Developing Distributed Object Computing Applications with CORBA

Douglas C. Schmidt

Associate Professor
schmidt@uci.edu
www.eng.uci.edu/~schmidt/
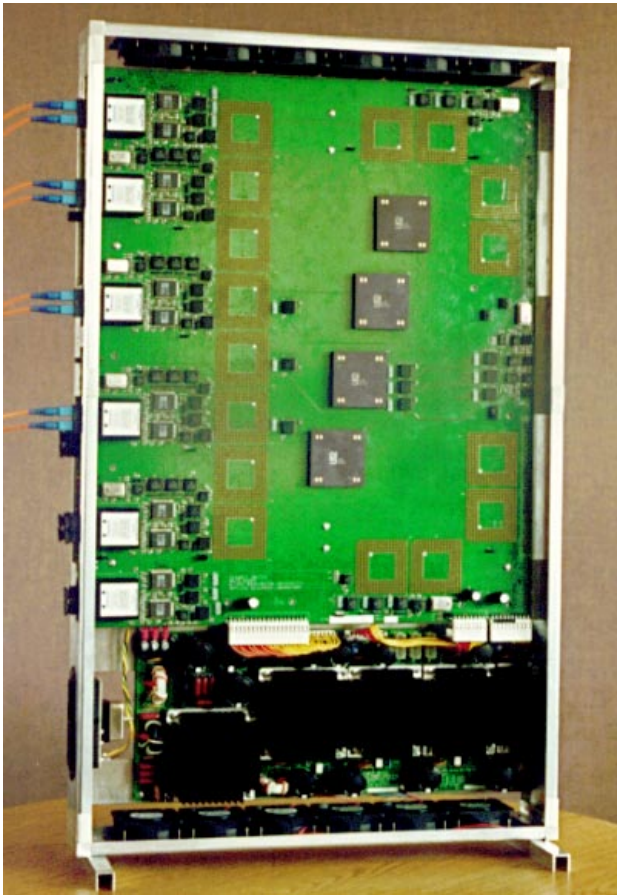
Elec. & Comp. Eng. Dept.
University of California, Irvine
(949) 824-1901

Motivation: the Distributed Software Crisis

Symptoms

- **Hardware** gets smaller, faster, cheaper

- **Software** gets larger, slower, more expensive
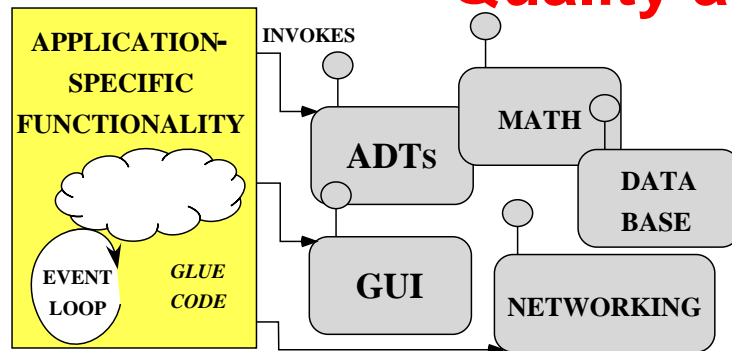
Culprits

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

Solution Approach

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

Techniques for Improving Software Quality and Productivity

(A) CLASS LIBRARY ARCHITECTURE
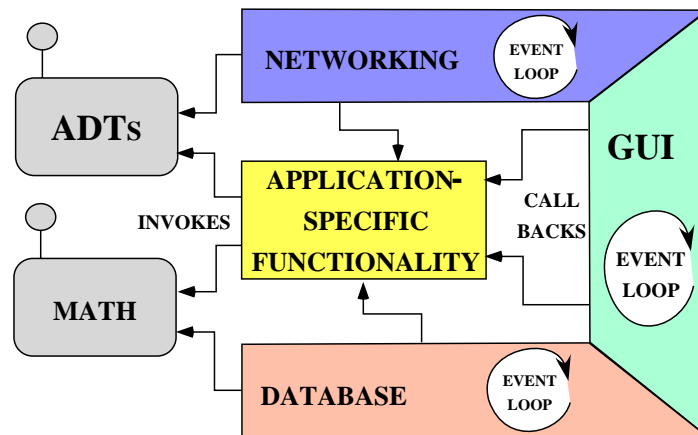
(B) FRAMEWORK ARCHITECTURE

Proven solutions $\rightarrow$

- *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
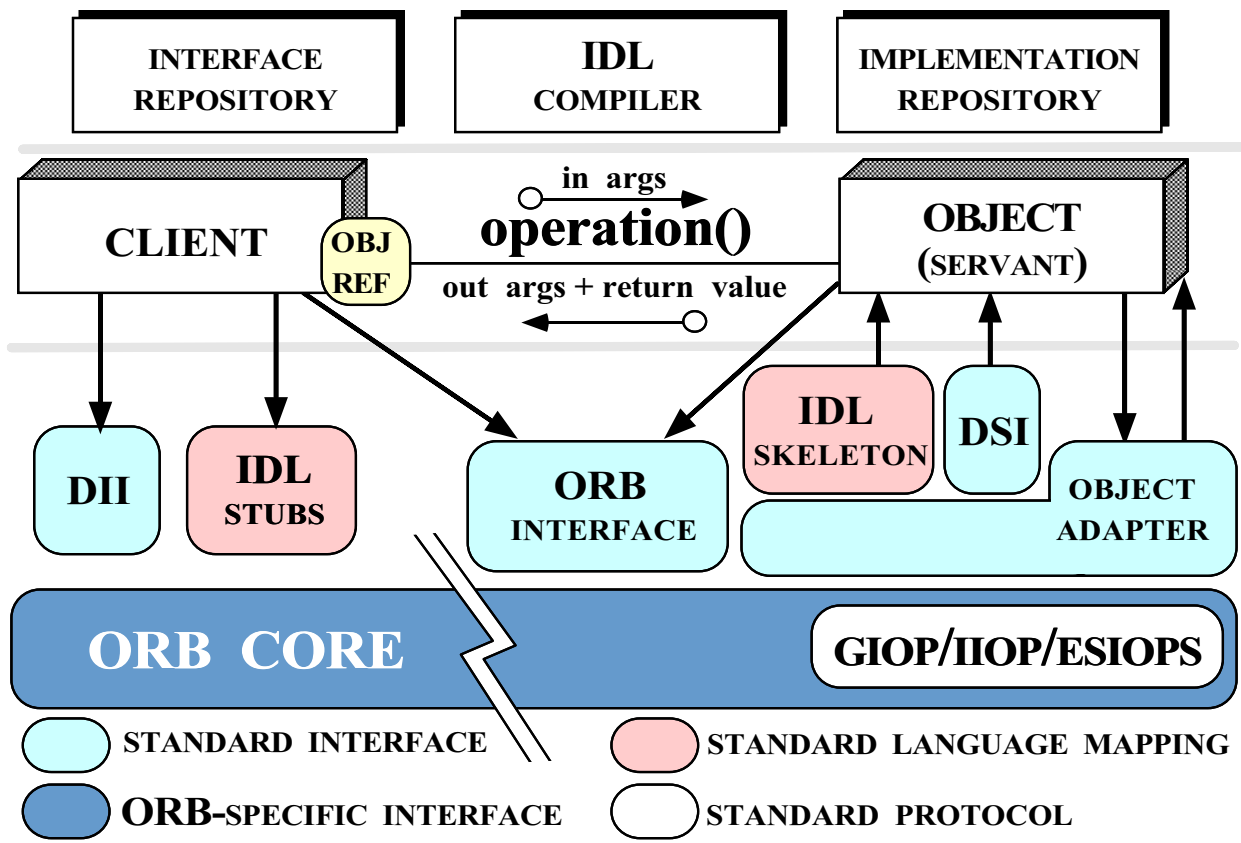
Goals of CORBA

- Simplify distribution by automating

  - Object location & activation
  - Parameter marshaling
  - Demultiplexing
  - Error handling

- Provide foundation for higher-level services

Applying CORBA to Medical Imaging
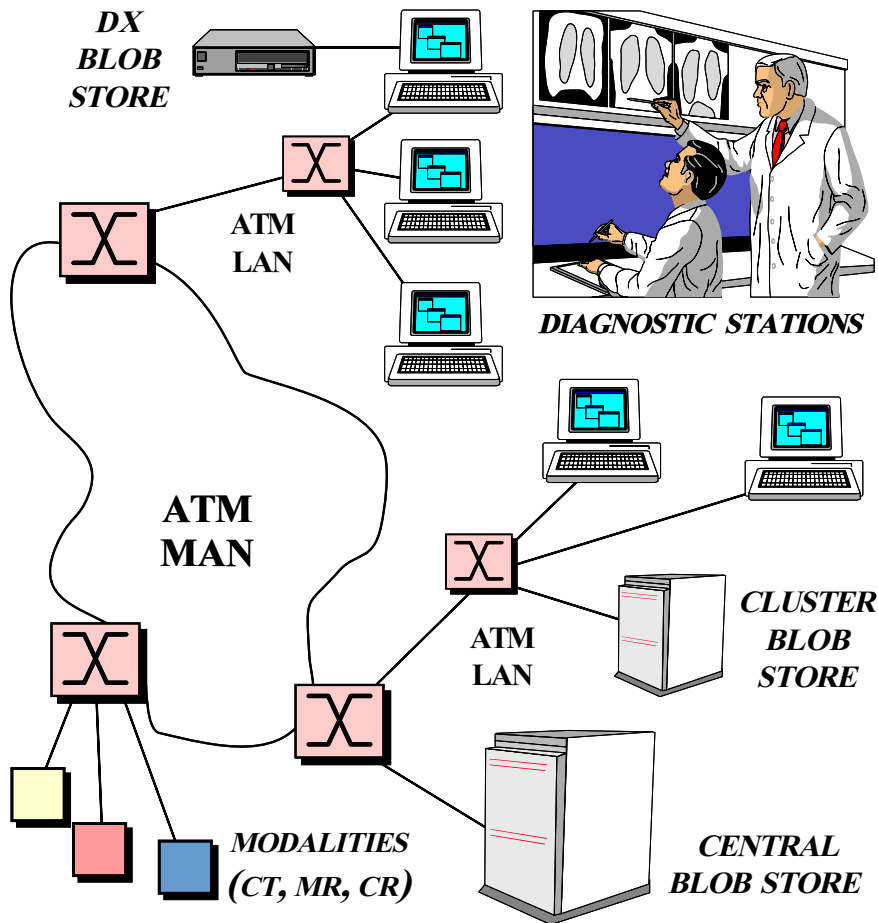
- **Domain Challenges**

  - Large volume of "Blob" data
    * *e.g.*, 10 to 40 Mbps
  - "Lossy compression" isn't viable
  - Prioritization of requests

- **URLs**

  - ~schmidt/COOTS-96.ps.gz
  -
    ~schmidt/av_chapter.ps.gz
  - ~schmidt/NMVC.html

Applying CORBA to Real-time Avionics
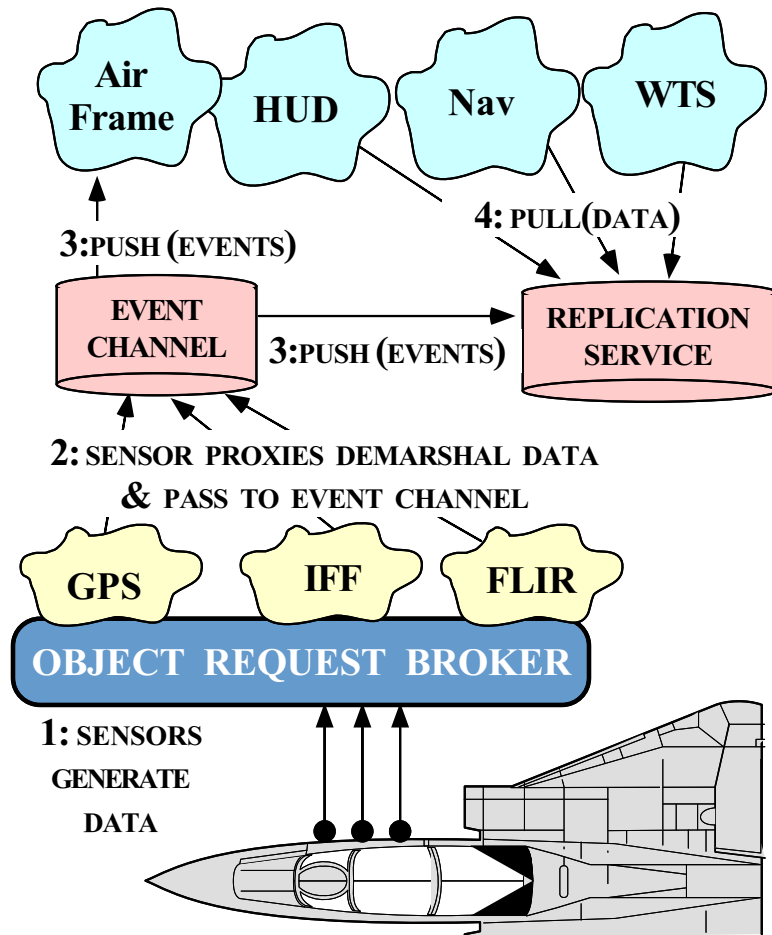
- **Domain Challenges**

  - Real-time periodic processing
  - Complex dependencies
  - Very low latency

- **URLs**

  - ~schmidt/JSAC-98.ps.gz
  - ~schmidt/TAO-boeing.html

Applying CORBA to Global PCS
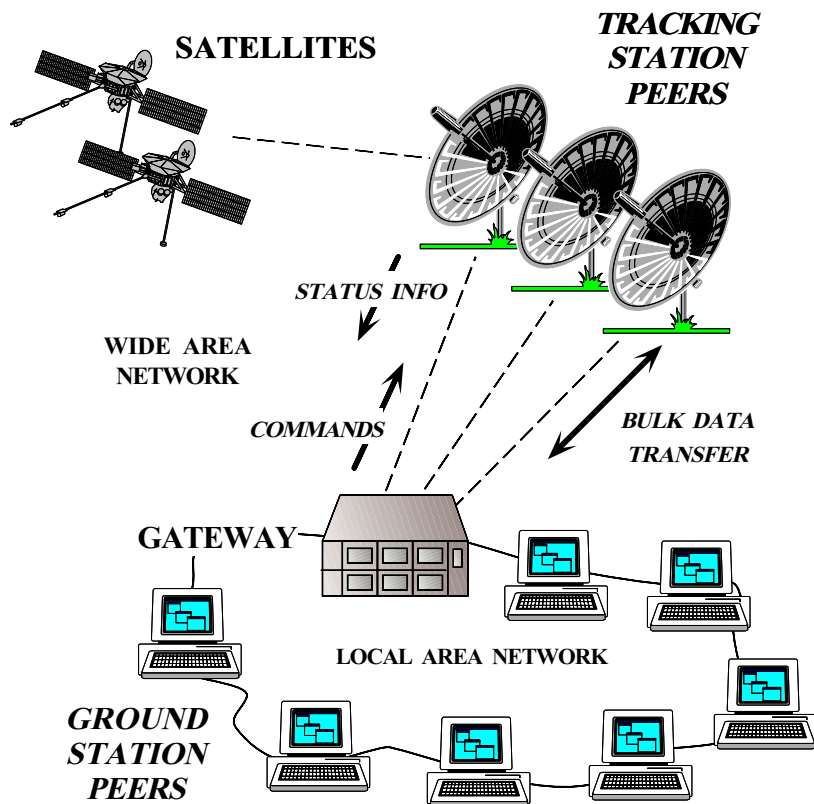
- **Domain Challenges**

  - Long latency satellite links
  - High reliability
  - Prioritization

- **URL**

  - ~schmidt/TAPOS-95.ps.gz

Tutorial Outline

- Motivation

- Example CORBA Applications

- Coping with Changing Requirements

- Overview of CORBA Architecture

- 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$1,000 computer companies

  - Sun, HP, DEC, IBM, IONA, Inprise, Cisco, Motorola, Boeing, etc.

