PENNSTATE World Campus

SWENG568: Enterprise Integration

Lesson 3: Evolution of Integration Patterns (II): Distributed Objects and Messaging

Introduction (1 of 14)

Learning Objectives

- Understand the need for distributed objects and messaging in enterprise integration
- Understand how distributed objects and messaging can better serve the need of enterprise integration
- Master basic concepts: distributed objects, messaging, middleware, message oriented middleware

By the end of this week, make sure you have completed the readings and activities found in the Lesson 3 Course Schedule.

Shortcomings of RPC (2 of 14)

Shortcomings of RPC

To meet the dynamic business needs, it becomes essential for IT system to interact with each other in a real time fashion. As we learn that the only constant is change these days, we have to have approaches that enable reusability, scalability, and transparency to ensure that interactions can be adapted to the future changes. As we learn in last Lesson, RPC introduces new concepts and features in order to make functionality sharing more computationally powerful and capable, ensuring the enabled mechanism leading to a scalable and reusable manner. These concepts and features include:

- The concept of interface
- The concept of a service provider (i.e., server) at the application level
- The introduction of stub/proxy/skeleton, which shields the programmer from system and network calls
- The concept of marshalling/unmarshalling of arguments for transmission over the network
- The concept of platform independence via the use of external data representation (XDR), which encodes data in a machine-independent format

However, RPC does not well provide all these fundamental supports due to the following shortcomings due to its primitive design and technical limits at the time when it was introduced [1]:

- There is little room for code reuse. As the example discussed in last lesson, the code for marshalling and unmarshalling and the code for network communication are buried inside the client and server applications.
- General speaking, RPC is not programming language independent. For example, if a client is written in C and running on a Unix machine, while a server is written in Visual Basic and running on a Windows machines, RPC can't make them share functionality.
- Typically, RPC creates point-to-point integration architecture, which could result in the egregious complexity of integrated systems in organizations. As the functionality embedded in a server is what a client wants to invoke, the relationship between the client and the server is not peer-to-peer.

To have all application silos integrated across business domains is the ultimate goal of enterprise integration. Thus, the concepts and features introduced in RPC should ideally be able to address the data and functionality sharing issues in a heterogeneous computing environment. Figure 3.1 illustrates two different applications written in different languages and residing on different platforms. As RPC can't solve these challenges, obviously, the question becomes how we can overcome the shortcomings in RPC. What a new kind of solution is needed in order to make the concepts and features more generic and well supported in IT systems, leading to improved reusability, scalability, and transparency of software services?

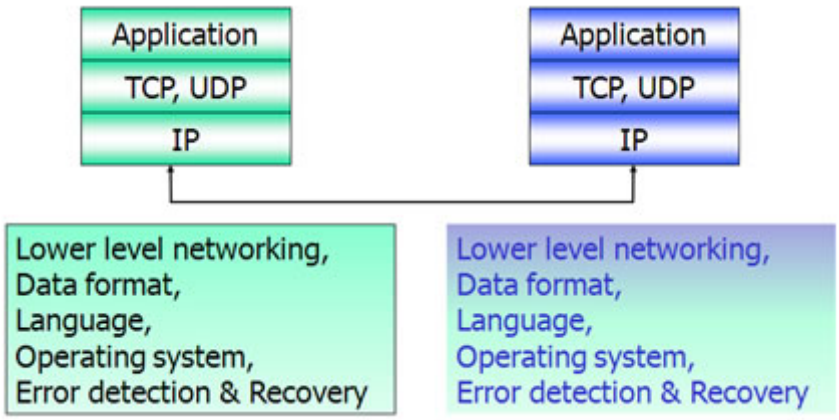

Figure 3.1: A Programmer's View of Implementation Challenges between Heterogeneous Applications

The Need for a New Solution

As more and more applications are essentially distributed in enterprises, a distributed computing environment will be implied in all the discussions from now on.

When the concepts and features introduced in RPC get further developed in today's heterogeneous computing environment, apparently the new solution should overcome the RPC shortcomings by providing the following capabilities, aimed at addressing the data and functionality sharing issues in a more effective and efficient manner:

- code reuse by separating out the code for marshalling and network communication into a standardized support,

- a peer-to-peer rather than point-to-point relationship between applications that should be formed from the above separation
- language-independence, and
- platform independence

What is an Object?

