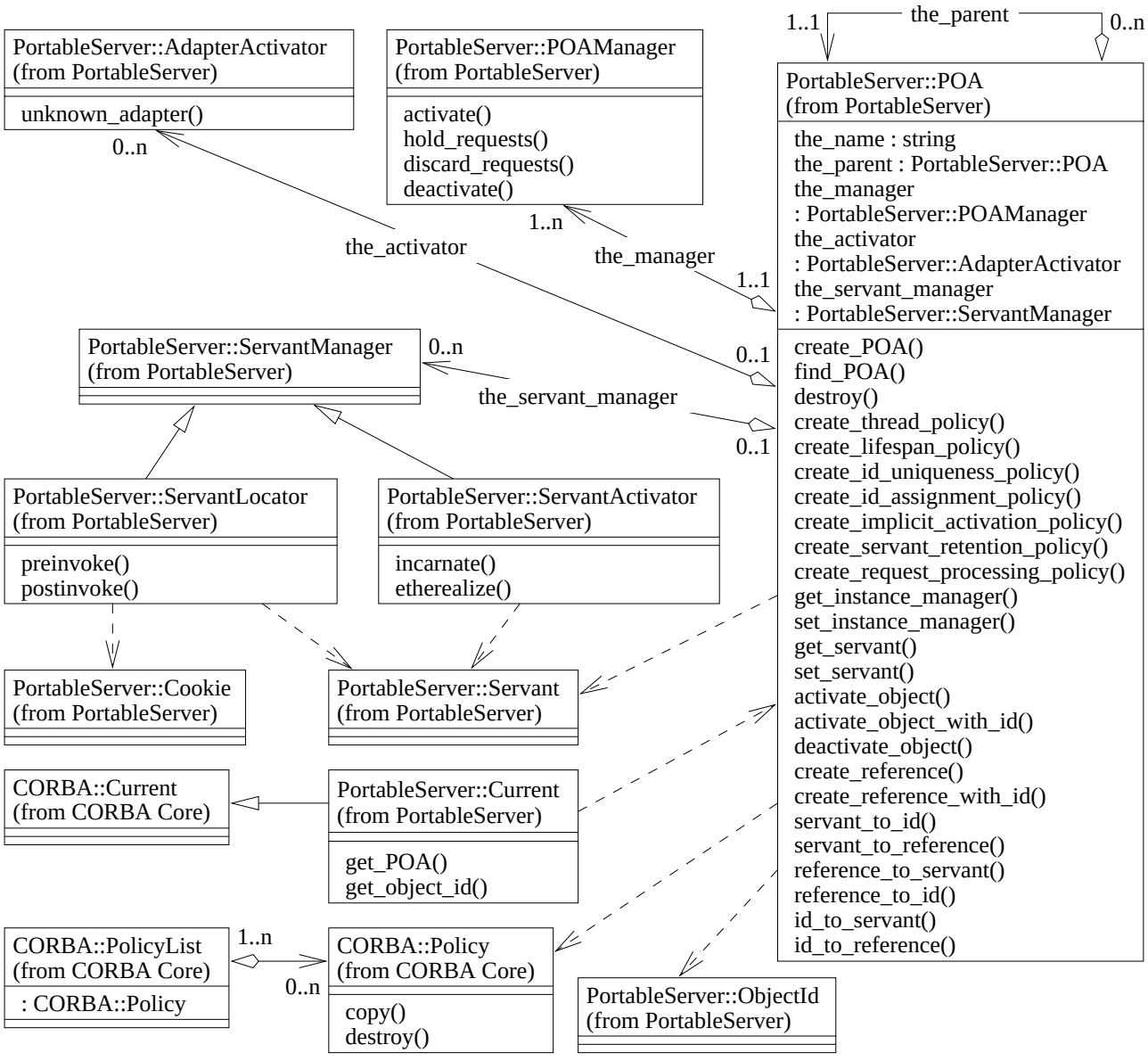
Overview of Object IDs

- Object IDs are the value used by the POA and by the ORB implementation to identify particular CORBA objects
  - Object ID values may be assigned by the POA or by the application
  - Object ID values are encapsulated by references and hidden from clients
  - Object ID have no standard form; they are managed by the POA as uninterpreted octet sequences
- An object reference encapsulates an object Id, a POA identity, and transport profiles

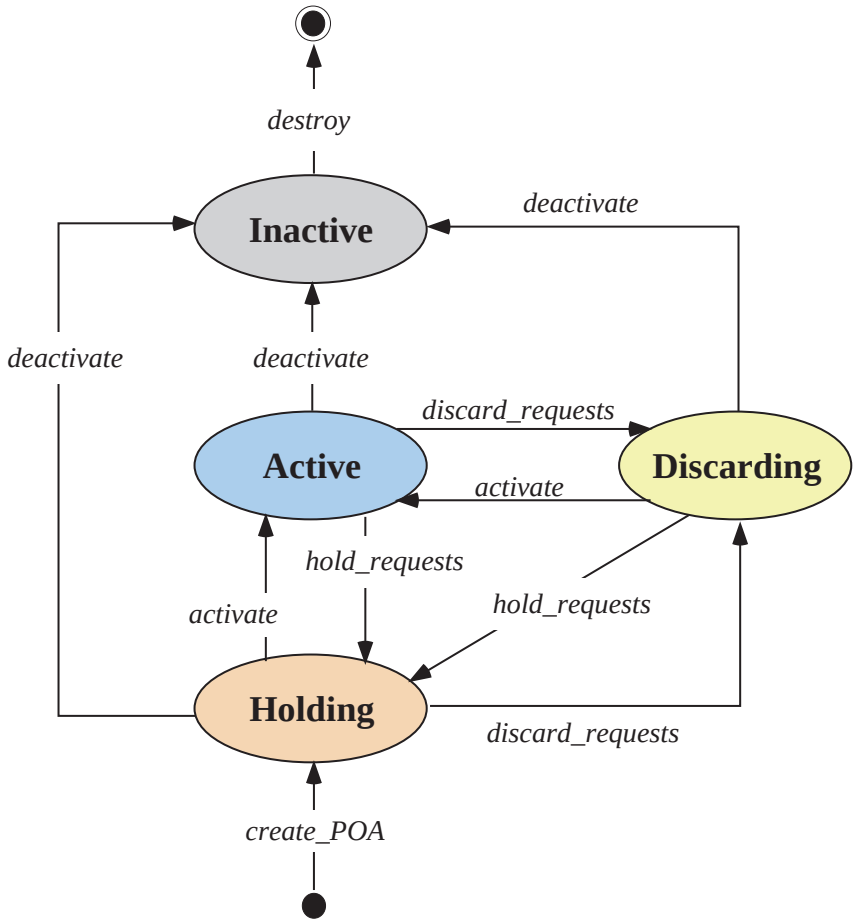
Overview of POAs

- POAs form a hierarchical namespace for objects in servers
  - *i.e.*, a namespace for object ids and child POAs
- Each servant belongs to one POA, but a POA can contain many servants
- A POA is a manager for object lifecycles, *e.g.*:
  - A factory for creating object refs
  - Activates and deactivates objects
  - Etherealizes and incarnates servants
- A POA maps client requests to servants
- POA policies specify the characteristics of a child POA when it is created