- The latest version of the CORBA spec is now available

  - www.omg.org/technology/documents/formal/

Sources of Complexity for Distributed Applications

PRINTER

COMPUTER

CD ROM

FILE
SYSTEM

(1) STAND-ALONE APPLICATION ARCHITECTURE

NAME
SERVICE

TIME
SERVICE

CYCLE
SERVICE

DISPLAY
SERVICE

FILE
SERVICE

NETWORK

PRINT
SERVICE

CD ROM

PRINTER

FILE SYSTEM
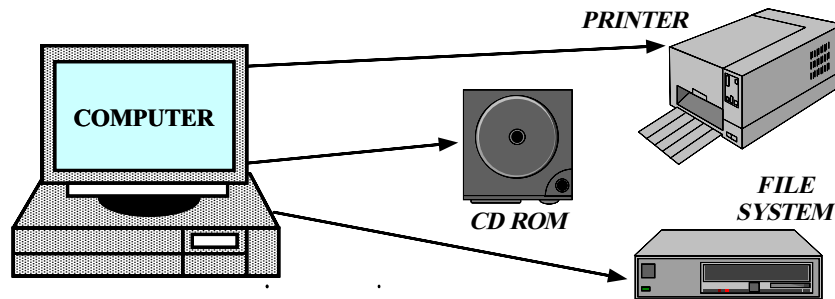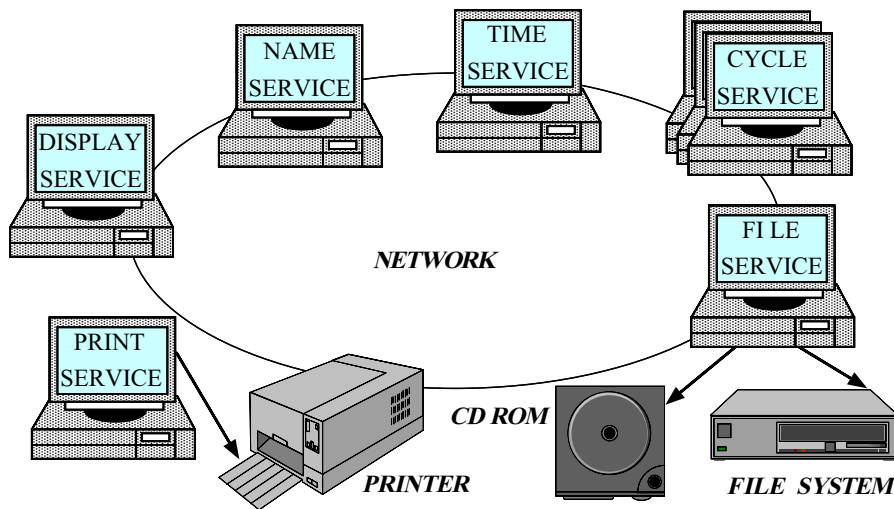
(2) DISTRIBUTED APPLICATION ARCHITECTURE

- **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 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

- Key limitations 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**

  - CORBA provides higher level integration than traditional *untyped TCP bytestreams*

- **Benefits for distributed programming similar to OO languages for non-distributed programming**

  - *e.g.*, encapsulation, interface inheritance, polymorphism, and exception handling

- **Provides a foundation for higher-level distributed object collaboration**

  - *e.g.*, ActiveX and the OMG Common Object Service Specification (COSS)

CORBA Quoter Example

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

OMG IDL Compiler

interface   Quoter
{
  long get_quote (in string name);
};

IDL FILE

IDL COMPILER

| CLIENT STUB BODY | CLIENT STUB HEADER | SERVER SKELETON HEADER | SERVER SKELETON BODY |

CLIENT SOURCE CODE

CLIENT SOURCE CODE

CLIENT SOURCE CODE

SERVER SOURCE CODE

SERVER SOURCE CODE

SERVER SOURCE CODE

JAVA COMPILER

C++ COMPILER

CLIENT PROGRAM

CORBA RUN-TIME LIBRARIES

SERVER PROGRAM
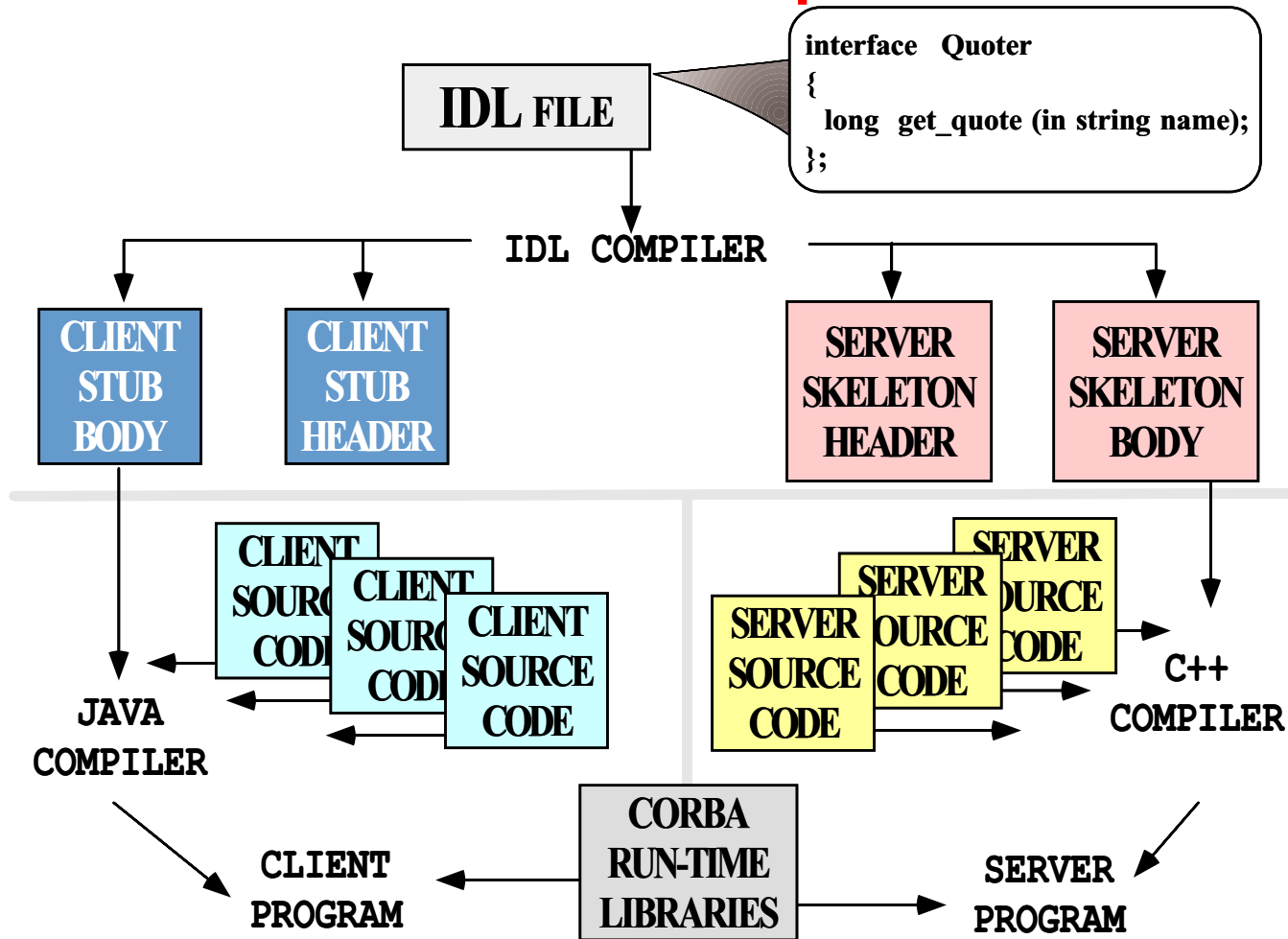
Software Bus
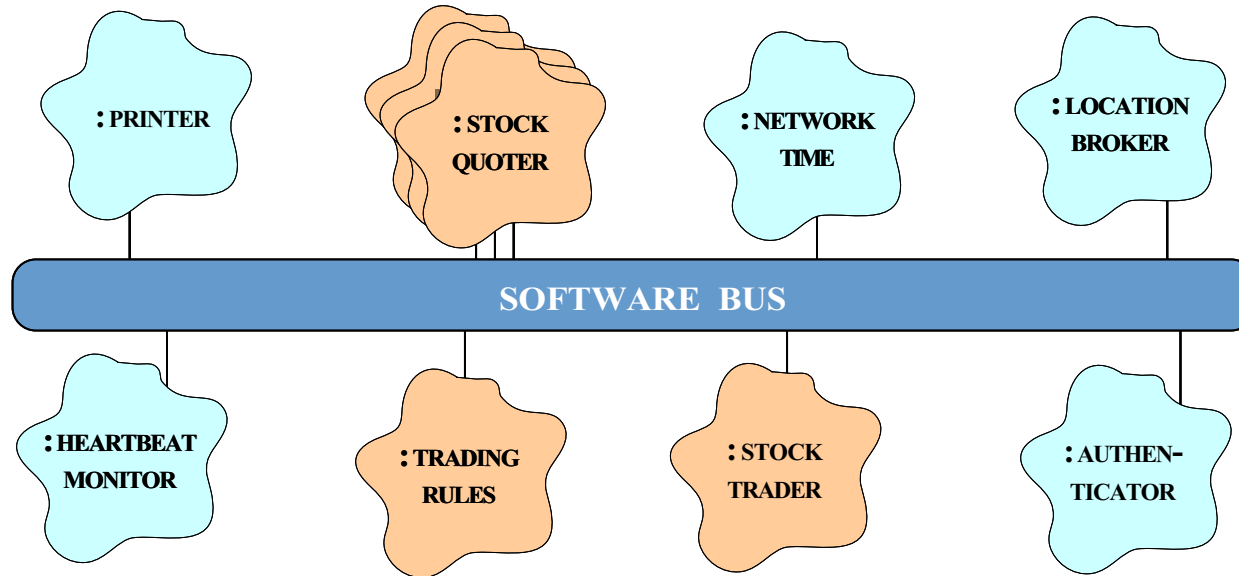
- CORBA provides a communication infrastructure for a heterogeneous, distributed collection of collaborating objects

- Analogous to "hardware bus"

CORBA Object Collaboration
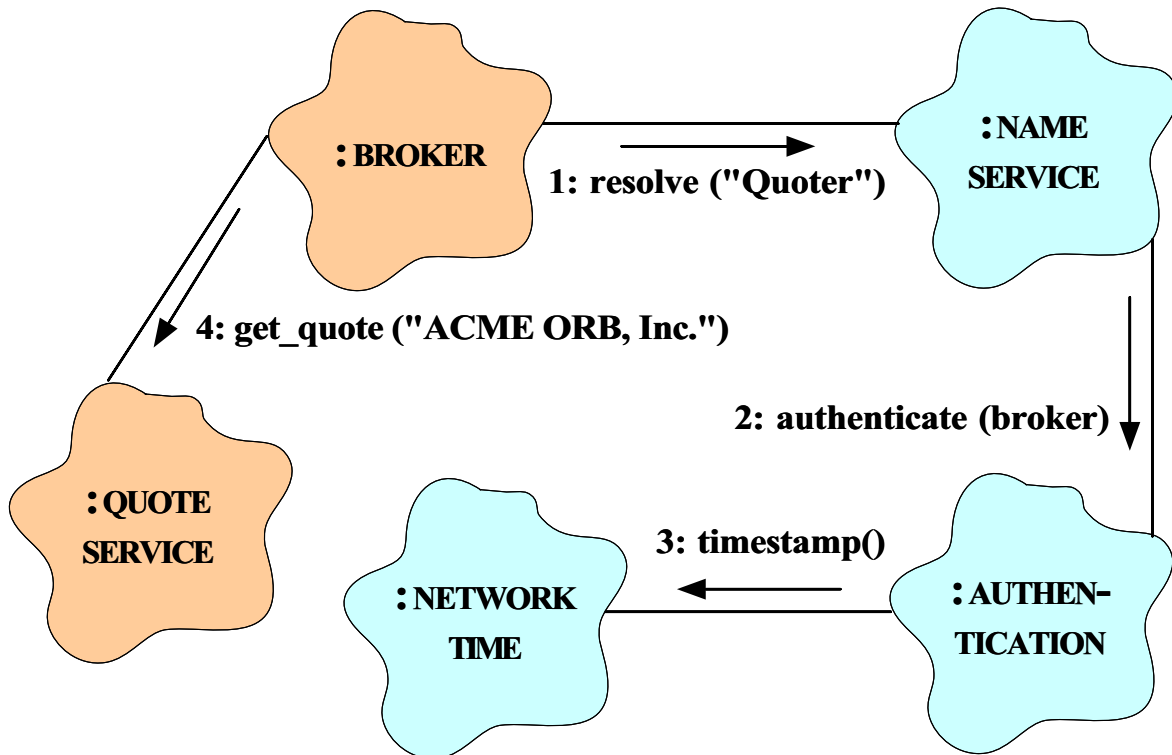
- Collaborating objects can be either remote or local

    - *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 Events & Notification Services define a publish & subscribe communication paradigm
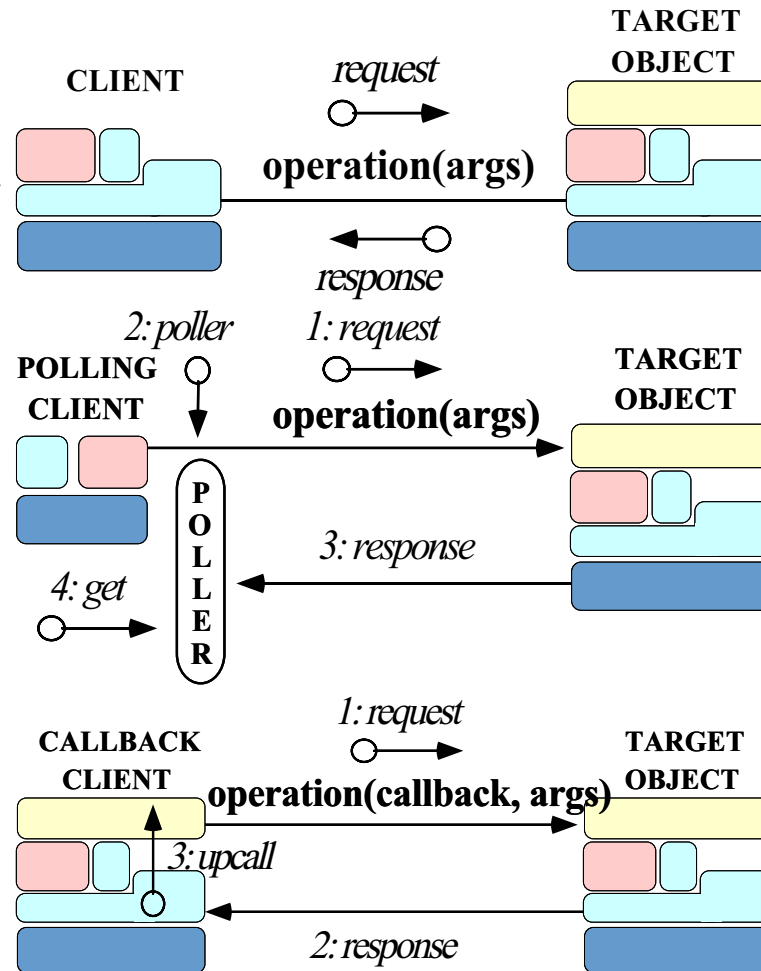
Fundamental CORBA Design Principles

- 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

- Traditional RPC (*e.g.*, DCE)

  - Only supports "procedural" integration of application services
  - Doesn't provide object abstractions, async message passing, or dynamic invocation
  - Doesn't address inheritance of interfaces

- Windows COM/DCOM/COM+

  - Traditionally limited to desktop applications
  - Does not address heterogeneous distributed computing

Related Work (cont'd)

- ## 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

CORBA Stock Quoter Application Example
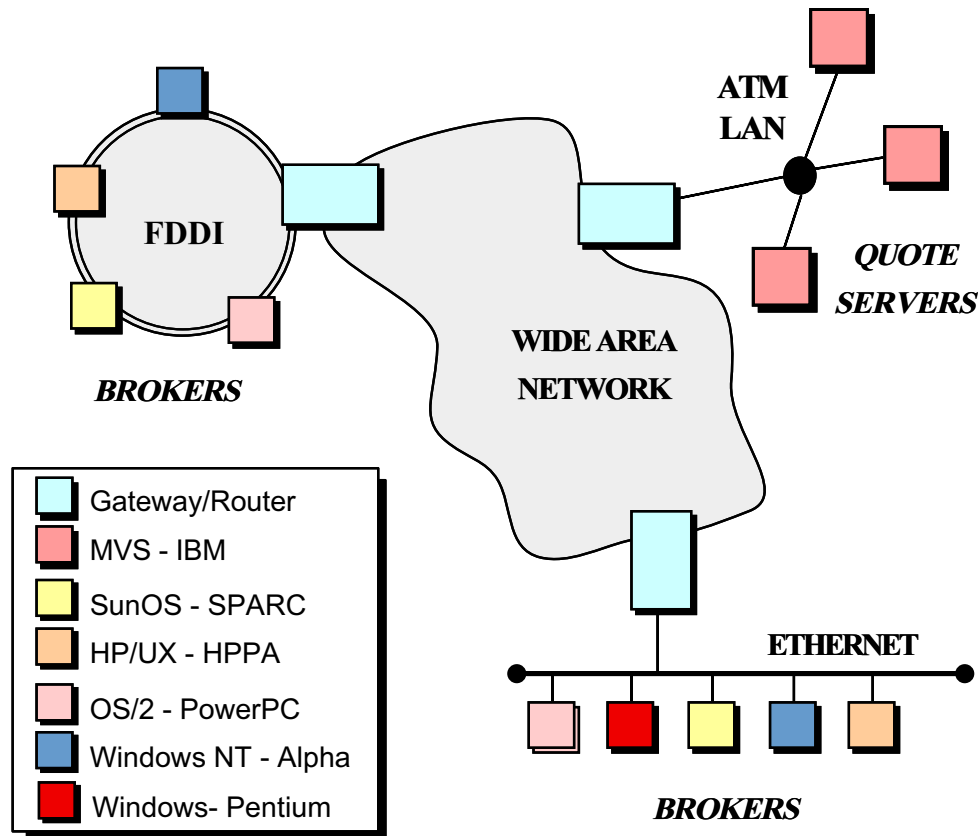
- The quote server(s) maintains the current stock prices

- Brokers access the quote server(s) via CORBA

- Note all the heterogeneity!

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);
  };
};
```

UC Irvine

Revised OMG IDL Quoter Definition

Apply the CORBA Lifecycle Service

```
module Stock {
  exception Invalid_Stock {}; // Similar to structs.