Programming languages evolve in order to overcome discovered shortcomings in a conventional programming language, for instance, C++ over C aimed at finding a better approach to do modularization, avoid the use of error-prone global variables, and enable many other features in software development. Object-oriented design (OOD) and object-oriented programming (OOP) emerged.

A software object is computationally a compilation of attributes (object elements) and behaviors (methods or operations) encapsulating an entity. Other software objects can access the object only by invoking its methods that have been allowed to be externally called. In computer science, an object is an instance of a defined class (Figure 3.2).

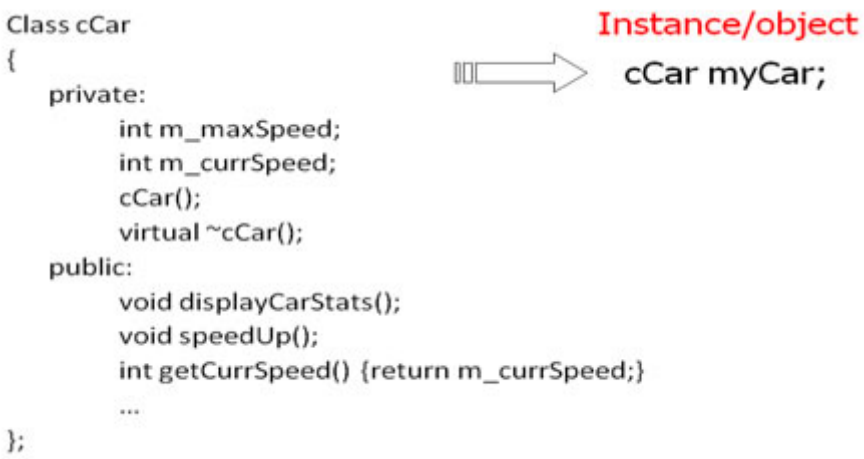

Figure 3.2: An Example of Class Definition

Although there will be some implementation differences from language to language, the following concepts extracted from [2] are typically found in all object-oriented programming languages:

(Click the tabs to learn more)

Class

Defines the abstract characteristics of a thing (object), including the thing's characteristics (its attributes, fields or properties) and the thing's behaviors (the things it can do, or methods, operations or features). One might say that a class is a blueprint or factory that describes the nature of something. For example, the class cCar would consist of traits shared

by all cars, such as current running velocity and maximum velocity (characteristics), and the ability to start, accelerate, de-accelerate, and stop. Classes provide modularity and structure in an object-oriented computer program.

Object

An exemplar pattern of a class. The class cCar defines all possible cars by listing the characteristics and behaviors they can have; the object myCar is one particular car, with particular versions of the characteristics.

Instance

The instance is the actual object created at runtime.

Method

An object's abilities. In language, methods (i.e., functions) are verbs. Within the program, using a method usually affects only one particular object.

Inheritance

Subclasses" are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own.

“Multiple-inheritance” is inheritance from more than one ancestor class, neither of these ancestors being an ancestor of the other. This is not always supported in all programming languages, as it can be hard to implement.

Abstraction

Abstraction is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem.

Abstraction is also achieved through Composition. For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only

how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

Encapsulation

Encapsulation conceals the functional details of a class from objects that send messages to it.

Encapsulation is achieved by specifying which classes may use the members of an object. The result is that each object exposes to any class a certain interface — those members accessible to that class. The reason for encapsulation is to prevent clients of an interface from depending on those parts of the implementation that are likely to change in the future, thereby allowing those changes to be made more easily, that is, without changes to clients.

Members are often specified as **public**, **protected** or **private**, determining whether they are available to all classes, sub-classes or only the defining class.

Polymorphism

Polymorphism allows the programmer to treat derived class members just like their parent class's members. More precisely, Polymorphism in object-oriented programming is the ability of objects belonging to different data types to respond to calls of methods of the same name, each one according to an appropriate type-specific behavior. One method, or an operator such as +, -, or *, can be abstractly applied in many different situations.