POA Architecture in UML



Overview of the POA Manager



- Encapsulates the processing state of associated POAs
- Can *dispatch*, *hold*, or *discard* requests for the associated POAs and deactivate POA(s)
  - The default state is “holding”
- A POA manager is associated with a POA at creation time and cannot be changed after creation

Overview of POA Policies

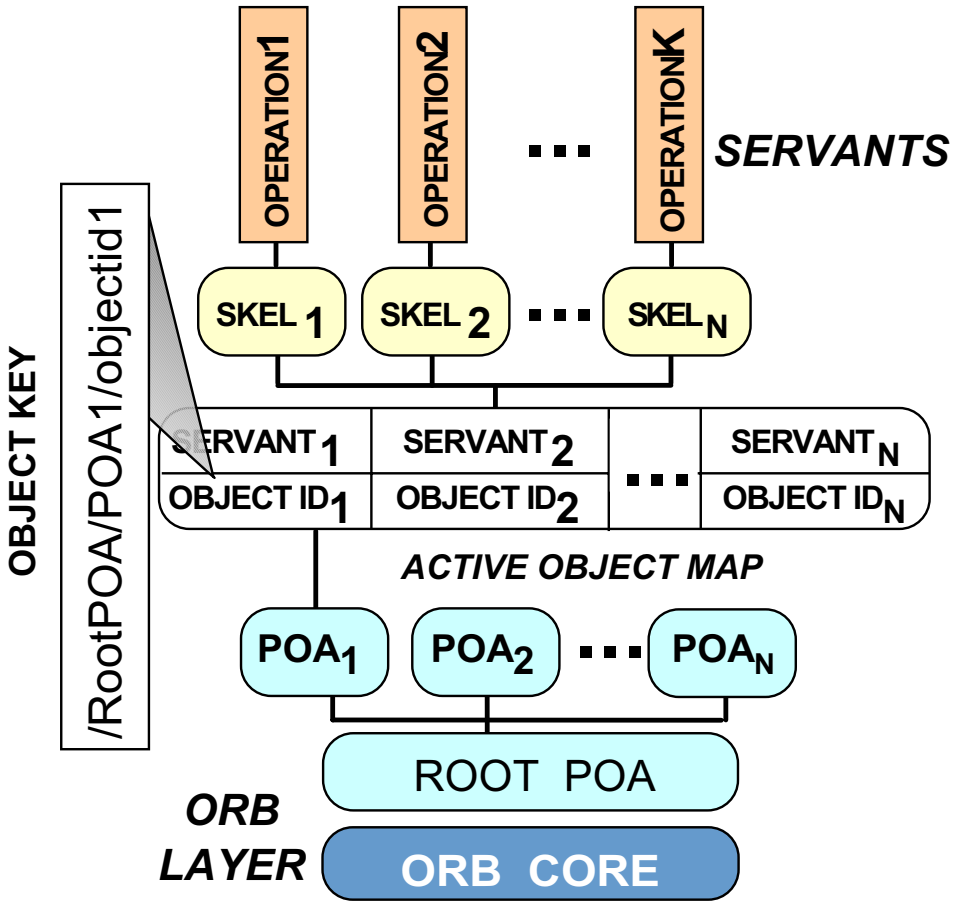
- When a POA is created, its behavior can be controlled by up to seven policies
  - *i.e.*, lifespan, ID assignment, ID uniqueness, implicit activation, request processing, servant retention, and thread policies
- These policies all inherit from the `CORBA::Policy` interface

```
module CORBA {
  typedef unsigned long PolicyType;
  interface Policy {
    readonly attribute PolicyType policy_type;
    Policy copy ();
    void destroy ();
  };
  typedef sequence<Policy> PolicyList;
  // ...
```

- The POA interface defines a factory method to create each policy

Overview of the Active Object Map

- By default, a POA contains an active object map (AOM)
- The object ID in the object key sent by the client is the index into the AOM



Overview of the Root POA

- The Root POA has a preordained set of policies that cannot be changed:
  - The lifespan policy is *transient*
  - The ID assignment policy uses *system IDs*
  - The ID uniqueness policy uses *unique IDs*
  - The implicit activation policy is *enabled* (not default)
  - The request processing policy uses an *active object map*
  - The servant retention policy *retains* servants
  - The thread policy gives the *ORB control*
- If these policies are inappropriate, you can create your own child POAs via the `PortableServer::POA::create_POA()` factory

Canonical Steps to Obtain the Root POA

```
// ORB is ‘‘locality constrained’’
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);

// Root POA is the default POA (locality constrained)
CORBA::Object_var obj =
  orb->resolve_initial_references ("RootPOA");

// Type-safe downcast.
PortableServer::POA_var root_poa
  = PortableServer::POA::_narrow (obj.in ());

// Activate the POA.
PortableServer::POA_Manager_var poa_manager =
  root_poa->the_POAManager ();
poa_manager->activate ();
// FMM 2
// root_poa->the_POAManager ()->activate ();
```

Overview of Implicit Activation Policy

- This policy controls whether a servant can be activated implicitly or explicitly

```
enum ImplicitActivationPolicyValue {
  IMPLICIT_ACTIVATION,
  NO_IMPLICIT_ACTIVATION /* DEFAULT */ };

interface ImplicitActivationPolicy : CORBA::Policy {
  readonly attribute ImplicitActivationPolicyValue value;
}
```

- When the `IMPLICIT_ACTIVATION` policy value is used with `RETAIN` and `SYSTEM_ID` policy values servants are added to the AOM by calling `_this()`
- The `NO_IMPLICIT_ACTIVATION` policy value requires servants to be activated via one of the `POA::activate_object*()` calls

Implict Activation with System IDs

This example illustrates `_this()`:

```
interface Quoter { // ... IDL
  long get_quote (in string stock_name)
    raises (Invalid_Stock);
};

// Auto-generated for use by servants.
class My_Quoter : public virtual POA_Stock::Quoter
{
public:
  // ...
  CORBA::Long get_quote (const char *stock_name);
};

My_Quoter *quoter = new My_Quoter;

// FMM 6 -- not transfering the ownership to
// PortableServer::ServantBase_var
// Implicit activation with system ID
CORBA::Object_var objref = quoter->_this ();

PortableServer::POA_Manager_var poa_manager =
  root_poa->the_POAManager ();
poa_manager->activate ();
orb->run ();
```