  // Interface is similar to a C++ class.
  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);
  };
};
```

RPC-style vs. Object-style Communication
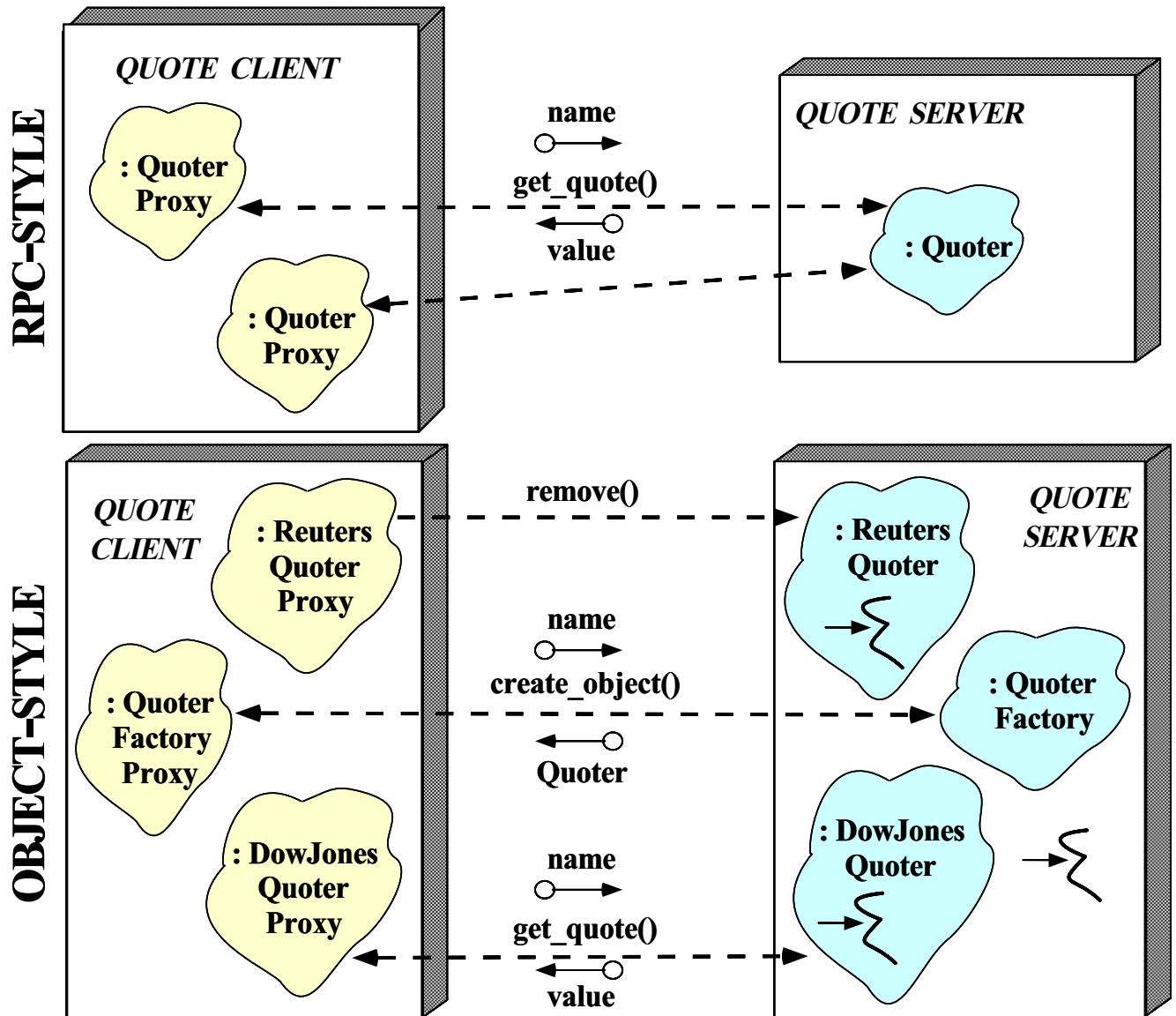
Compiling the Interface Definition

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

  - The stub is a *proxy* that marshals parameters on the client
  - The skeleton is an *adapter* that demarshals parameters on the server

- CORBA associates a servant to a generated IDL skeleton 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`

Automatically-Generated
Client-side Stubs

```cpp
namespace Stock
{
  class Quoter
    : public virtual CosLifeCycle::LifeCycleObject
      // Quoter also IS-A CORBA::Object.
  {
  public:
    // Proxy interface.
    CORBA::Long get_quote (const char *stock_name);
  };

  class Quoter_Factory
    : public virtual CosLifeCycle::GenericFactory
      // GenericFactory IS-A CORBA::Object.
  {
  public:
    // Proxy Factory method for creation.
    // Inherits:
    // CORBA::Object_ptr create_object
    //    (const CosLifeCycle::Key &factory_key,
    //     const CosLifeCycle::Criteria &criteria)
  };
  // ...
};
```

The Class Form of the Adapter Pattern

client

1: create_object()

Servant Base
A

CosLifeCycle GenericFactory
———————————
create_object() = 0
A

My Quoter Factory
———————————
create_object()

Quoter Factory
———————————
create_object() = 0
A
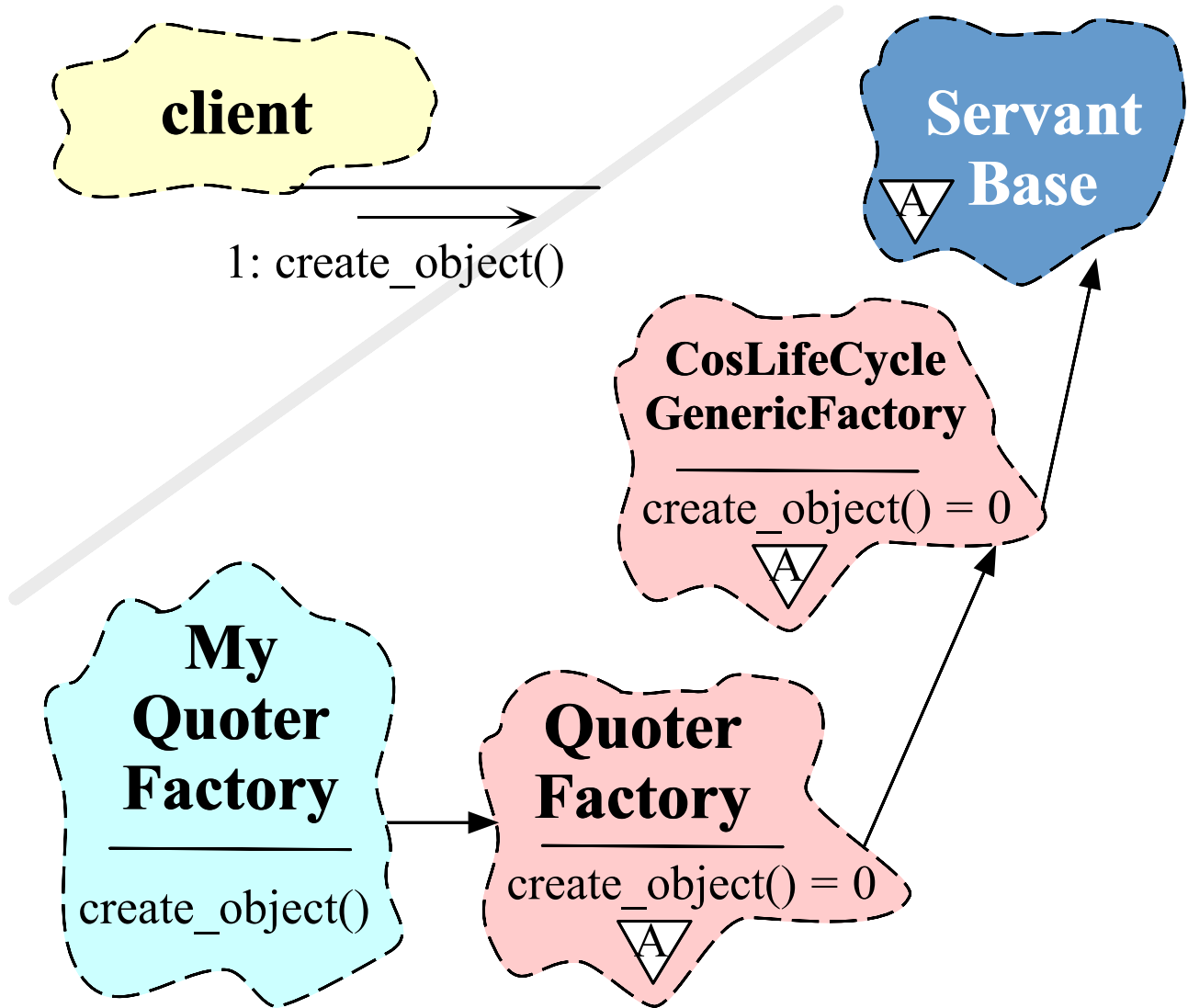
Defining a Servant Using Inheritance

```
class My_Quoter_Factory : public virtual POA_Stock::Quoter_Factory
{
public:
  My_Quoter_Factory (const char *factory_name =
                       "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);
};
```

The drawback is that implementations inherit from generated skeletons

- Can create a "brittle" hierarchy and make it hard to integrate with legacy code, *i.e.*, distributing a stand-alone application

- Virtual inheritance is often poorly implemented

The Object Form of the Adapter Pattern

client

1: create_quoter ()

Servant
Base
A

CosLifeCycle
GenericFactory
———————
create_object() = 0
A

Quoter
Factory
———————
create_object() = 0
A

My_Quoter
Factory

Quoter
Factory_tie
———————
create_object()

2: create_object()

My
Quoter
Factory
———————
create_object()
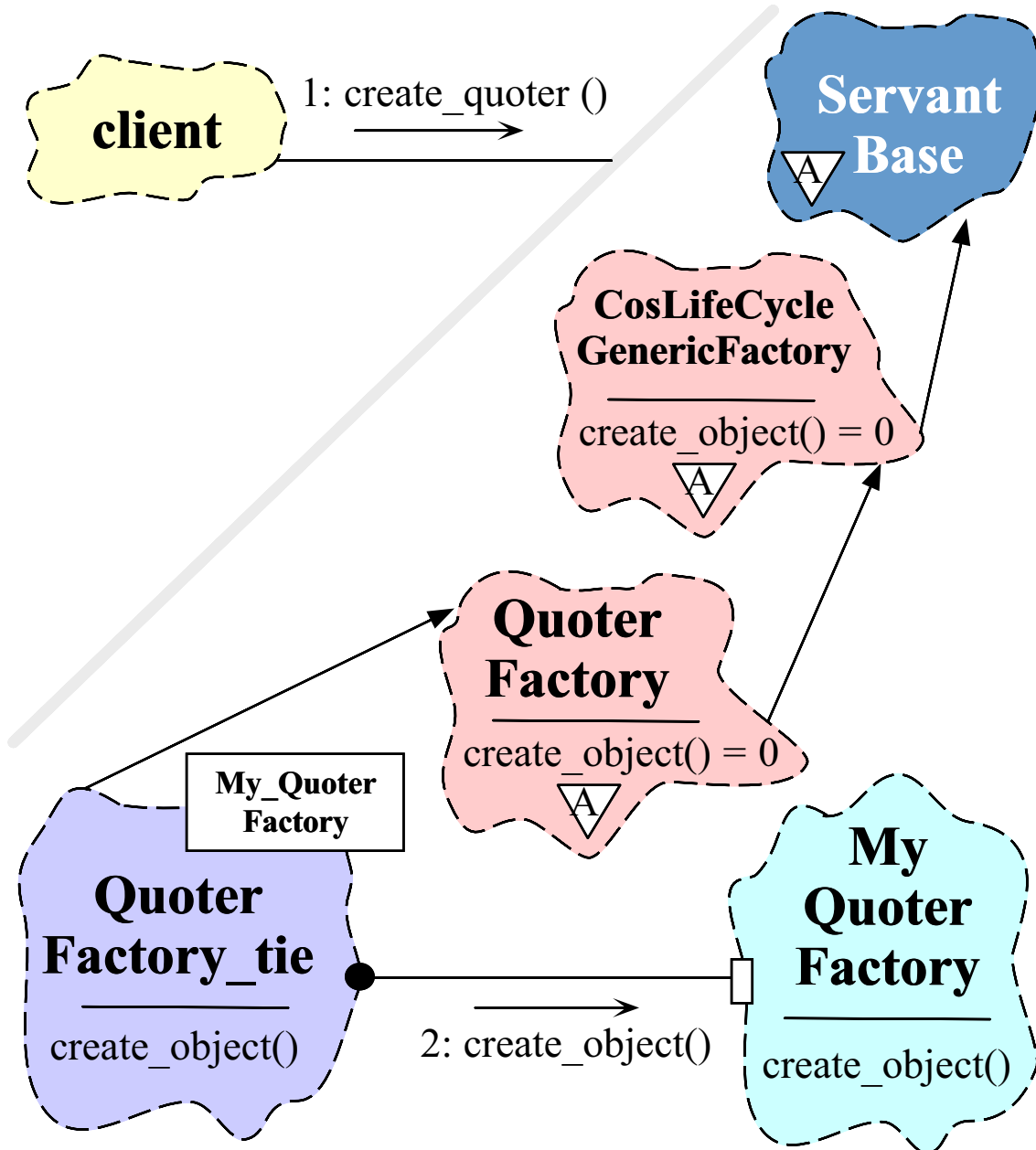
A TIE-based Implementation

```
class My_Quoter_Factory {
public:
  My_Quoter_Factory (const char *factory_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);
};
```

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 can be 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);
```

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

Implementing My_Quoter_Factory

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

  // Perform Factory Method selection of
  // the subclass of Quoter.
`  if (strcmp (factory_key.id,
              "Dow Jones") == 0)
    quoter = new Dow_Jones_Quoter;
  // ...
  else if (strcmp (factory_key.id,
                   "My Quoter") == 0)
    // Dynamically allocate a My_Quoter object.
    quoter = new My_Quoter;
  else
    // Raise 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 ();
};
```

Registering `My_Quoter_Factory` with the Naming Service

```cpp
extern CosNaming::NamingContext_ptr
        name_context;

My_Quoter_Factory::My_Quoter_Factory
  (const char *factory_name)
{
  CosNaming::Name name;
  name.length (1);
  name[0].id = factory_name;
  name[0].kind = "object impl";

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

  // Export our object reference to the
  // naming context.
  name_context->bind (name, qf);
};
```

Real code should handle exceptions...

Using the CORBA Naming Service
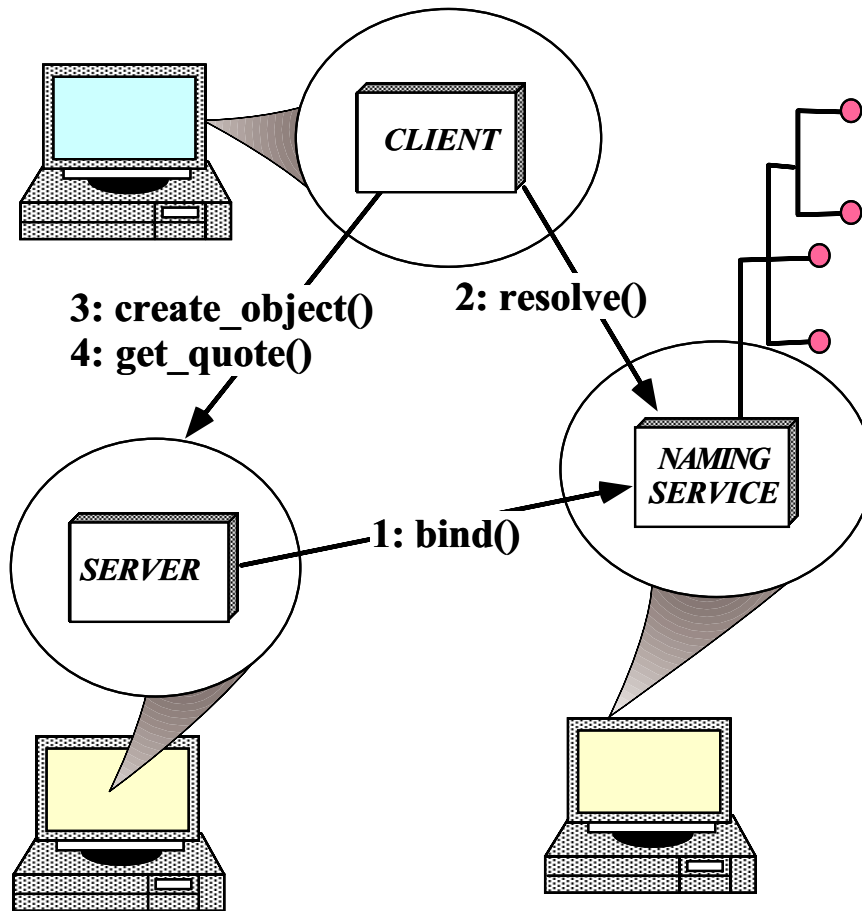
- **Purpose**

  – Maps sequences of strings to object references

- **Implementation**

  – `Naming Context` can be a hierarchically nested graph
  – Written using ACE `Hash Maps`

Diagram labels:
- CLIENT
- SERVER
- NAMING SERVICE
- 3: create_object()
- 4: get_quote()
- 2: resolve()
- 1: bind()

The Main Server Program

Uses *persistent activation* mode

```cpp
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 (factory_name);

  // Could use the TIE approach and explicitly
  // register the servant with the POA, i.e.:
  // MY_QUOTER_FACTORY factory
  //    (new My_Quoter_Factory (factory_name));
  // orb_manager.activate (&factory);

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

Server Initialization Wrapper Facades

```cpp
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);
    poa_manager_ = poa_->the_POAManager ();
  }

  // Register <servant> with the <poa_>.
  int activate (PortableServer::Servant servant) {
    return poa_->activate_object (servant);
  }
  // ORB Accessor.
  CORBA::ORB_ptr orb (void) { return orb_; }

  // Run the main ORB event loop.
  int run (void) {
    poa_manager_->activate ();
    return orb_->run ();
  }
private:
  CORBA::ORB_var orb_;
  PortableServer::POA_var poa_;
  PortableServer::POA_Manager_var poa_manager_;
```