Message Passing

The process by which an object sends data to another object to call a method residing in the called object.

Historically, a piece of software has been viewed as a suite of logical modules that takes input data, processes them, and produces output data. By taking advantage of the concepts and features in the Object-orientation computing world [2], OOP evolves as a popular programming model that makes software modules implemented around "objects" rather than "actions" and data rather than application logic. OOD is the process of planning a system of interacting objects for the purpose of solving a software problem in an efficient and effective manner. Rational Rose family is one of the popular tools adopted in many organizations.

The Need for Distributed Objects and Messaging in Enterprise Integration

As time goes, organizations have put significant investments on IT systems. Different IT applications were developed or purchased to meet different needs from time to time, creating a heterogeneous enterprise information system environment. In other words, most of IT systems are silo applications suitable for specific business domain in the organization; they were not designed for enterprise integration. To deliver the right data/information to the right user at the right time is what we need to run a world-class business. Thus, enterprise (application) integration is essential for improving or maintaining business competitiveness. Due to the heterogeneity, enterprise integration becomes an extremely challenging process as it typically involves many different types of distributed applications written in different programming languages and running on different types of platforms.

As discussed earlier, OOD/OOP becomes a very popular software development paradigm, resulting in more and more object-oriented applications deployed in organizations. By leveraging the advantage of *distributed computing*, objects in applications interact with each other using message passing to enable mechanisms for the delivery of the right data/information to the right user at the right time to accomplish some computational task (Figure 3.3). Therefore, the term **distributed objects** refers to software modules that can work together, but reside either in multiple networked computers or in different processes on the same computer. One object sends a message to another object in a remote machine or process to perform some task. The computed results are sent back to the invoking object.

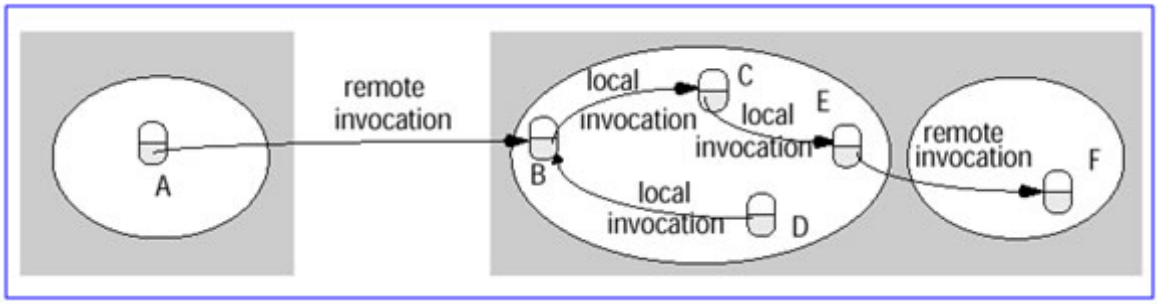

Figure 3.3: A Schematic View of Local and Remote Services

Local and distributed objects differ in many respects. Here are some of them [2]:

- ***Life Cycle****: Creation, migration and deletion of distributed objects is different from local objects*
- ***Reference****: Remote references to distributed objects are more complex than simple pointers to memory addresses*
- ***Request Latency****: A distributed object request is orders of magnitude slower than local method invocation*
- ***Object Activation****: Distributed objects may not always be available to serve an object request at the point of need*
- ***Parallelism****: Distributed objects may be executed in parallel.*

- ***Communication**: There are different communication primitives available for distributed objects requests*
- ***Failure**: Distributed objects have far more points of failure than typical local objects*
- ***Security**: Distribution makes them vulnerable to attack.*

Managing Distributed Objects (6 of 14)

Managing Distributed Objects

When the concepts and features introduced in RPC are applied in OO software components, we have Figure 3.4 to schematically show the interaction between objects residing in different applications. Software components might be developed in a variety of programming languages and applications which hold an assembly of instantiated software objects reside on different platforms across the networks.