Overview of ID Assignment Policy

- This policy controls whether object IDs are created by the ORB or by an application

```
enum IdAssignmentPolicyValue {
  USER_ID,
  SYSTEM_ID /* DEFAULT */
};

interface IdAssignmentPolicy : CORBA::Policy {
  readonly attribute IdAssignmentPolicyValue value;
}
```

- The `USER_ID` policy value works best with the `NO_IMPLICIT_ACTIVATION` and `PERSISTENT` policy values
- The `SYSTEM_ID` policy value works best with the `IMPLICIT_ACTIVATION` and `TRANSIENT` policy values

Overview of Lifespan Policy

- This policy controls whether object references are transcient or persistent

```
enum LifespanPolicyValue {
  PERSISTENT,
  TRANSIENT /* DEFAULT */
};

interface LifespanPolicy : CORBA::Policy {
  readonly attribute LifespanPolicyValue value;
}
```

- The `PERSISTENT` policy value works best with the `NO_IMPLICIT_ACTIVATION` and `USER_ID` policy values
- The `TRANSIENT` policy value works best with the `IMPLICIT_ACTIVATION` and `SYSTEM_ID` policy values

Creating a Child POA

We use the `PortableServer::POA::create_POA()` operation to create a new POA with the `USER_ID` and `PERSISTENT` policies

```
// FMM 4: Not calling length() is a mistake!
CORBA::PolicyList policies (2); policies.length (2);

policies[0] = root_poa->create_id_assignment_policy
  (PortableServer::IdAssignmentPolicy::USER_ID);

policies[1] = root_poa->create_lifespan_policy
  (PortableServer::LifespanPolicy::PERSISTENT);

PortableServer::POA_var child_poa =
  root_poa->create_POA
    ("child_poa",                              // New POA name
      PortableServer::POAManager::_nil (), // Non-shared POA manager
      policies);                               // New POA policies

for (CORBA::ULong i = 0; i != policies.length (); ++i)
  policies[i]->destroy ();
```

Explicit Activation with User IDs

This example illustrates `POA::activate_object_with_id()`:

```
// Create a new servant instance.
My_Quoter *quoter = new My_Quoter;

// Create a new user-defined object ID for the object.
PortableServer::ObjectId_var oid =
  PortableServer::string_to_ObjectId ("my quoter");

// Activate the object with the new object ID
child_poa->activate_object_with_id (oid.in (), quoter);
PortableServer::POA_Manager_var poa_manager =
  child_poa->the_POAManager ();
poa_manager ()->activate ();
// Run the ORB's event loop.
orb->run ();
```

Deactivating Objects

- There are certain steps to follow when deactivating objects

```
void My_Quoter::remove (void)
  throw (CORBA::SystemException,
         CosLifeCycle::LifeCycleObject::NotRemovable);
{
  PortableServer::POA_var poa = this->_default_POA ();
  PortableServer::ObjectId_var oid = poa->servant_to_id (this);
  // POA calls _remove_ref() on servant once all
  // operations are completed
  poa->deactivate_object (oid.in ());
}
```

- Calling `_remove_ref()` from the application could destroy the servant, but the POA has no knowledge of this and could potentially dispatch calls to the same servant since object entries in the active object map are still active and they haven't been invalidated

The Servant Retention Policy

- This policy controls whether a POA has an active object map.

```
enum ServantRetentionPolicyValue
{ NON_RETAIN, RETAIN /* DEFAULT */ };

interface ServantRetentionPolicy : CORBA::Policy {
  readonly attribute ServantRetentionPolicyValue value;
}
```

- The `NON_RETAIN` policy value must be used in conjunction with the request processing policy of either
  - `USE_DEFAULT_SERVANT`, in which case the POA delegates incoming requests to a default servant (used for DSI)
  - `USE_SERVANT_MANAGER`, in which case the POA uses the Interceptor pattern to determine how to associate a servant with the request

POA Policies for Lazy Object Allocation

The following example illustrates how to create references without first activating objects:

```
PortableServer::ObjectId_var oid =
  PortableServer::string_to_ObjectId ("my quoter");
CORBA::Object_var obj =
  child_poa->create_reference_with_id
    (oid.in (),
     "IDL:Stock/Quoter:1.0"); // Repository ID.

// Insert into a name context.
name_context->bind (svc_name, obj.in ());

// Later the following steps happen:
// 1. A new My_Quoter servant is created
// 2. This object is activated in the child_poa
```

Overview of Repository IDs

- An IDL compiler generates a unique repository ID for each identifier in an IDL file

```
module Stock {        // IDL:Stock:1.0
  interface Quoter { // IDL:Stock/Quoter:1.0
    long get_quote (in string stock_name);
    // IDL:Stock/Quoter/get_quote:1.0
  };
```

- You can use `#pragma prefix` to ensure the uniqueness of repository IDs

```
#pragma prefix "wallstreet.com"
module Stock {        // IDL:wallstreet.com/Stock:1.0
  interface Quoter { // IDL:wallstreet.com/Stock/Quoter:1.0
    long get_quote (in string stock_name);
    // IDL:wallstreet.com/Stock/Quoter/get_quote:1.0
  };
```

- You can use `#pragma version` to change the version number

Overview of Servant Managers

- The POA defines *servant managers* to support the lazy object allocation approach described above
- A servant manager is an interceptor that incarnates and etherealizes servants on-demand
- Two types of servant managers are supported
  - `ServantActivator`, which allocates a servant the first time it's accessed
  - `ServantLocator`, which allocates and deallocates a servant on each request
- Naturally, each type of servant manager can be selected via POA policies

The Request Processing Policy

- This policy controls whether a POA uses an AOM, a default servant, or “faults in” servants on-demand

```
enum RequestProcessingPolicyValue {
  USE_ACTIVE_OBJECT_MAP_ONLY /* DEFAULT */,
  USE_DEFAULT_SERVANT,
  USE_SERVANT_MANAGER
};

interface RequestProcessingPolicy : CORBA::Policy {
  readonly attribute RequestProcessingPolicyValue value;
}
```

- The `USE_ACTIVE_OBJECT_MAP_ONLY` policy value must be used in conjunction with the `RETAIN` servant retention policy
- The `USE_DEFAULT_SERVANT` policy value must be used in conjunction with the `MULTIPLE_ID` ID uniqueness policy

Servant Activator Definition

A POA created with RETAIN servant retention policy and the USE_SERVANT_MANAGER request processing policy uses the *servant activator* to "fault in" servants into the POA

```
typedef ServantBase *Servant;

// Skeleton class
namespace POA_PortableServer
{
  class ServantActivator :
    public virtual ServantManager
  {
    // Destructor.
    virtual ~ServantActivator (void);

    // Create a new servant for <id>.
    virtual Servant incarnate
      (const ObjectId &id,
       POA_ptr poa) = 0;

    // <servant> is no longer active in <poa>.
    virtual void etherealize
      (const ObjectId &,
       POA_ptr poa,
       Servant servant,
       Boolean remaining_activations) = 0;
  };
}
```

Custom ServantActivator Definition and Creation

```
// Implementation class.
class My_Quoter_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.in (), "my quoter") == 0)
      return new My_Quoter;
    else
      throw CORBA::OBJECT_NOT_EXIST ();
  }

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

Overview of the `String_var` Class

`String_var` is a "smart pointer" class

```
class String_var {
public:
  // Initialization and termination methods.
  String_var (char *); // Assumes ownership.
  String_var (const char *); // CORBA::string_dup().
  // ... (assignment operators are similar)
  ~String_var (); // Deletes the string.