POA IDL Mappings

CLIENT-SIDE MAPPING

SERVER-SIDE MAPPING

CORBA:: Object

Servant Base

CosLifeCycle:: GenericFactory

create_object() = 0

POA_CosLifeCycle:: GenericFactory

create_object() = 0

Stock:: Quoter Factory

create_object() = 0

POA_Stock:: Quoter_Factory

create_object() = 0

Stock:: Quoter Factory_var

create_object()

My Quoter Factory

create_object()
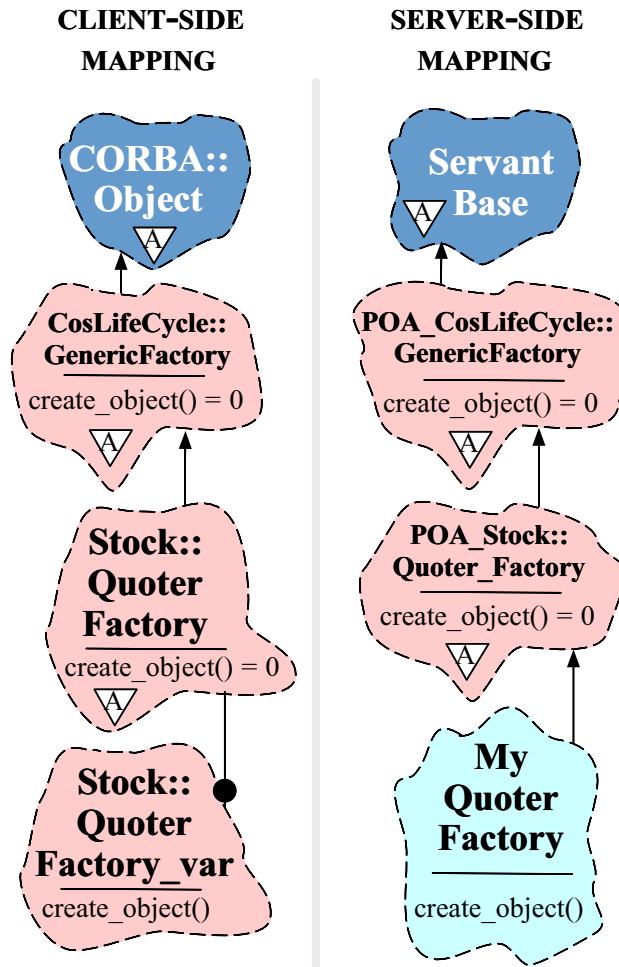
- The client-side mapping inherits all interfaces from the `Object` interface

  - Similar to Java

- The server-side mapping inherits all Servants from `ServantBase`

- Note that older versions of CORBA may not support this standard

OMG IDL Mapping Rules

- The CORBA specification defines mappings from CORBA IDL to various programming languages

  – *e.g.*, C++, C, Smalltalk, Java, COBOL

- Mapping OMG IDL to C++

  – Each `module` is mapped to a class or namespace
  – 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
    * A read-only attribute is only mapped to a single get method
  – An `Environment` is defined to carry exceptions in languages that lack this feature

Binding a Client to 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 supports various types of "factory" patterns

Programming with Object References

- Object references are represented by different generated types

  - `_ptr` → C++ pointer to object reference
    - ∗ Requires programmer management of reference ownership via `_duplicate` and `_release`
  - `_var` → Auto pointer to object reference
    - ∗ Internally manages reference ownership
  - `_out` → eases passing `out` parameters between client and server
    - ∗ Never used directly by user

A Client Program

```cpp
int main (int argc, char *argv[])
{
  // Manages refcounts.
  Stock::Quoter_var quoter;

  try { // Use a factory to bind to any quoter.
    Stock::Quoter_Factory_var qf =
      bind_service<Stock::Quoter_Factory>
        ("my quoter factory", argc, argv);
    if (CORBA::is_nil (qf)) return -1;
    const char *stock_name = "ACME ORB Inc.";
    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);
    CORBA::Long value =
      quoter->get_quote (stock_name);
    cout << stock_name << " = " << value << endl;
    // Destructors of *_var release memory.
  } catch (Stock::Invalid_Stock &) {
    cerr << stock_name << " not valid" << endl;
  } catch (...) { /* Handle exception... */ }
  quoter->remove ();
}
```

Obtaining an Object Reference
via the Naming Service

```cpp
static CORBA::ORB_ptr orb;
extern CosNaming::NamingContext_ptr name_context;

template <class T> typename T::_ptr_type /* trait */
bind_service (const char *n, int argc, char *argv[])
{
  CORBA::Object_var obj; // "First time" check.
  if (CORBA::is_nil (name_context)) {
    // Get reference to name service.
    orb = CORBA::ORB_init (argc, argv, 0);
    obj = orb->resolve_initial_references
      ("NameService");
    name_context =
      CosNaming::NamingContext::_narrow (obj);
    if (CORBA::is_nil (name_context)) return T::_nil
  }
  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);

  // 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 new ORB features

  - Asynchronous Method Invocations (AMI)
  - Server location independence (requires smart ORB)

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
{
  // ...
  interface Quoter { /* ... */ };

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

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

New Interfaces and Operations

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);
    void sell (in string name,
               inout long num_shares,
               in long min_value) raises (Invalid_Stock);
  };
  interface Trader_Factory { /* ... */ };
};
```

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

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

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 {
    string name;
    long value;
    double change;
    long volume;
  };
  typedef sequence<Stock_Info> Info;
  exception No_Such_Stock { Names stock; };

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

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
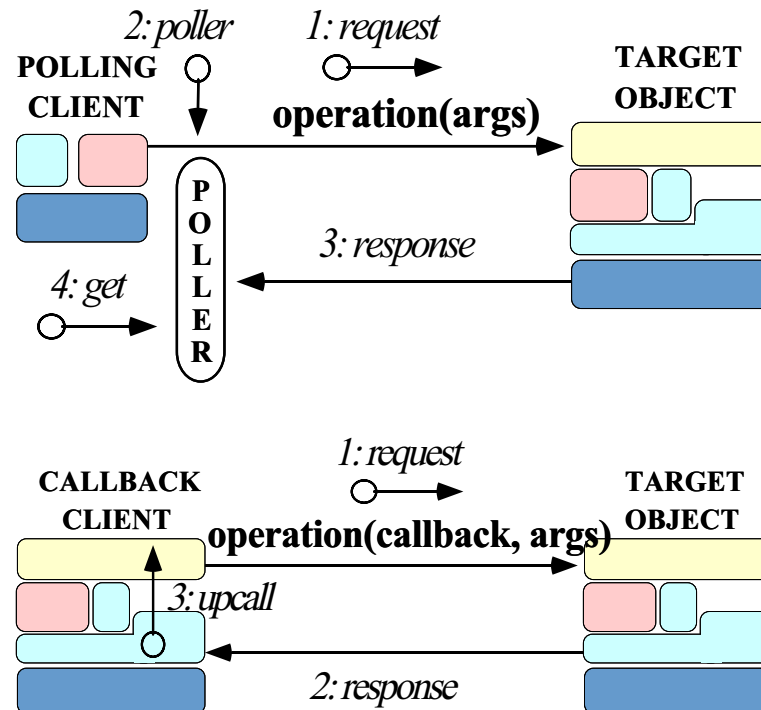
AMI Callback Overview

Implied-IDL for client-side:
```
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
      (AMI_QuoterHandler handler,
       in string stock);
  };
};
```
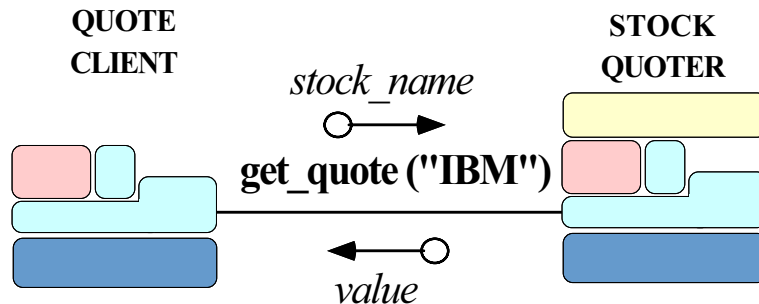
Quoter IDL Interface:
```
module Stock {
 interface Quoter {
   // Two-way operation to
   // get current stock value.
   long get_quote
     (in string stock_name);
 };
 // ...
}
```

Example: Synchronous Client

QUOTE CLIENT   *stock_name*   **STOCK QUOTER**

get_quote ("IBM")

value
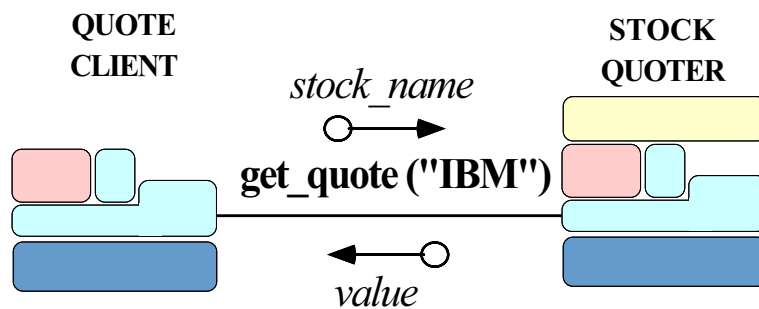
IDL-generated stub:

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

Application:

```
// NASDAQ abbreviations for ORB vendors.
static const char *stocks[] =
{
   "IONAY" // IONA Orbix
   "INPR"  // Inprise VisiBroker
   "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
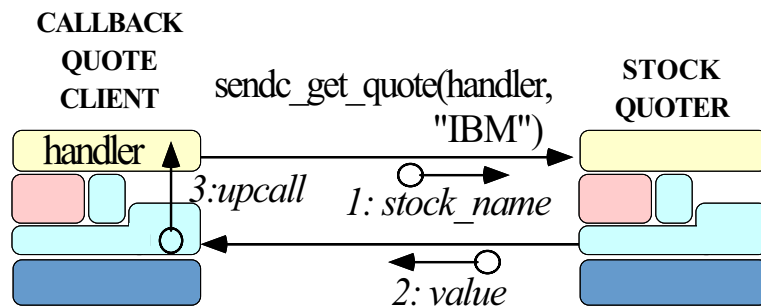
Asynchronous stub:
```
void
Stock::Quoter::sendc_get_quote
        (AMI_QuoterHandler_ptr,
         const char *name)
{
 // 1. Setup connection
 // 2. Store reply handler
 //    in ORB
 // 3. Marshal
 // 4. Send request
 // 5. Return
};
```

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_ << " stock: "
        << ami_return_val << endl;
    // Decrement global reply count.
    reply_count--;
 }
private:
  CORBA::String_var stock_;
};
```

Example: AMI Callback Client (cont'd)

```cpp
// 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]);
```

```cpp
// 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],
     stocks[i]);

// ...

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

Additional Information on AMI

- See Asynchronous Messaging specification

  - www.cs.wustl.edu/~schmidt/CORBA-docs/00-02-05.pdf.gz

- See Vinoski's CACM article on CORBA 3.0 for more info.

  - www.cs.wustl.edu/~schmidt/vinoski-98.pdf.gz

- See our papers on AMI

  - www.cs.wustl.edu/~schmidt/report-doc.html
  - www.cs.wustl.edu/~schmidt/ami1.ps.gz
  - www.cs.wustl.edu/~schmidt/ami2.ps.gz

- See TAO release to experiment with working AMI examples

  - `$TAO_ROOT/examples/AMI/`

CORBA ORB Architecture

NAMING SERVICE		IDL COMPILER	
	INTERFACE REPOSITORY		IMPLEMENTATION REPOSITORY

in args

operation()

out args + return value

CLIENT

OBJECT (SERVANT)

DII

IDL STUBS

ORB INTERFACE

IDL SKELETON

DSI

OBJECT ADAPTER

GIOP/IIOP

ORB CORE

OS KERNEL

OS I/O SUBSYSTEM

NETWORK INTERFACES

OS KERNEL

OS I/O SUBSYSTEM

NETWORK INTERFACES

NETWORK
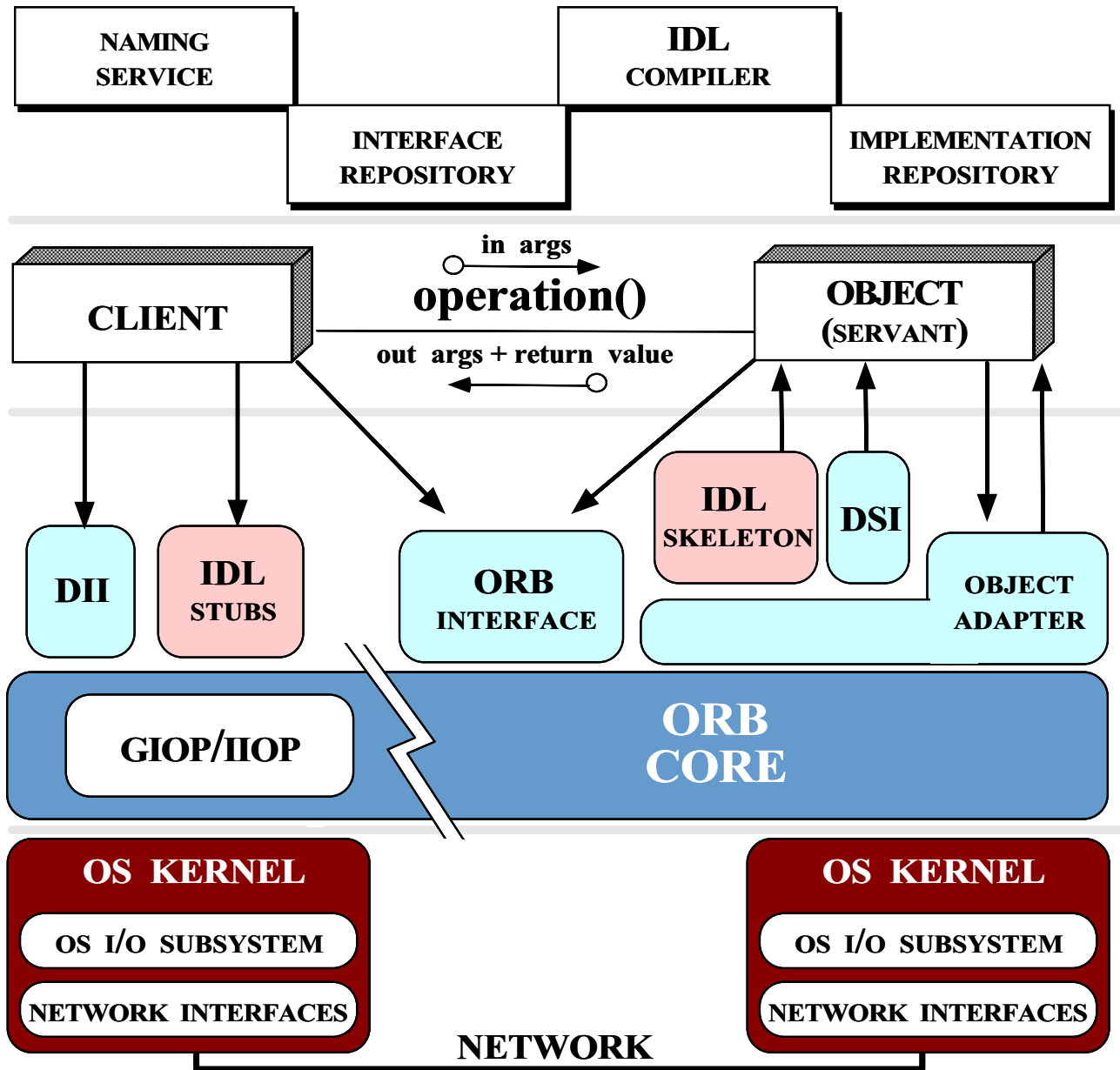
Overview of CORBA Components

- The CORBA specification contains several 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) and Dynamic Skeleton Interface (DSI)
    - Portable Object Adapter (POA)
    - Interface and Implementation Repositories

OMA Reference Model Interface Categories