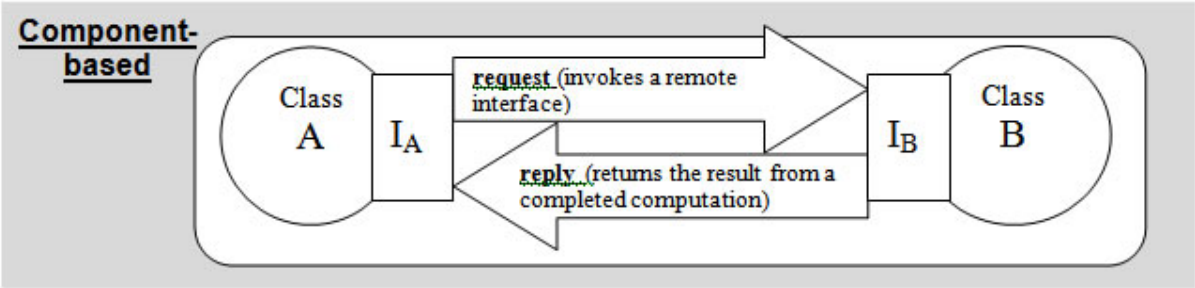

Figure 3.4: A Programmer's View of RPC-type Implementation between Distributed Objects

How Do Two Remote Objects Communicate with Each Other? (7 of 14)

How Do Two Remote Objects Communicate with Each Other?

Two methods, passing by reference and passing by value, are frequently used for one object in an application to communicate with the other object in another application. Figure 3.5 and 3.6 clearly show the difference between those two methods. In case of enterprise integration, passing by reference is widely used in a heterogeneous computing environment.