  // Indexing operators.
  char &operator[] (CORBA::ULong index);
  char operator[] (CORBA::ULong index) const;

  // Workarounds for broken C++ compilers.
  const char *in () const;
  char *&inout ();
  char *&out ();

  // Relinquishes ownership.
  char *_retn ();
};

istream &operator >> (istream, CORBA::String_var &);
ostream &operator << (ostream,
                      const CORBA::String_var);
```

Servant Locator Definition

A POA created with NON_RETAIN servant retention policy and the USE_SERVANT_MANAGER request processing policy uses the *servant locator* to create/destroy a servant for each request

```
namespace POA_PortableServer
{
  class ServantLocator :
    public virtual ServantManager {
    // Destructor.
    virtual ~ServantLocator (void);

    // Create a new servant for <id>.
    virtual PortableServer::Servant preinvoke
      (const PortableServer::ObjectId &id,
       PortableServer::POA_ptr poa,
       const char *operation,
       PortableServer::Cookie &cookie) = 0;

    // <servant> is no longer active in <poa>.
    virtual void postinvoke
      (const PortableServer::ObjectId &id,
       PortableServer::POA_ptr poa,
       const char *operation,
       PortableServer::Cookie cookie,
       PortableServer::Servant servant) = 0;
  };
}
```

Custom ServantLocator Definition and Creation

```
// Implementation class.
class My_Quoter_Servant_Locator :
  public POA_PortableServer::ServantLocator {
  Servant preinvoke
    (const PortableServer::ObjectId &oid,
     PortableServer::POA_ptr poa,
     const char *operation,
     PortableServer::Cookie &cookie) {
    CORBA::String_var key =
      PortableServer::ObjectId_to_string (oid);
    Object_State state;
    if (database_lookup (key, state) == -1)
      throw CORBA::OBJECT_NOT_EXIST ();
    return new My_Quoter (&state);
  }

  void postinvoke
    (const PortableServer::ObjectId &id,
     PortableServer::POA_ptr poa,
     const char *operation,
     PortableServer::Cookie cookie,
     PortableServer::Servant servant) {
    database_update (servant);
    servant->_remove_ref ();
  }
};
```

Registering Servant Locators

```
PortableServer::ObjectId_var oid =
  PortableServer::string_to_ObjectId ("my quoter");
CORBA::Object_var obj =
  poa->create_reference_with_id (oid.in (),
                                 "IDL:Quoter:1.0");
// Insert into a name context.
name_context->bind (svc_name, obj.in ());

My_Quoter_Servant_Locator *quoter_locator =
  new My_Quoter_Servant_Locator;

// Locality constrained.
ServantLocator_var locator = quoter_locator->_this ();
poa->set_servant_manager (locator.in ());
PortableServer::POA_Manager_var poa_manager =
  poa->the_POAManager ();
poa_manager ()->activate ();
orb->run ();
```

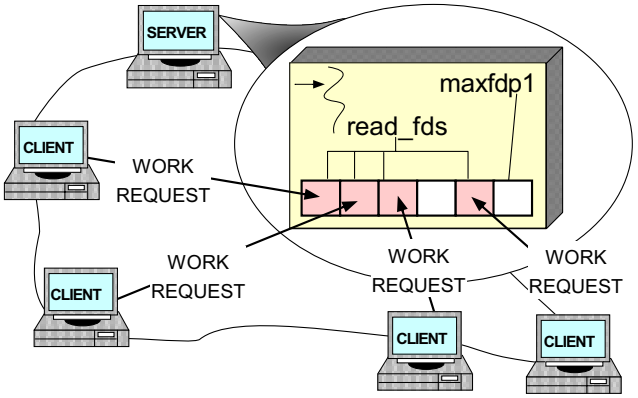
Overview of Adapter Activators

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

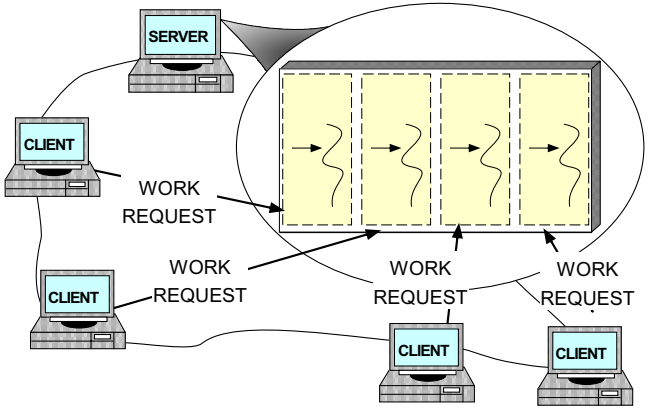
Additional Information on the POA

- See OMG POA specification for some examples:
  - One Servant for all Objects
  - Single Servant, many objects and types, using DSI
- See Vinoski/Henning book for even more examples
- See Schmidt/Vinoski C++ Report columns
  - www.cs.wustl.edu/~schmidt/report-doc.html
- See TAO release to experiment with working POA examples
  - `$TAO_ROOT/tests/POA/`
  - `$TAO_ROOT/examples/POA/`

Motivation for Concurrency in CORBA



(1) ITERATIVE SERVER



(2) CONCURRENT SERVER

- *Leverage hardware/software*
  - *e.g.*, multi-processors and OS thread support
- *Increase performance*
  - *e.g.*, overlap computation and communication
- *Improve response-time*
  - *e.g.*, GUIs and network servers
- *Simplify program structure*
  - *e.g.*, sync vs. async

Overview of the Thread Policy

- This policy controls whether requests are dispatched serially (*i.e.*, single-threaded) or whether they are dispatched using an ORB-defined threading model