```
┌──────────────────┐   ┌──────────────────┐   ┌──────────────────┐
│   APPLICATION    │   │      DOMAIN      │   │      COMMON      │
│    INTERFACES    │   │    INTERFACES    │   │    FACILITIES    │
└──────────────────┘   └──────────────────┘   └──────────────────┘
    ↕    ↕    ↕            ↕    ↕    ↕            ↕    ↕    ↕
┌──────────────────────────────────────────────────────────────────┐
│              OBJECT  REQUEST  BROKER                               │
└──────────────────────────────────────────────────────────────────┘
                      ↕    ↕    ↕
              ┌──────────────────┐
              │      OBJECT      │
              │     SERVICES     │
              └──────────────────┘
```
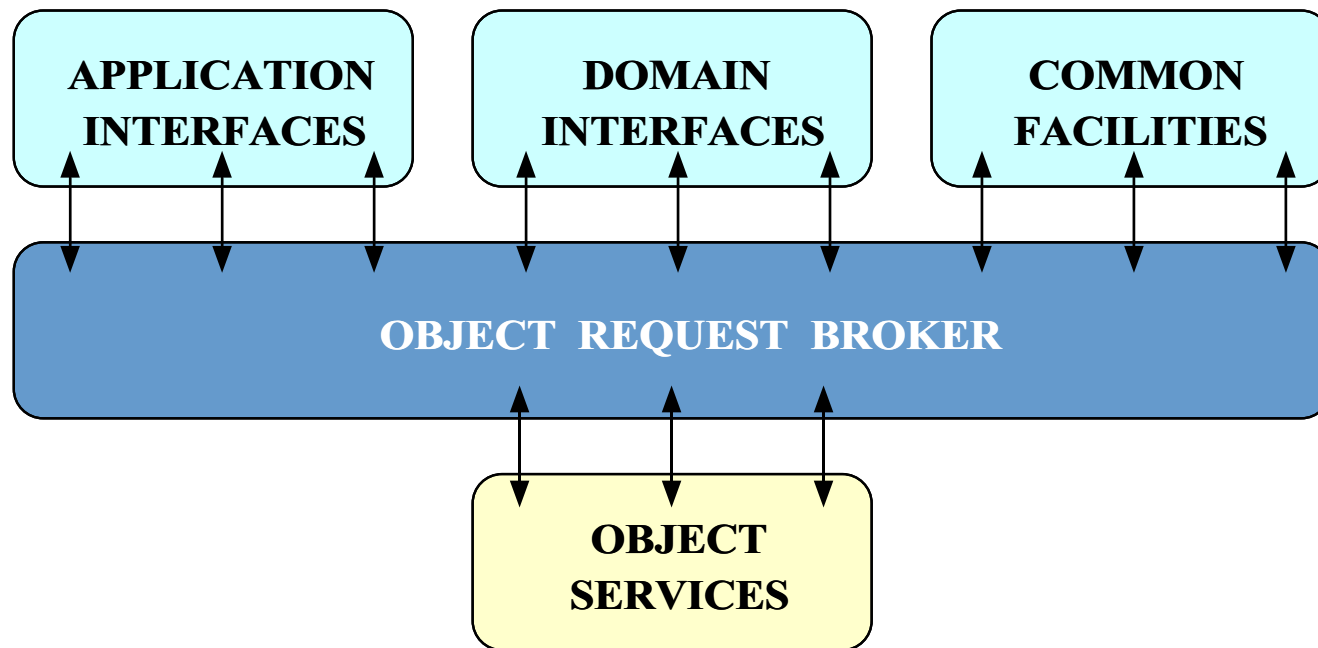
The Object Management Architecture (OMA) Reference Model
describes the interactions between various CORBA components and
layers

Overview of the ORB Core
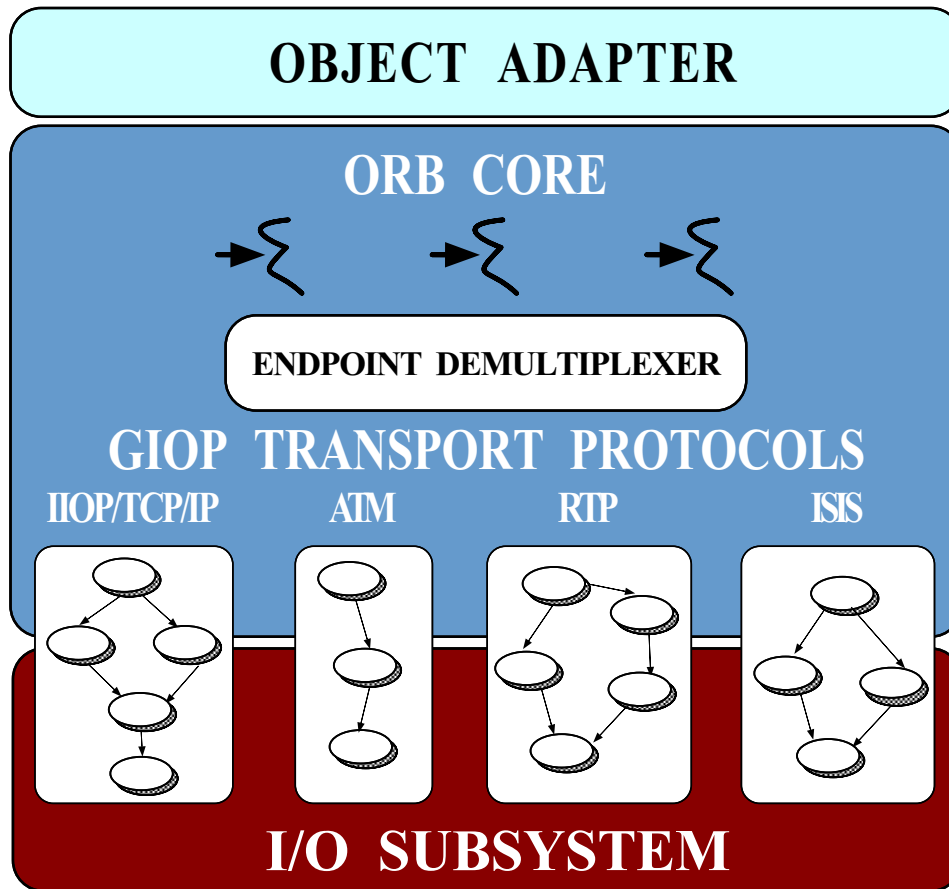
Features

- Connection/memory management

- Request transfer

- Endpoint demuxing

- Concurrency control

CORBA Interoperability Protocols

STANDARD CORBA PROGRAMMING API

ORB MESSAGING COMPONENT	**GIOP**	**GIOP**LITE	**ESIOP**
ORB TRANSPORT ADAPTER COMPONENT	**IIOP**	**VME-IOP**	**ATM-IOP** RELIABLE SEQUENCED
TRANSPORT LAYER	**TCP**	**VME**	**AAL5**
NETWORK LAYER	**IP**	**DRIVER**	**ATM**

PROTOCOL CONFIGURATIONS
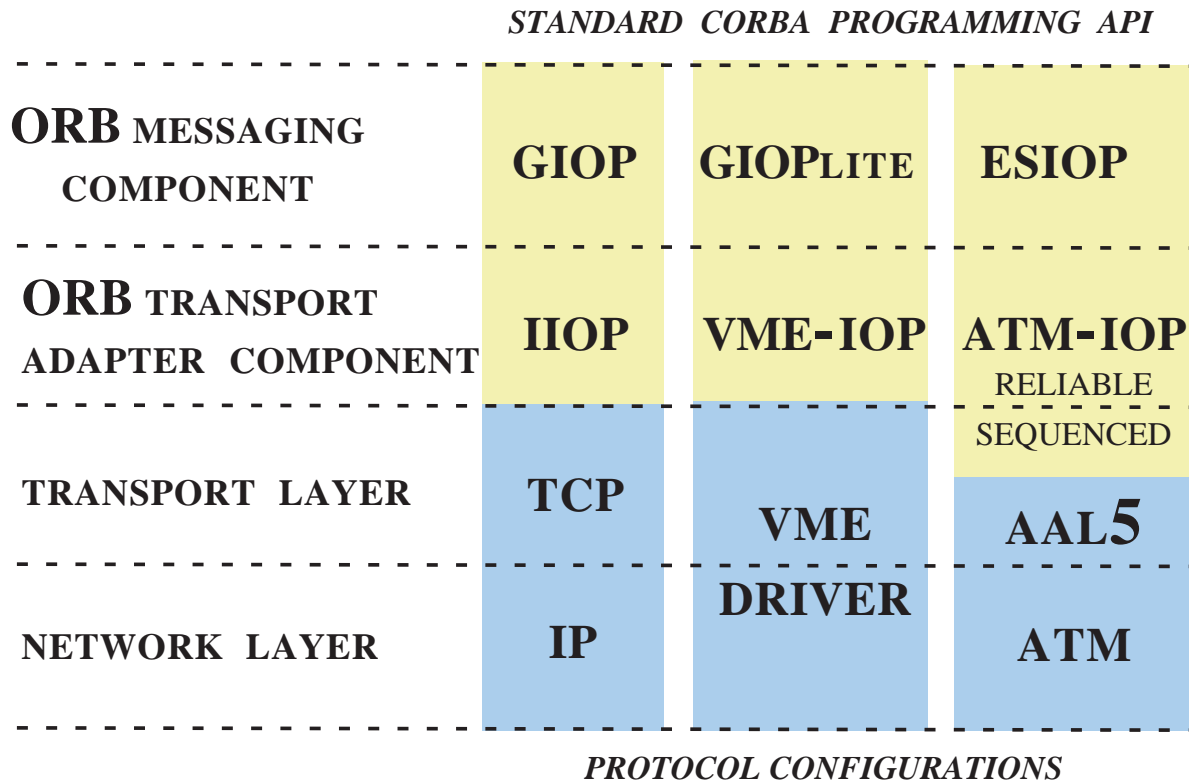
- GIOP

  – Enables ORB-to-ORB interoperability

- IIOP

  – Works directly over TCP/IP, no RPC

- ESIOPs

  – *e.g.*, DCE, DCOM, wireless, etc.

GIOP Overview

- **Common Data Representation** (CDR)

  - Transfer syntax mapping OMG-IDL data types into a bi-canonical low-level representation
    * Supports variable byte ordering and aligned primitive types

- **Message transfer**

  - Request multiplexing, *i.e.*, shared connections
  - Ordering constraints are minimal, *i.e.*, can be asynchronous

- **Message formats**

  - Client: `Request, CancelRequest, LocateRequest`
  - Server: `Reply, LocateReply, CloseConnection`
  - Both: `MessageError`

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;
  };
  // ...
```

IIOP Overview

- IIOP adds to GIOP semantics for TCP/IP connection management

- IIOP bundled with Netscape 4.0

- Inter-ORB Engine available from SunSoft

  - ftp://ftp.omg.org/pub/interop/iiop.tar.Z

- TAO is originally based on SunSoft IIOP

  - However, TAO adds *many* enhancements and optimizations
    * www.cs.wustl.edu/~schmidt/JSAC-99.ps.gz
    * www.cs.wustl.edu/~schmidt/TAO.html

Interpreted vs. Compiled (De)marshaling
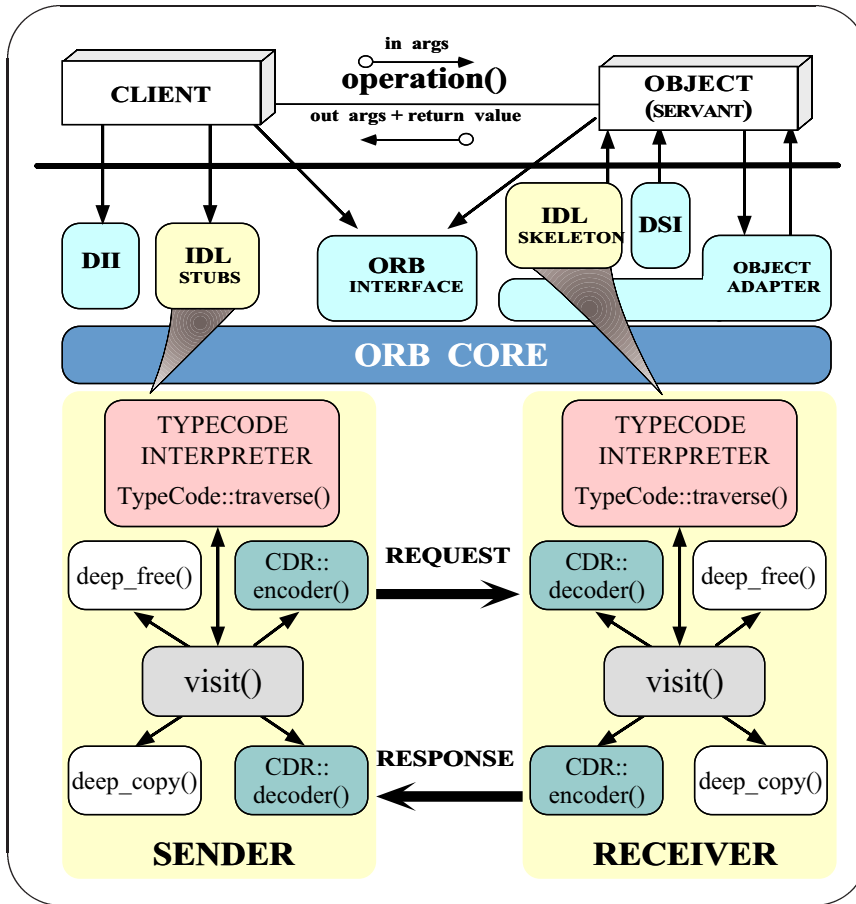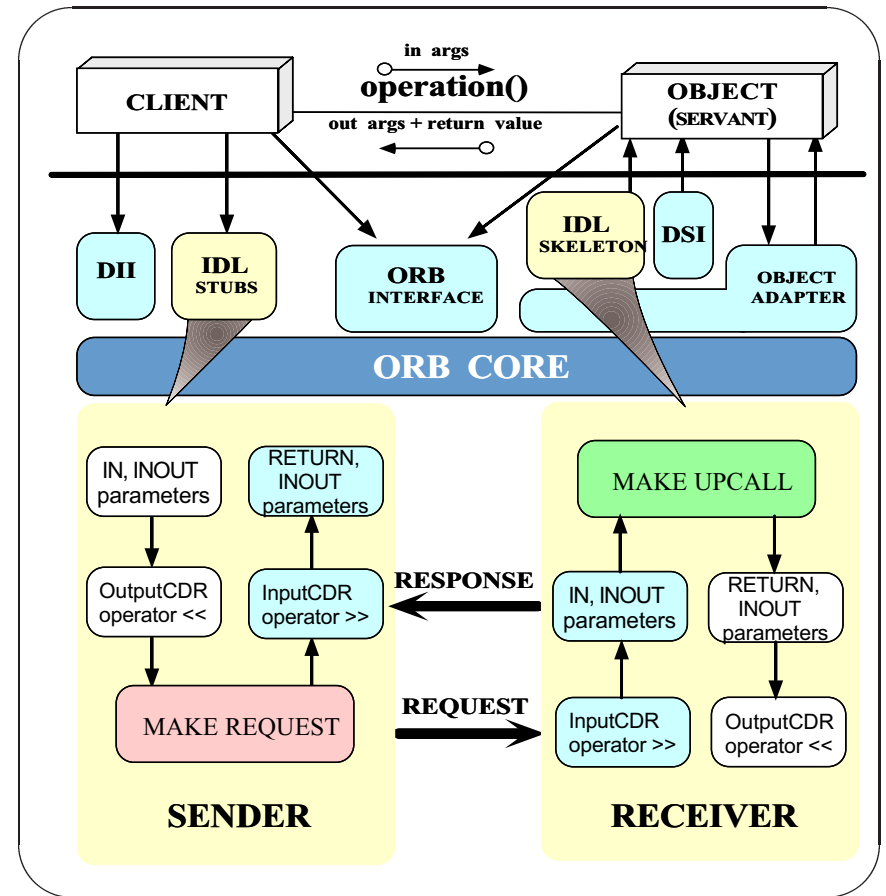
Interface Definition Language (IDL)

- Motivation

  - Developing flexible distributed applications on heterogeneous platforms requires a 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.

Related IDLs

- Many IDLs are currently available, *e.g.*,

    - OSI ASN.1
    - OSI GDMO
    - SNMP SMI
    - DCE IDL
    - Microsoft's IDL (MIDL)
    - OMG IDL
    - ONC's XDR

- However, many of these are *procedural* IDLs

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

CORBA Interface Definition Language (IDL)

- OMG IDL is an object-oriented interface definition language

  - Used to specify interfaces containing *operations* and *attributes*
  - OMG IDL support interface inheritance (both single and multiple inheritance)

- OMG IDL is designed to map onto multiple programming languages

  - *e.g.*, C, C++, Smalltalk, COBOL, Modula 3, DCE, Java, etc.

- OMG IDL is similar to Java `interfaces` and C++ abstract classes

Application Interfaces

- Interfaces described using OMG IDL may be application-specific, *e.g.*,

  - Databases
  - Spreadsheets
  - Spell checker
  - Network manager
  - Air traffic control
  - Documents
  - Medical imaging systems

- Objects may be defined at any level of granularity

  - *e.g.*, from fine-grained GUI objects to multi-megabyte multimedia "Blobs"

OMG IDL Features

- OMG IDL is similar to Java `interfaces` or C++ abstract classes

  - It is not a complete programming language, however, since it only defines *interfaces*

- OMG IDL supports the following features:

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

OMG IDL Differences from C++ and Java

- No control constructs

- No data members

- No pointers

- 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

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

Primary advantages of SII are *simplicity*, *typesafety*, and *efficiency*

Stubs use the Proxy Pattern

1: METHOD
CALL

: BROKER

: QUOTER
PROXY

4: METHOD
RETURN

2: FORWARD
REQUEST

3: RESPONSE

CLIENT

: QUOTER

NETWORK

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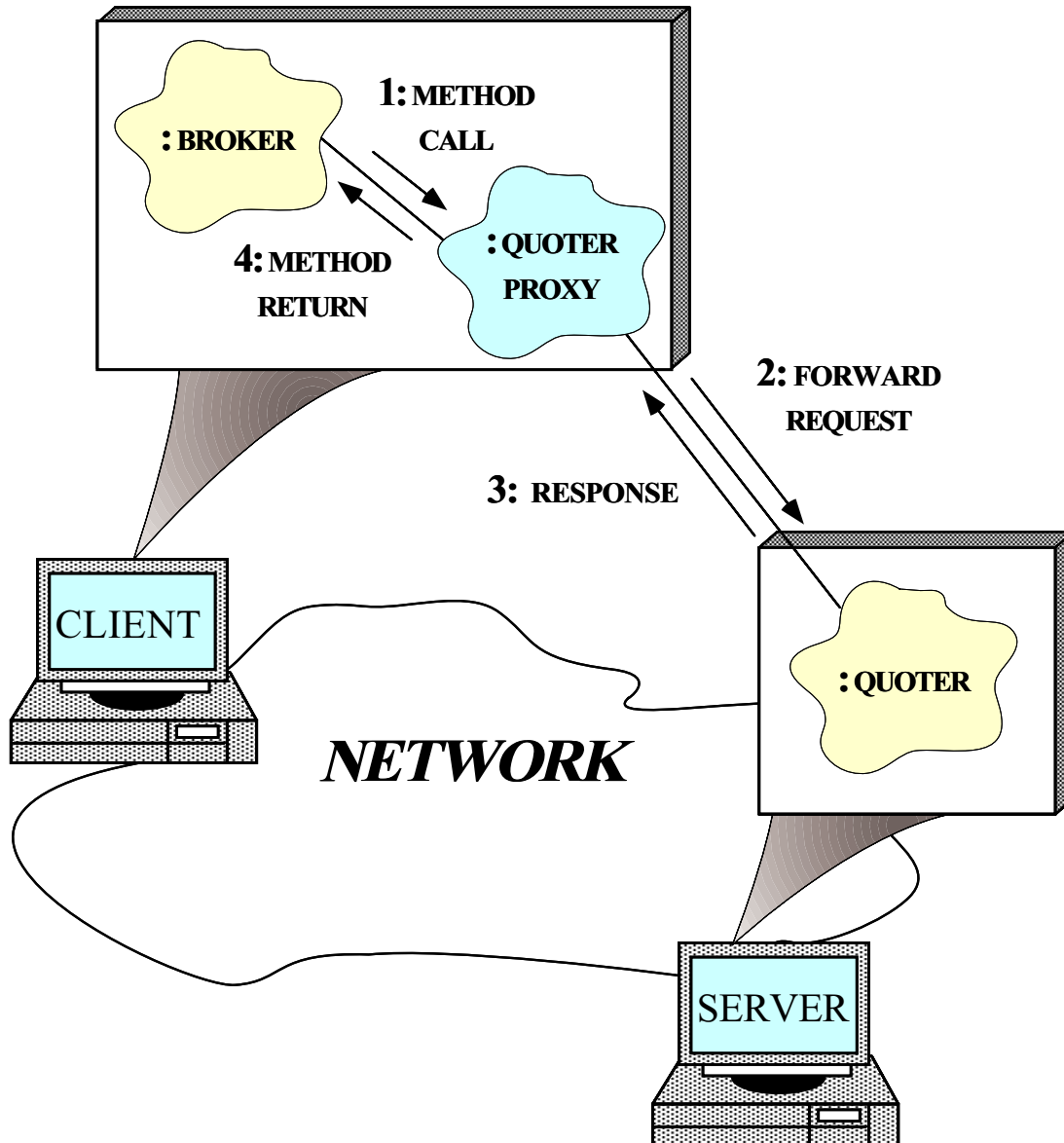
Intent: provide a surrogate for another object that
controls access to it

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 SII

  - *e.g.*, it supports *deferred synchronous* invocation

- However, the DII is also more complicated, less typesafe, and inefficient

An Example DII Client

```cpp
// 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.
request->add_in_arg () <<= "IONAY";
request->set_return_type (CORBA::_tc_long);

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

// Retrieve/print value.
if (request->return_value () >>= value)
  cout << "Current value of IONA stock: "
       << value << endl;
```

This example is *much* more complicated and inefficient than simply using SII...

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-side that the DII provides on the client-side

  - It is defined primarily to build ORB "Bridges"
  - The DSI lets server code handle arbitrary invocations on CORBA objects

Object References

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

  - It identifies the object's location

- Object references may be passed among processes on separate hosts

  - The underlying CORBA ORB will correctly convert object references into a form that can be transmitted over the network
  - The ORB provides the receiver with a pointer to a proxy in its own address space
    * This proxy refers to remote object implementation

- Object references are a powerful feature of CORBA

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

Object Reference in URL Format
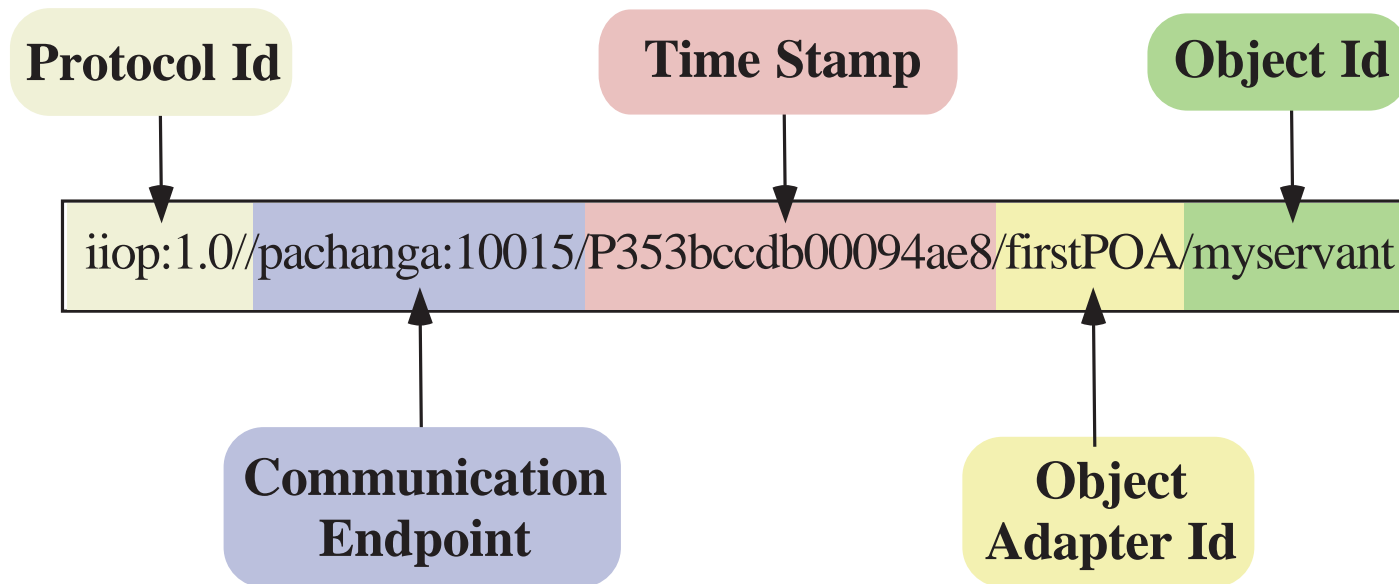
- Transient object reference

Using Object References for Publish/Subscribe Architectures

CONSUMER
A

Notifier
ObjRef

A :
Consumer

1: subscribe (ObjRef$_A$)

4: push (event)

: Notifier

Con
ObjRef$_A$

Con
ObjRef$_B$

3: push (event)

Notifier
ObjRef

2: subscribe (ObjRef$_B$)

4: push (event)

NOTIFIER

NETWORK

B :
Consumer

Notifier
ObjRef

CONSUMER
B

SUPPLIER
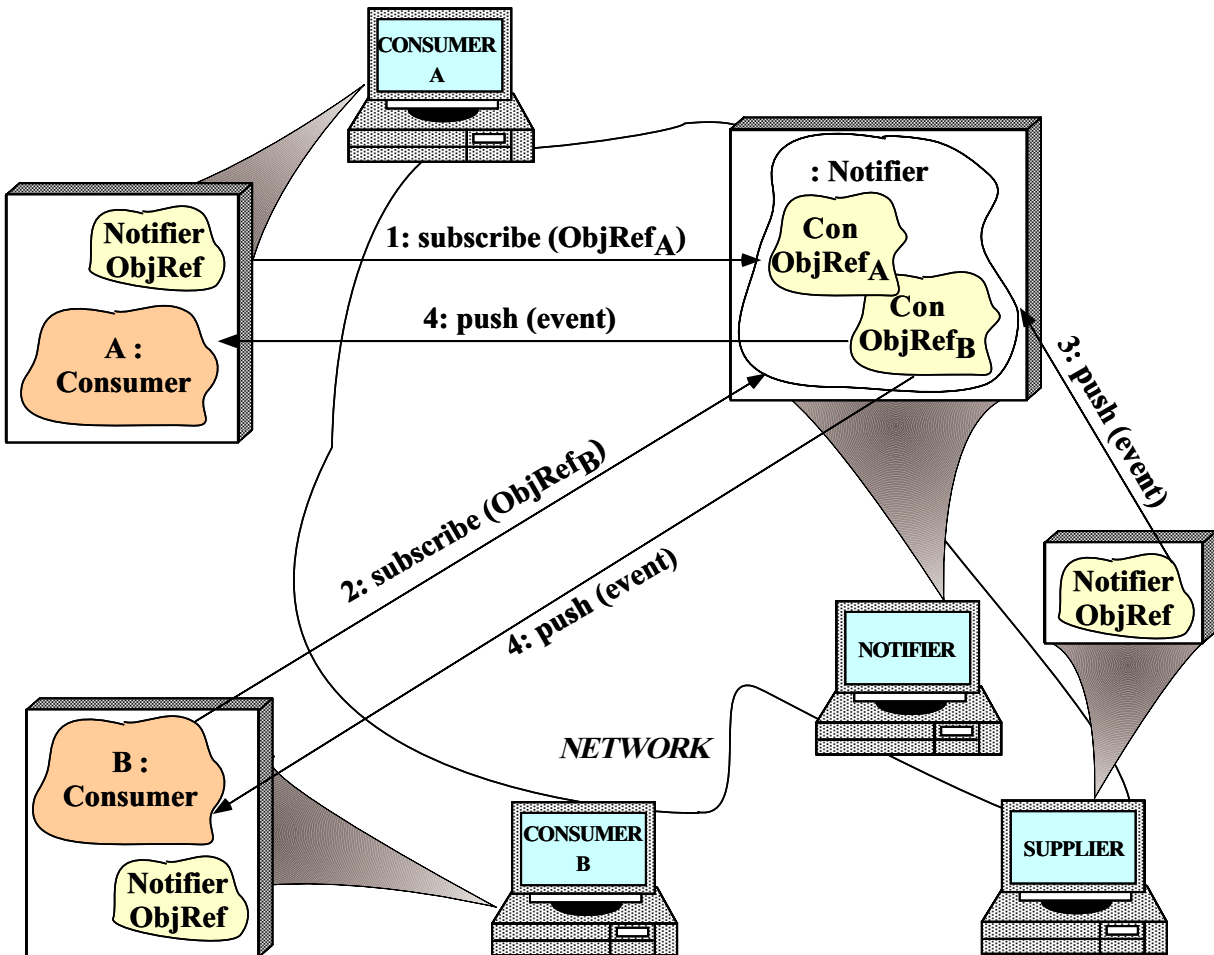
Note the use of the Observer pattern

Event Receiver Interface

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

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

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

A `Consumer` is called back by the Notifier

Notifier Interface

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

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

A Notifier publishes `Events` to subscribed
Consumers

Notifier Implementation

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

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

Advanced CORBA Features
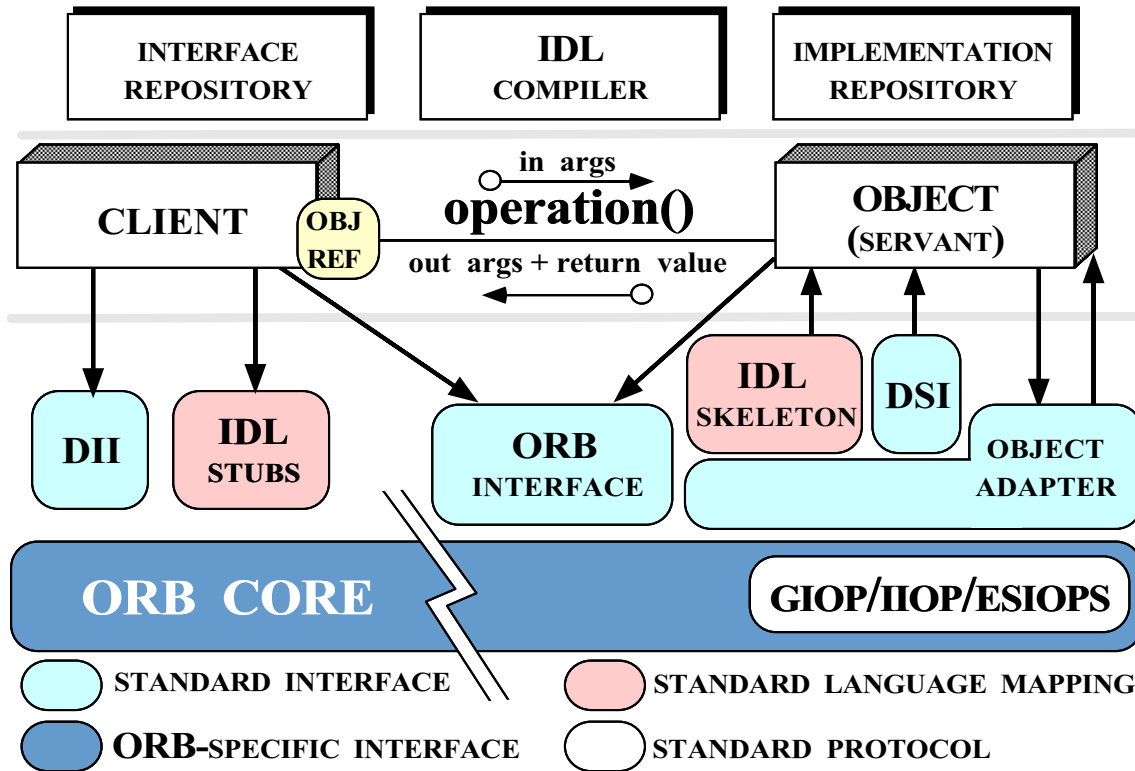
Features

- Portable Object Adapter

- Multi-threading

- Implementation Repository

- CORBA Component Model

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

Overview of the Portable Object Adapter (POA)
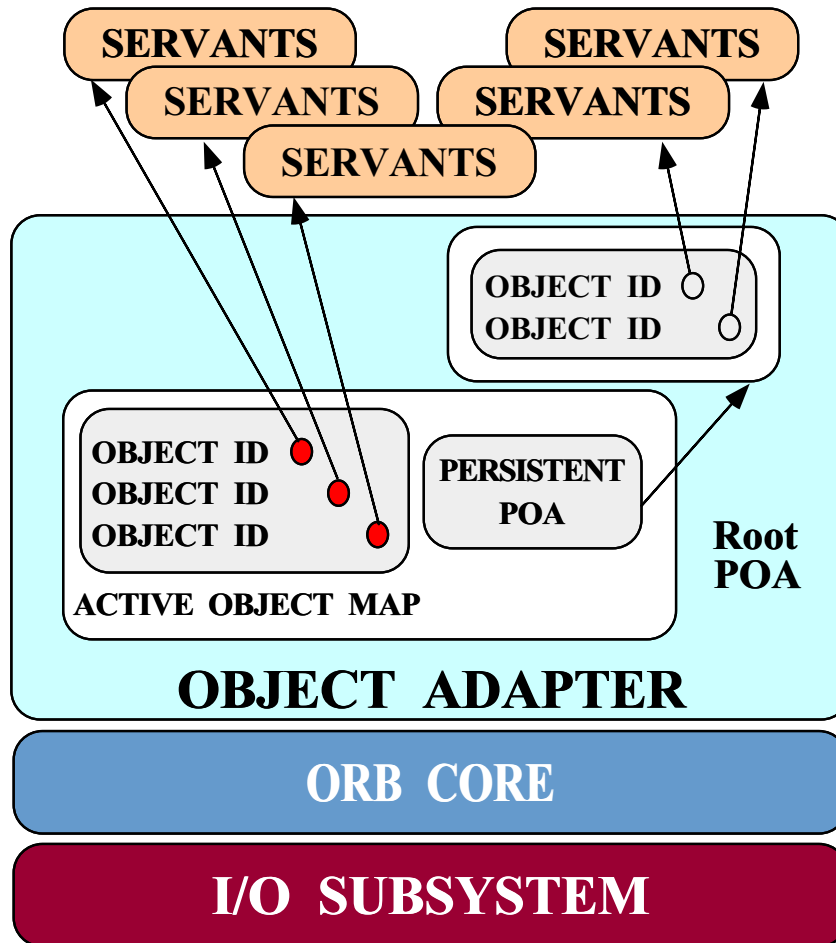
POA Features