```
enum ThreadPolicyValue
{ SINGLE_THREAD_MODEL, ORB_CTRL_MODEL /* DEFAULT */ };

interface ThreadPolicy : CORBA::Policy {
  readonly attribute ThreadPolicyValue value;
}
```

- The `SINGLE_THREAD_MODEL` policy value serializes all requests within a particular POA (but not between POAs, so beware of "servant sharing"...)
- The `ORB_CTRL_MODEL` can be used to allow the ORB to select the type of threading model and synchronization for a particular POA (which is not very portable, of course...)

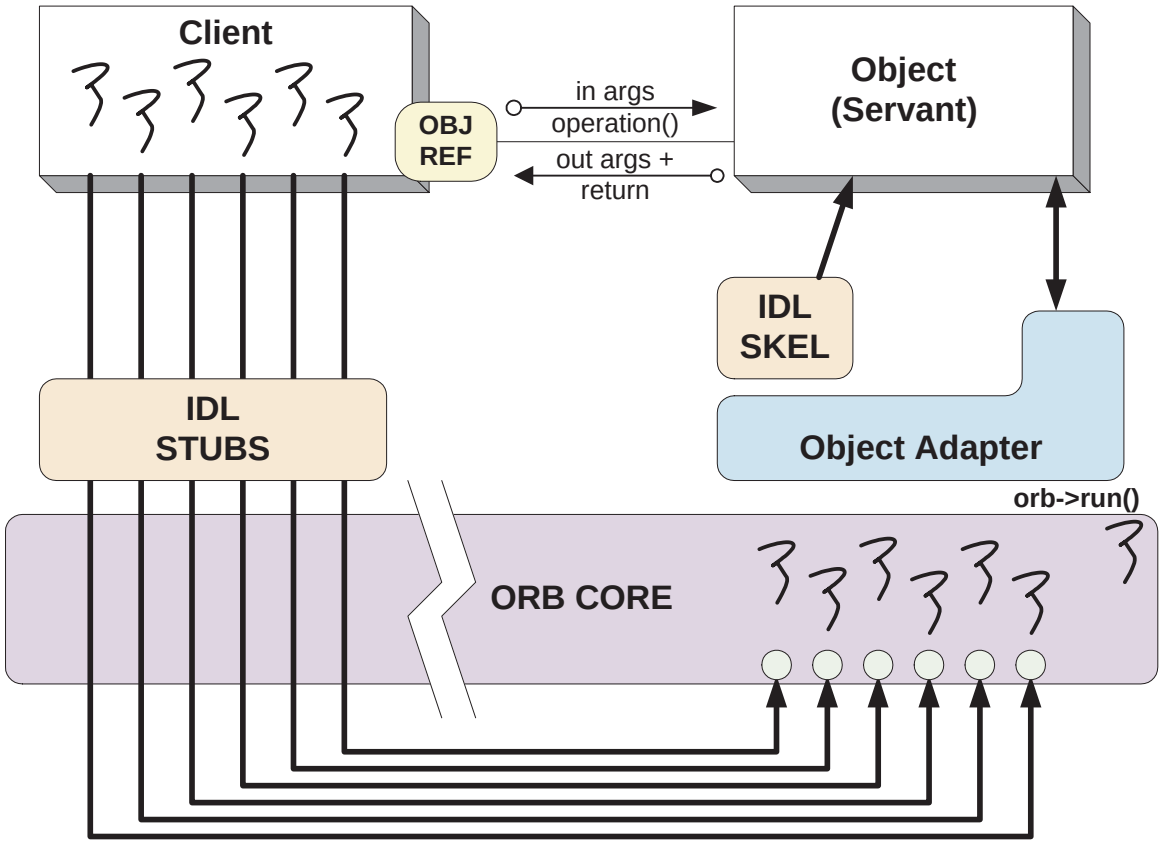
Threading in TAO

- An application can choose to ignore threads and if it creates none, it need not be thread-safe
- TAO can be configured with various concurrency strategies:
  - *Reactive* (default)
  - *Thread-per-Connection*
  - *Thread Pool*
  - *Thread-per-Endpoint*
- TAO also provides many locking strategies
  - TAO doesn't automatically synchronize access to application objects
  - Therefore, applications must synchronize access to their own objects

TAO Multi-threading Examples

- Each example implements a concurrent CORBA stock quote service
  - Show how threads can be used on the server
- The server is implemented in two different ways:
  1. *Thread-per-Connection* $\rightarrow$ Every client connection causes a new thread to be spawned to process it
  2. *Thread Pool* $\rightarrow$ A fixed number of threads are generated in the server at start-up to service all incoming requests
- Note that clients are unaware which concurrency model is being used...

TAO's Thread-per-Connection Concurrency Architecture



Pros

- Simple to implement and efficient for long-duration requests

Cons

- Excessive overhead for short-duration requests
- Permits unbounded number of concurrent requests

Thread-per-Connection Main Program

Server creates a `Quoter_Factory` and waits in ORB's event loop

```
int main (void) {
  ORB_Manager orb_manager (argc, argv);

  const char *factory_name = "my quoter factory";

  // Create servant (registers with rootPOA and Naming Service).
  My_Quoter_Factory *factory =
    new My_Quoter_Factory (factory_name);
  // Transfer ownership to smart pointer.
  PortableServer::ServantBase_var xfer (factory);

  // Block indefinitely dispatching upcalls.
  orb_manager.run ();
  // After run() returns, the ORB has shutdown.
}
```

The ORB's `svc.conf` file

```
static Advanced_Resource_Factory "-ORBReactorType select_mt"
static Server_Strategy_Factory "-ORBConcurrency thread-per-connection"
```

Thread-per-Connection Quoter Interface

Implementation of the `Quoter` IDL interface

```
typedef u_long COUNTER; // Maintain request count.

class My_Quoter : virtual public POA_Stock::Quoter
{
public:
  My_Quoter (void *state); // Constructor.

  // Returns the current stock value.
  long get_quote (const char *stock_name)
    throw (CORBA::SystemException, Quoter::InvalidStock);