- Creates object refs

- Activates and deactivates objects

- Etherealizes and incarnates servants

- Maps requests to servants

The POA is very important for certain applications

- *e.g.*, telecom MIBs, enterprise servers

SERVANTS
SERVANTS
SERVANTS
SERVANTS
SERVANTS

OBJECT ID
OBJECT ID

OBJECT ID
OBJECT ID
OBJECT ID

ACTIVE OBJECT MAP

PERSISTENT POA

Root POA

OBJECT ADAPTER

ORB CORE

I/O SUBSYSTEM

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
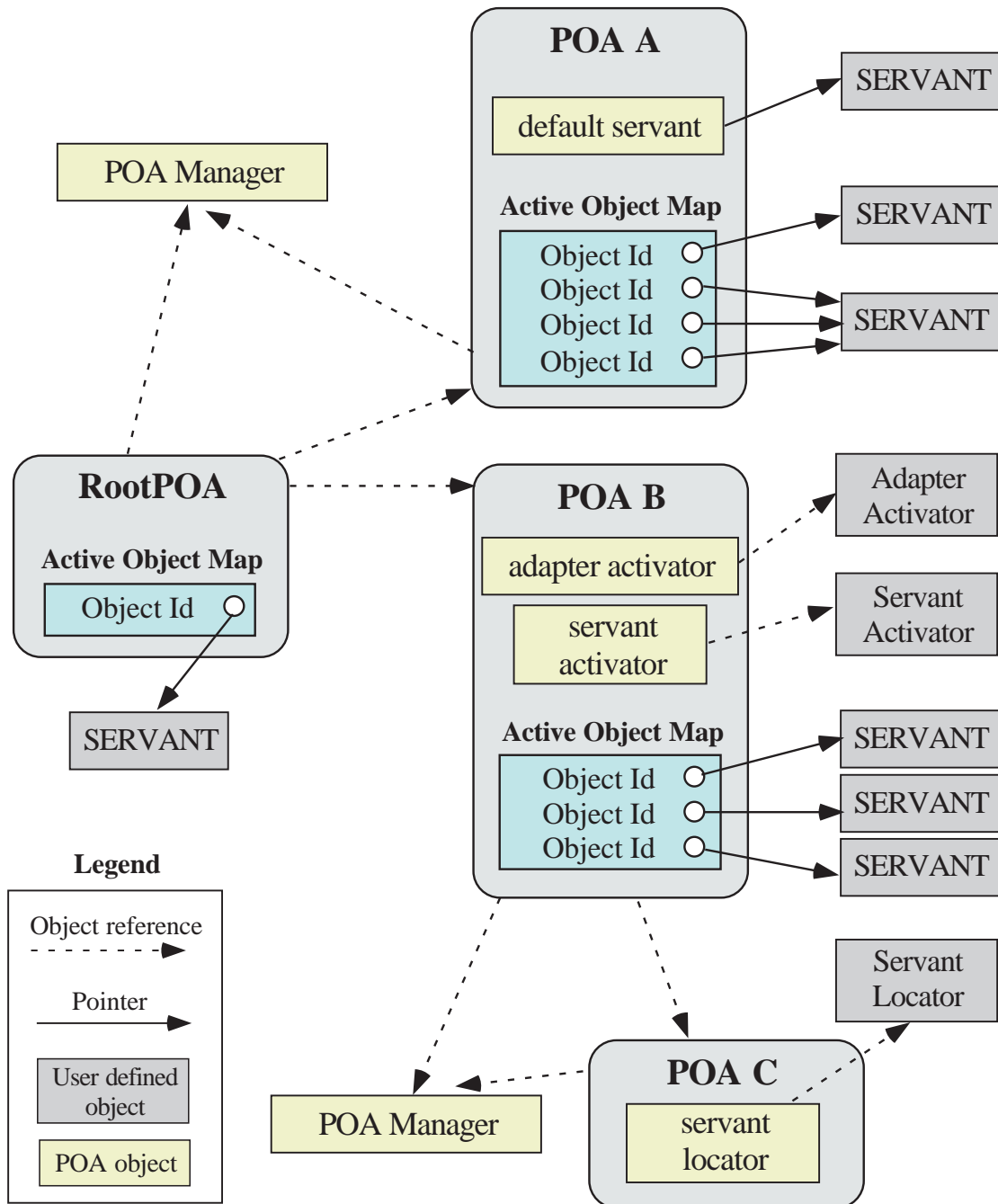
POA Components

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

- **Server**: Computational context for servants

  - Generally, a server corresponds to a process
  - Client and server are "roles" - a program can play both roles

- **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

- **Policy**: Specifics the characteristics of a POA or child POA

POA Components (cont'd)

* Object Id: A value that is used by the POA and by the implementation to identify a particular CORBA object

    - Object Id values may be assigned by the POA, or by the user implementation
    - Object Id values are hidden from clients, encapsulated by references
    - Object Ids have no standard form; they are managed by the POA as uninterpreted octet sequences

* Object Reference: Encapsulates an Object Id, a POA identity, and transport profiles

* POA: A namespace for Object Ids and a namespace for child POAs

    - Nested POAs form a hierarchical name space for objects in servers

POA Components (cont'd)

- POA Manager: Encapsulates the processing state of associated POAs

  – Can dispatch, hold, or discard requests for the associated POAs and deactivate POA(s)

- Servant Manager: Two kinds of callback objects used to incarnate and etherealize servants on demand

  – `ServantActivator` $\rightarrow$ first time
  – `ServantLocator` $\rightarrow$ each time

- 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

POA Architecture in UML

PortableServer::AdapterActivator
(from PortableServer)

unknown_adapter()

0..n

PortableServer::POAManager
(from PortableServer)

activate()
hold_requests()
discard_requests()
deactivate()

1..n

the_activator

the_manager

1..1

1..1 the_parent 0..n

PortableServer::POA
(from PortableServer)

the_name : string
the_parent : PortableServer::POA
the_manager
: PortableServer::POAManager
the_activator
: PortableServer::AdapterActivator
the_servant_manager
: PortableServer::ServantManager

create_POA()
find_POA()
destroy()
create_thread_policy()
create_lifespan_policy()
create_id_uniqueness_policy()
create_id_assignment_policy()
create_implicit_activation_policy()
create_servant_retention_policy()
create_request_processing_policy()
get_instance_manager()
set_instance_manager()
get_servant()
set_servant()
activate_object()
activate_object_with_id()
deactivate_object()
create_reference()
create_reference_with_id()
servant_to_id()
servant_to_reference()
reference_to_servant()
reference_to_id()
id_to_servant()
id_to_reference()

PortableServer::ServantManager
(from PortableServer)

0..n

the_servant_manager

0..1

0..1

PortableServer::ServantLocator
(from PortableServer)

preinvoke()
postinvoke()

PortableServer::ServantActivator
(from PortableServer)

incarnate()
etherealize()

PortableServer::Cookie
(from PortableServer)

PortableServer::Servant
(from PortableServer)

CORBA::Current
(from CORBA Core)

PortableServer::Current
(from PortableServer)

get_POA()
get_object_id()

CORBA::PolicyList
(from CORBA Core)

: CORBA::Policy

1..n

0..n

CORBA::Policy
(from CORBA Core)

copy()
destroy()

PortableServer::ObjectId
(from PortableServer)

POA Manager Processing States
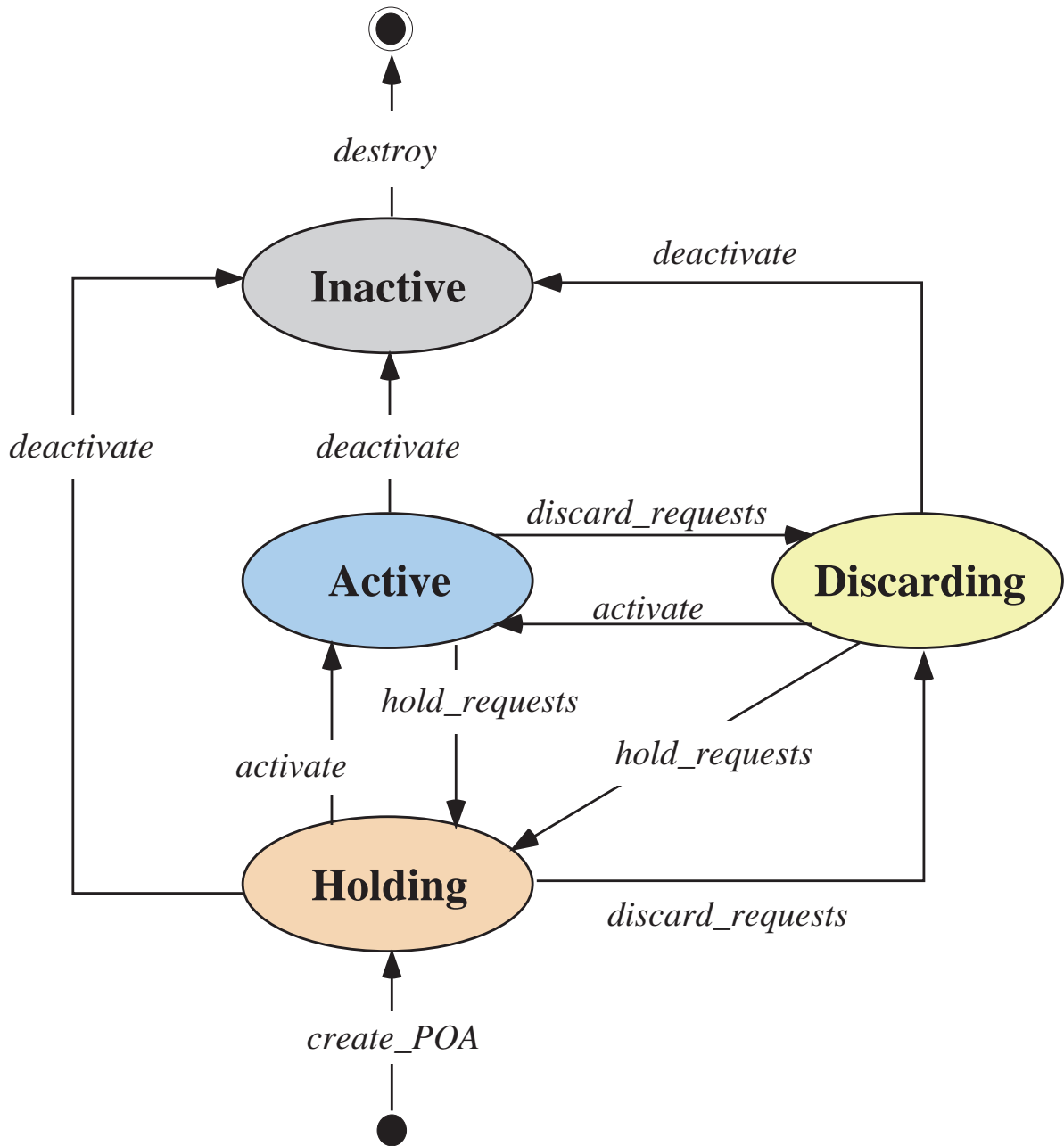
Getting the Root POA

```
// 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);

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

Creating a Child POA

```
CORBA::PolicyList policies (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_ptr child_poa =
  root_POA->create_POA
            ("child_poa",
             PortableServer::POAManager::_nil (),
             policies);

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

Explicit Activation with
POA-assigned Object Ids

```
// IDL
interface Quoter /* ... */
{
  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;
PortableServer::ObjectId_var oid =
  poa->activate_object (quoter);
PortableServer::POA_Manager_var poa_manager =
  poa->the_POAManager ();
poa_manager->activate ();
orb->run ();
```

Explicit Activation With User-assigned Object Ids

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

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

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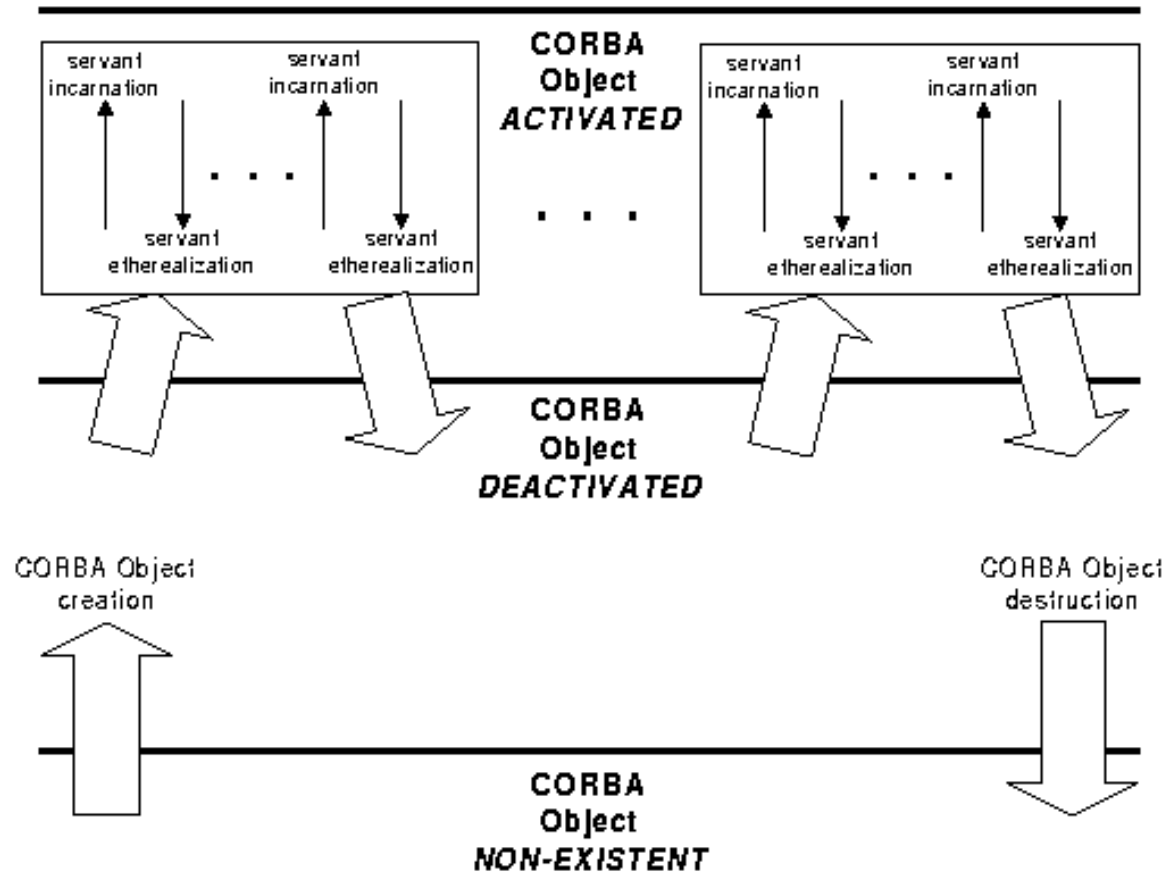
Creating References Before Activation

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

// ...later...

My_Quoter *quoter = new My_Quoter;
poa->activate_object_with_id (oid, quoter);
```

Request Lifecycle for POA

Servant Activator Definition

```
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

```cpp
// 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, "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)
      delete servant;
  }
};
```

Servant Locator Definition

```
typedef ServantBase *Servant;

// Skeleton class
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

```cpp
// 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 str =
      PortableServer::ObjectId_to_string (oid);
    Object_State state = database_lookup (str);
    if (val == -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);
    delete servant;
  }
};
```

Registering Servant Locators

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

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 ();
```

Additional Information on the POA

- See 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/examples/POA/`

Motivation for Concurrency in CORBA

CONCURRENT SERVER

PARALLEL SERVER
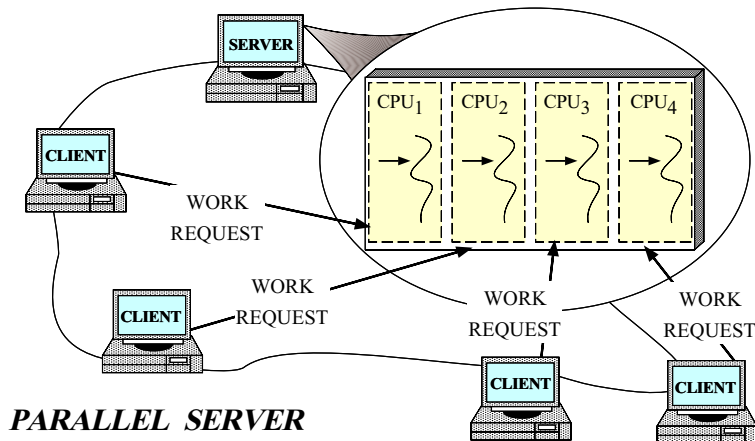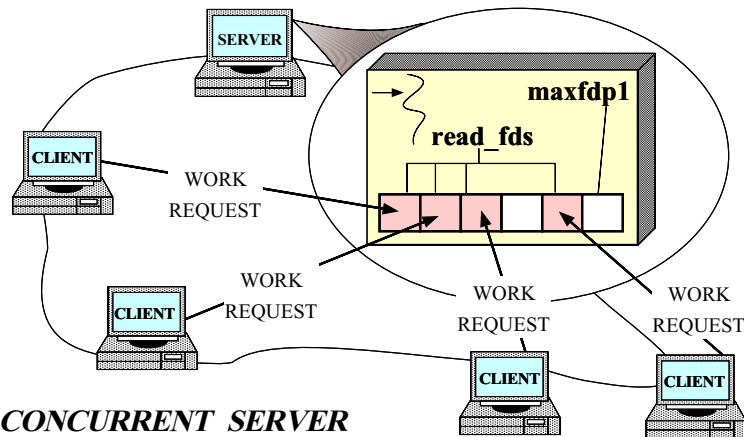
- *Leverage hardware/software*

  - *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

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:

  - *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
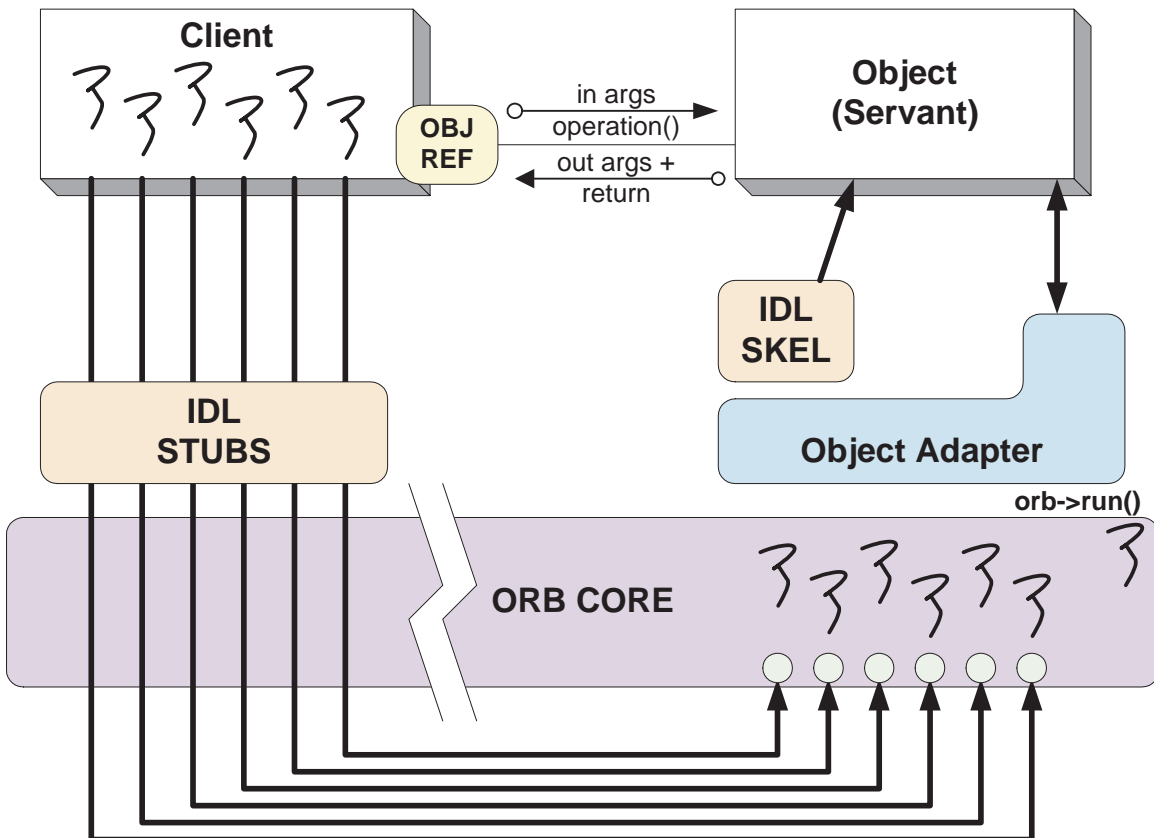
Pros

- 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

The server creates a single 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 the servant, which registers with rootPOA and Naming Service implicitly.
  My_Quoter_Factory factory (factory_name);

  // Block indefinitely waiting for incoming invocations and dispatch upcalls.
  orb_manager.run ();

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

The ORB's `svc.conf` file

```
static Resource_Factory "-ORBResources global -ORBReactorType select_mt"
static Server_Strategy_Factory "-ORBConcurrency thread-per-connection"
```

Thread-per-Connection
Quoter Interface

Implementation of the `Quoter` IDL interface

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

class My_Quoter :
  virtual public POA_Stock::Quoter,
  virtual public PortableServer::RefCountServantBase
{
public:
  // Constructor.
  My_Quoter (const char *name);

  // Returns the current stock value.
  long get_quote (const char *stock_name);
    throw (CORBA::SystemException,
           Quoter::InvalidStock);
private:
  // Serialize access to database.
  Thread_Mutex lock_;

  // Maintain request count.
  static COUNTER req_count_;
};
```

Thread-per-Connection Quoter Implementation

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

```cpp
long My_Quoter::get_quote (const char *stock_name)
{
  Guard<Thread_Mutex> guard (lock_);

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

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

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

  return value;
}
```
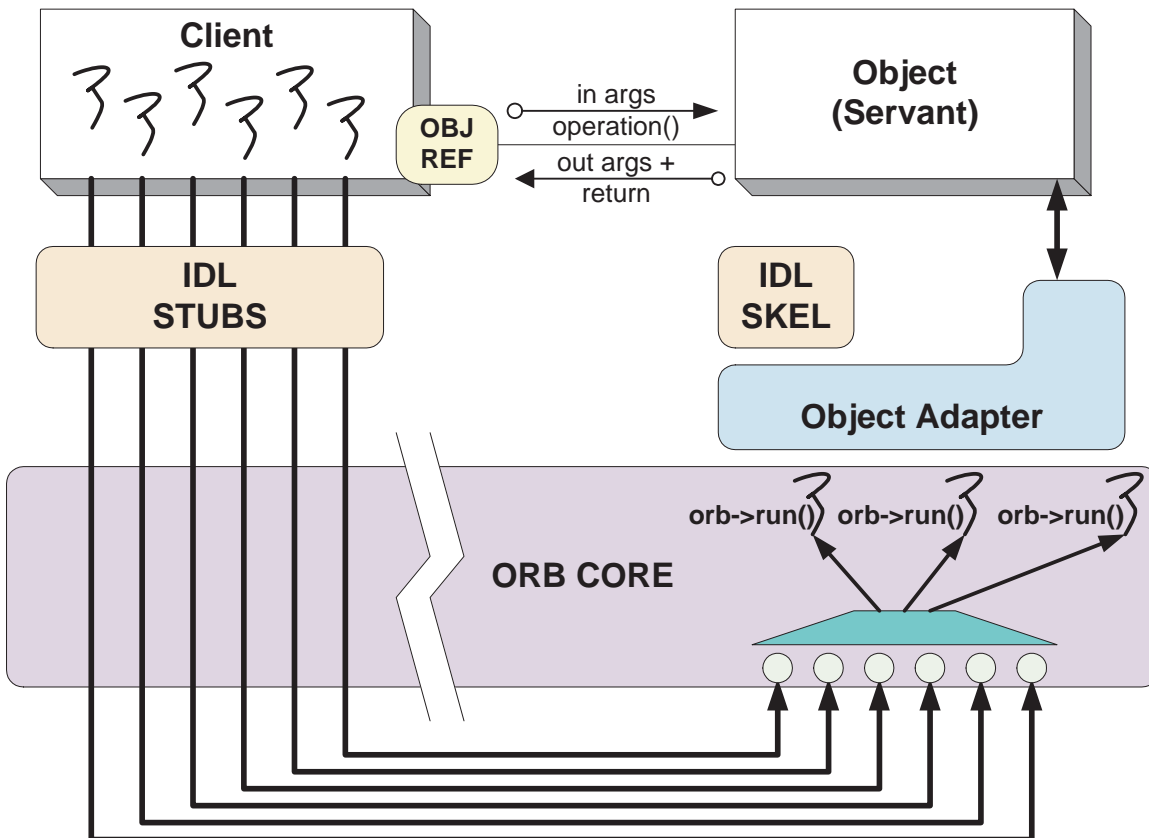
Thread Pool

- 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 `spawn_n` method for spawning 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

```cpp
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 (factory_name);

    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_reinterpret_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"
```
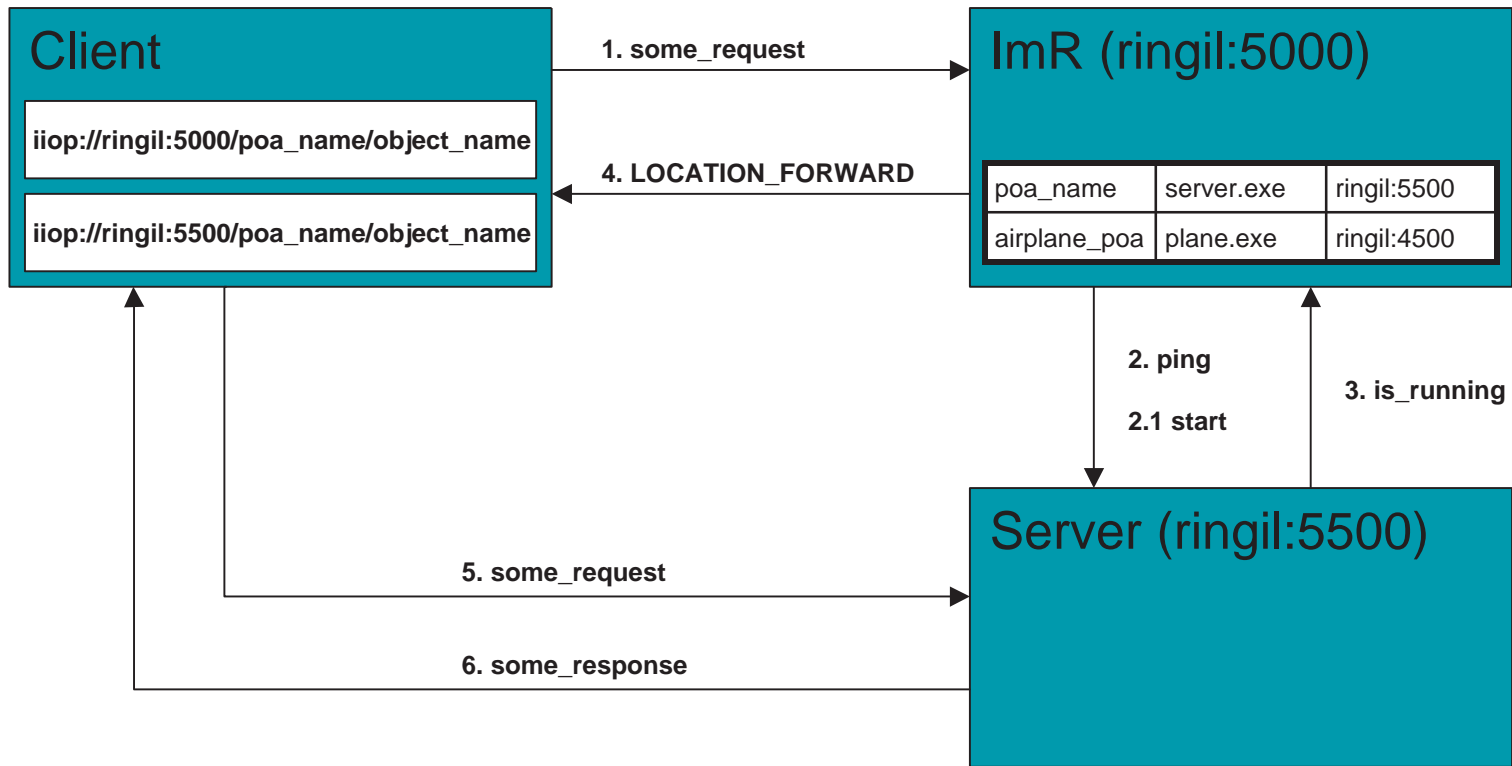
Additional Information on CORBA Threading

- See Real-time CORBA 1.0 specification

    – www.cs.wustl.edu/~schmidt/RT-ORB-std-new.pdf.gz

- See our papers on CORBA Threading

    – www.cs.wustl.edu/~schmidt/CACM-arch.ps.gz
    – www.cs.wustl.edu/~schmidt/RT-perf.ps.gz
    – www.cs.wustl.edu/~schmidt/COOTS-99.ps.gz
    – www.cs.wustl.edu/~schmidt/orc.ps.gz
    – 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

Typical Implementation Repository Use-case

Client		ImR (ringil:5000)		

1. some_request

4. LOCATION_FORWARD

iiop://ringil:5000/poa_name/object_name

iiop://ringil:5500/poa_name/object_name

poa_name	server.exe	ringil:5500
airplane_poa	plane.exe	ringil:4500

2. ping

2.1 start

3. is_running

Server (ringil:5500)

5. some_request

6. some_response

# 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
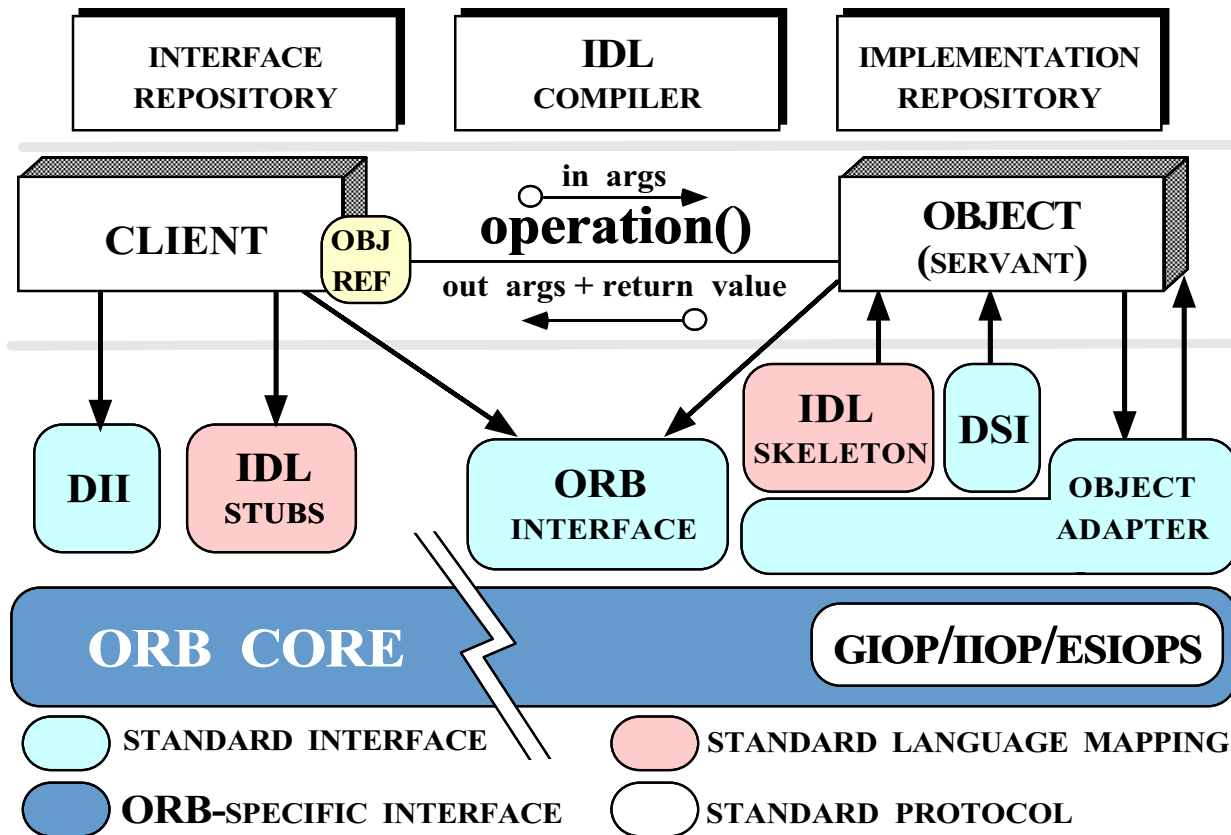
The CORBA Component Model
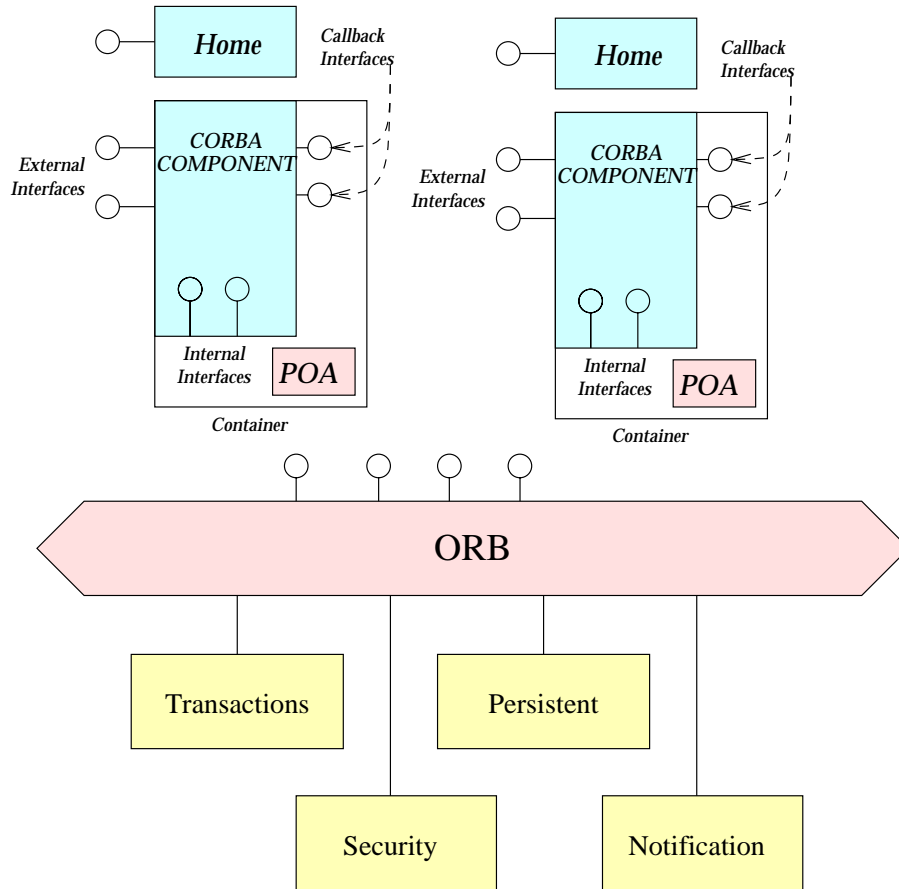
~schmidt/CBSE.ps.gz

- **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

Criteria

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

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

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-side
    * *i.e.*, the POA spec

- Most ORB implementations now support IIOP or GIOP robustly...

  – However, higher-level CORBA services aren't covered by ORB interoperability spec...

Portability

- To improve portability, 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

- Porting applications from ORB-to-ORB will be limited, however, until conformance tests become common-place

  - www.opengroup.org/testing/testsuites/vsorb.htm

- Moreover, CORBA spec doesn't really handle concurrency in a portable manner

Feature Limitations

- 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 (cont'd)

- Many ORBs do not yet support passing objects-by-value (OBV)

  – However, CORBA 2.3 OBV spec. defines a solution for this

- Most ORBs still support only 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

- Until OBV spec is ubiquitous, objects can be passed by value using
  hand-crafted "factories"

Feature Limitations (cont'd)

- Many ORBs do not yet support AMI and/or standard CORBA timeouts

  - However, these capabilities are defined in the OMG Messaging 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 from IONA
  - Visibroker from Inprise
  - BEA Web Logic Enterprise
  - Component Broker from IBM
  - eORB from Vertel, ORB Express from OIS, and HighComm from Highlander/Inprise
  - Open source ORBs → TAO, ORBacus(*), 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/
  - CETUS CORBA Page
    * www.cetus-links.org/oo_corba.html
  - LANL's OMG Page
    * www.acl.lanl.gov/CORBA