  void remove (void)
    throw (CORBA::SystemException,
           CosLifeCycle::LifeCycleObject::NotRemovable);
private:
  ACE_Thread_Mutex lock_; // Serialize access to database.
  static COUNTER req_count_; // Maintain request count.
  CORBA::String_var last_quote_; // The last symbol looked up.
};
```

Thread-per-Connection Quoter Implementation

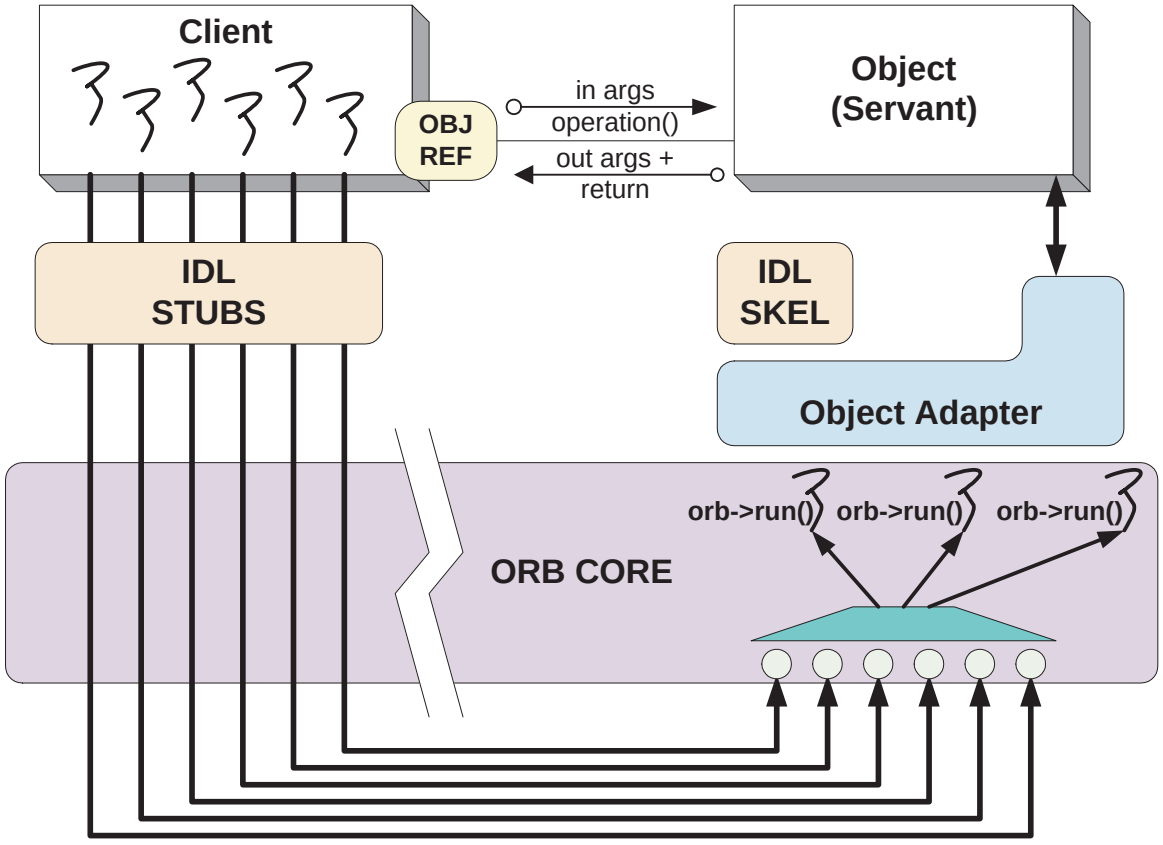
Implementation of multi-threaded Quoter callback invoked by the CORBA skeleton

```
long My_Quoter::get_quote (const char *stock_name) {
  ACE_GUARD_RETURN (ACE_Thread_Mutex, g, lock_, -1);

  ++My_Quoter::req_count_; // Increment the request count.

  // Obtain stock price (beware...).
  long value =
    Quote_Database::instance ()->lookup_stock_price (stock_name);

  if (value == -1)
    throw Stock::Invalid_Stock (); // Skeleton handles exceptions.

  last_quote_ = stock_name;
  return value;
}
```

Thread Pool

- This approach creates a thread pool to amortize the cost of dynamically creating threads
- In this scheme, before waiting for input the server code creates the following:
  1. A `Quoter_Factory` (as before)
  2. A pool of threads based upon the command line input
- Note the use of the `ACE_Thread_Manager::spawn_n()` method to spawn multiple pool threads

TAO's Thread Pool Concurrency Architecture



Pros

- Bounds the number of concurrent requests
- Scales nicely for multi-processor platforms, *e.g.*, permits load balancing

Cons

- May Deadlock

Thread Pool Main Program

```
int main (int argc, char *argv[])  {
  try {
    ORB_Manager orb_manager (argc, argv);

    const char *factory_name = "my quoter factory";

    // Create the servant, which registers with
    // the rootPOA and Naming Service implicitly.
    My_Quoter_Factory *factory =
      new My_Quoter_Factory (factory_name);
    // Transfer ownership to smart pointer.
    PortableServer::ServantBase_var xfer (factory);

    int pool_size = // ...

    // Create a thread pool.
    ACE_Thread_Manager::instance ()->spawn_n
      (pool_size,
       &run_orb,
       (void *) orb_manager.orb ());
    // Block indefinitely waiting for other
    // threads to exit.
    ACE_Thread_Manager::instance ()->wait ();

    // After run() returns, the ORB has shutdown.
    } catch (...) { /* handle exception ... */ }
}
```

Thread Pool Configuration

The `run_orb()` adapter function

```
void run_orb (void *arg)
{
  try {
    CORBA::ORB_ptr orb =
      ACE_static_cast (CORBA::ORB_ptr, arg);

    // Block indefinitely waiting for incoming
    // invocations and dispatch upcalls.
    orb->run ();

    // After run() returns, the ORB has shutdown.
  } catch (...) { /* handle exception ... */ }
}
```

The ORB's `svc.conf` file

```
static Resource_Factory "-ORBReactorType tp"
```

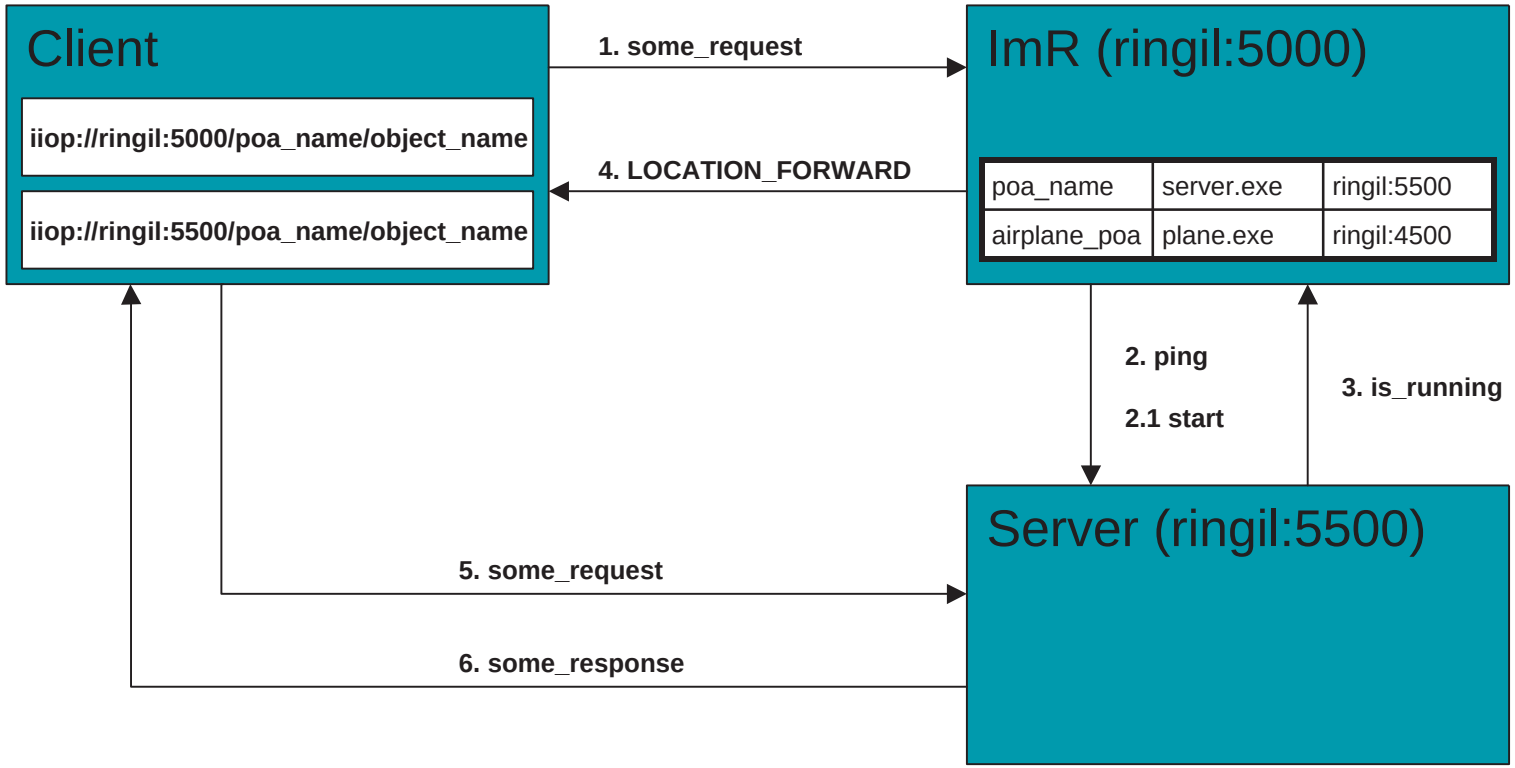
Additional Information on CORBA Threading

- See Real-time CORBA 1.0 specification
  - Now adopted as part of CORBA specifications
- See our papers on CORBA Threading
  - www.cs.wustl.edu/∼schmidt/PDF/CACM-arch.pdf
  - www.cs.wustl.edu/∼schmidt/PDF/RTAS-02.pdf
  - www.cs.wustl.edu/∼schmidt/PDF/RT-perf.pdf
  - www.cs.wustl.edu/∼schmidt/PDF/COOTS-99.pdf
  - www.cs.wustl.edu/∼schmidt/PDF/orc.pdf
  - www.cs.wustl.edu/∼schmidt/report-doc.html
- See TAO release to experiment with working threading examples
  - `$TAO_ROOT/tests/`

Implementation Repository

- Allows the ORB to activate servers to process operation invocations
- Store management information associated with objects
  - *e.g.*, resource allocation, security, administrative control, server activation modes, etc.
- Primarily designed to work with *persistent* object references
- From client's perspective, behavior is portable, but administrative details are highly specific to an ORB/OS environment
  - *i.e.*, not generally portable
- www.cs.wustl.edu/∼schmidt/PDF/binding.pdf

Typical Implementation Repository Use-case



Server Activation via Implementation Repository

- If the server isn't running when a client invokes an operation on an object it manages, the Implementation Repository automatically starts the server
- Servers can register with the Implementation Repository
  - *e.g.*, in TAO

```
% tao_imr add airplane_poa -c "plane.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

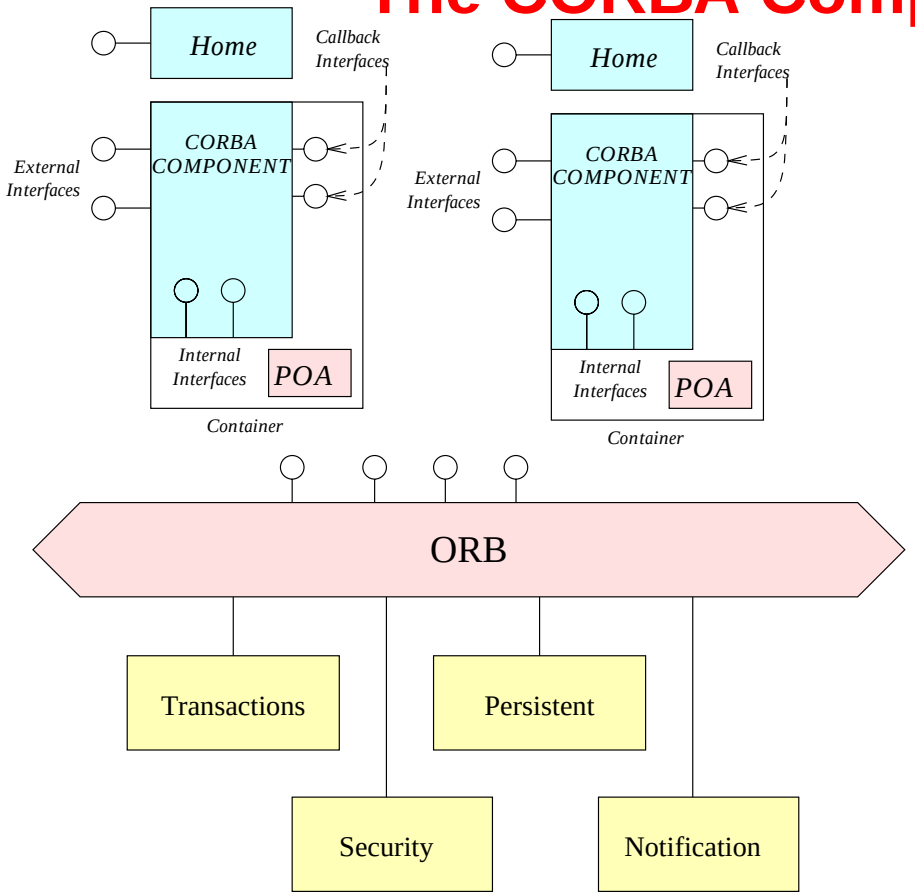
Server Activation Modes

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

- TAO's Implementation Repository supports four types of activation

  1. *Normal* $\rightarrow$ one server, started if needed but not running
  2. *Manual* $\rightarrow$ one server, will not be started on client request, *i.e.*, pre-launched
  3. *Per-client call* $\rightarrow$ one server activated for each request to the Implementation Repository
  4. *Automatic* $\rightarrow$ like *normal*, except will also be launched when the Implementation Repository starts

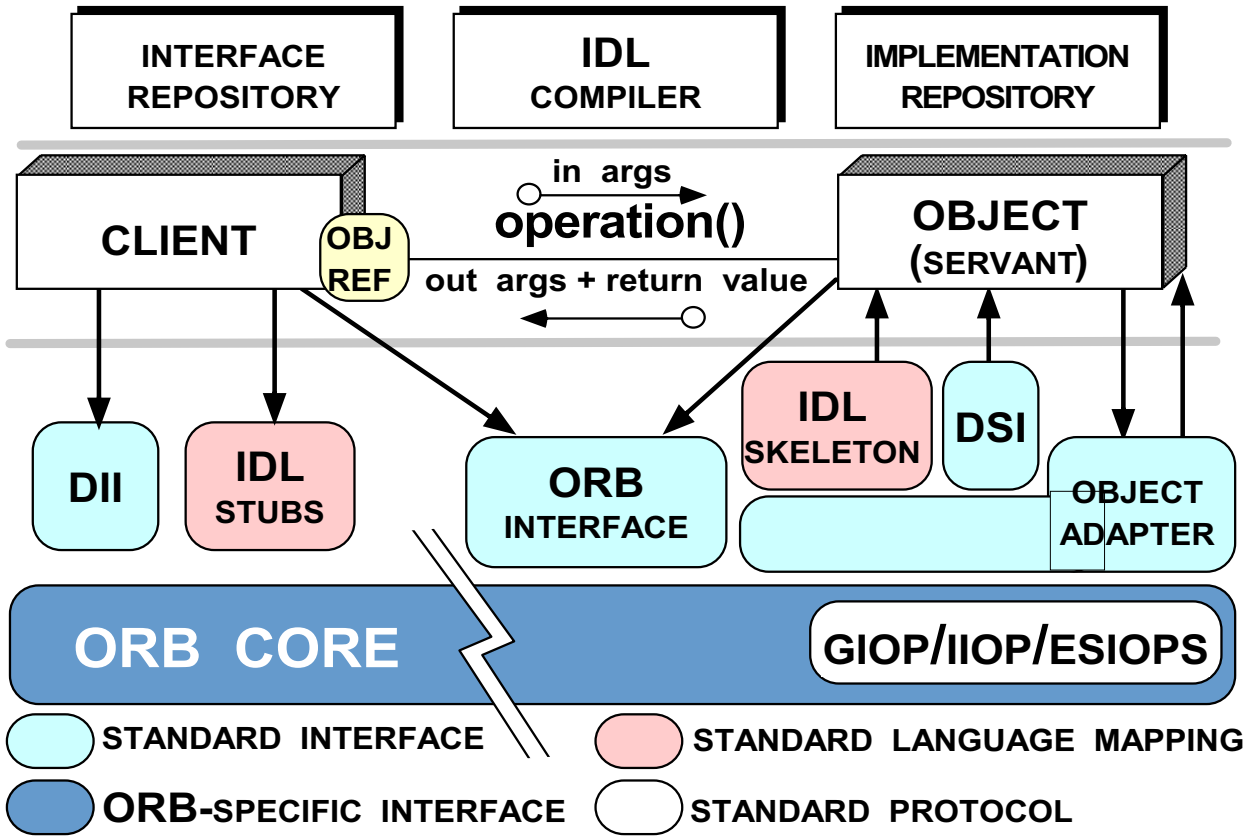
The CORBA Component Model



~schmidt/PDF/CBSE.pdf

- **Features**
  - Navigation among interfaces supported by components
  - Standardized system-component interaction
  - Standardized component life-cycle management
  - Component interconnections
  - Standardized component configuration
  - Standardized ORB services interfaces

Evaluating CORBA



www.cs.wustl.edu/~schmidt/corba.html

Criteria

- 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 supplier
- CORBA 2.x defines a useful interoperability specification
  - Later extensions deal with portability issues for server
    - *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, the latest CORBA specification standardizes
  - IDL-to-C++ language mapping
  - Naming service, event service, lifecycle service
  - ORB initialization service
  - Portable Object Adapter API
  - Servant mapping
  - Server thread pools (Real-time CORBA)
- Porting applications from ORB-to-ORB is greatly simplified by `corbaconf`
  - http://corbaconf.kiev.ua

Feature Limitations (1/3)

- Standard CORBA doesn't yet address all the "inherent" complexities of distributed computing, *e.g.*,
  - *Latency*
  - *Causal ordering*
  - *Deadlock*
- It does address
  - *Service partitioning*
  - *Fault tolerance*
  - *Security*

Feature Limitations (2/3)

- All ORBs support the following semantics:
  - Object references are passed by-reference
    - However, all operations are routed to the originator
  - C-style structures and discriminated unions may be passed by-value
    - However, these structures and unions do *not* contain any methods
- Older ORBs didn't support passing objects-by-value (OBV)
  - However, CORBA 2.3 OBV spec. defines a solution for this and many ORBs now implement it
- If OBV is not available, objects can be passed by value using hand-crafted "factories" (tedious)

Feature Limitations (3/3)

- Many ORBs do not yet support AMI and/or standard CORBA timeouts
  - However, these capabilities are defined in the OMG Messaging and are implemented by ORBs like TAO and Orbix 2000 Specification
- Most ORBs do not yet support fault tolerance
  - This was standardized by the OMG recently, however
  - www.omg.org/techprocess/meetings/schedule/Fault_Tolerance_RFP.html
- Versioning is supported in IDL via `pragmas`
  - Unlike Sun RPC or DCE, which include in language

Performance Limitations

- Performance may not be as good as hand-crafted code for some applications due to
  - Additional remote invocations for naming
  - Marshaling/demarshaling overhead
  - Data copying and memory management
  - Endpoint and request demultiplexing
  - Context switching and synchronization overhead
- Typical trade-off between extensibility, robustness, maintainability $\rightarrow$ *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
  - Orbix2000 and ORBacus from IONA
  - VisiBroker from Borland
  - BEA Web Logic Enterprise
  - Component Broker from IBM
  - e*ORB from PrismTech and ORB Express from OIS
  - Open-source ORBs $\rightarrow$ TAO, JacORB, omniORB, and MICO
- 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, relationship, security, concurrency, property, A/V streaming, 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++, Java, C, COBOL, etc.
  - A Static Invocation Interface, used to compose operation requests via proxies
  - A Dynamic Invocation Interface, used to compose operation 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
  - Doug Schmidt's CORBA page
    - www.cs.wustl.edu/~schmidt/corba.html
  - OMG's WWW Page
    - www.omg.org/corba/
  - CETUS CORBA Page
    - www.cetus-links.org/oo_corba.html