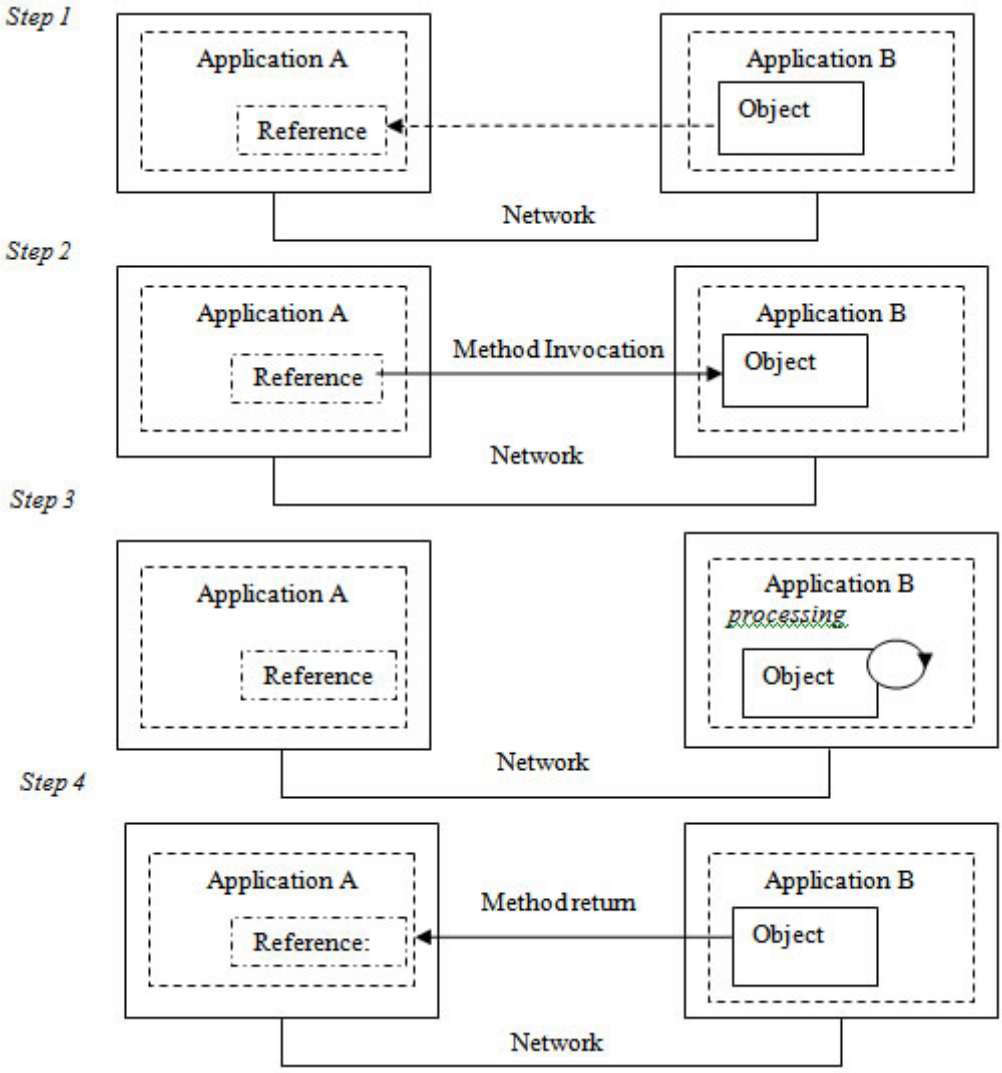

Figure 3.5: Passing by Reference [1]

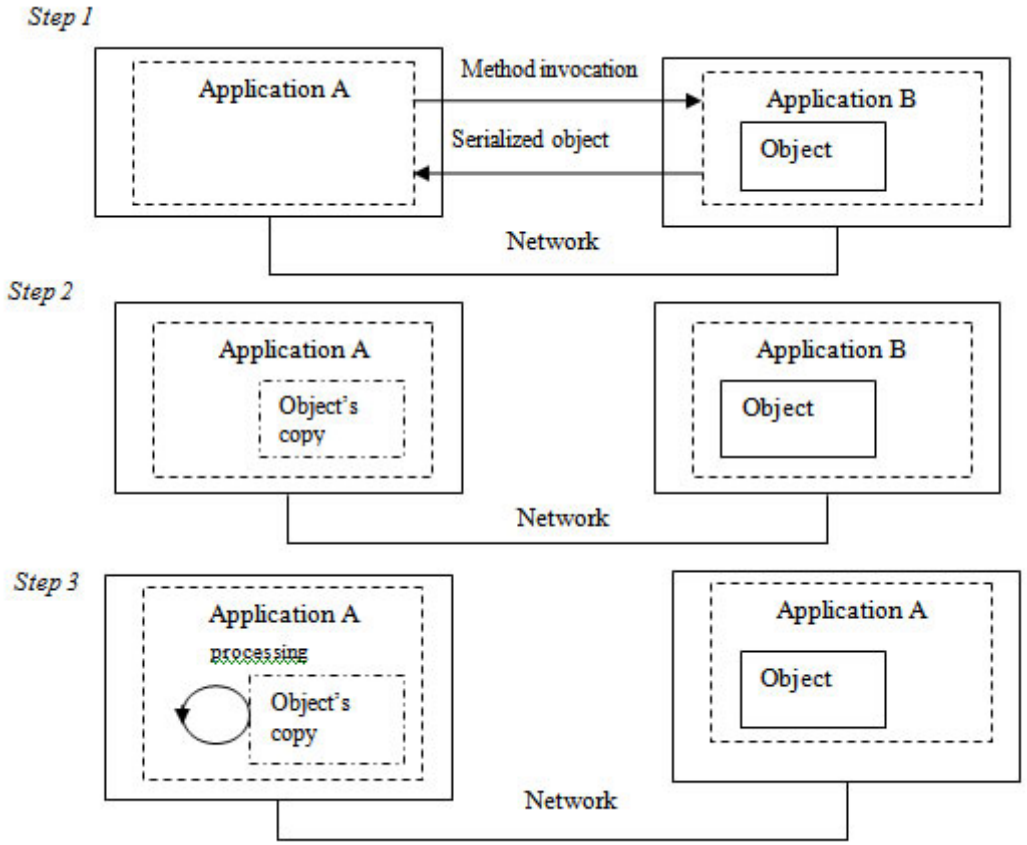

Figure 3.6: Passing by Value [1]

Standardized Common Shared Integration Infrastructure – Middleware Technology

Enabling standardized common shared integration infrastructures has been a dream approach in the computing industry for years. As mentioned in the above discussion, the data and functionality sharing can be done in a more effective and efficient manner in enterprise integration if code reuse to support data marshalling/unmarshalling and network communication programming, a peer-to-peer rather than point-to-point relationship between applications, language-independence, and platform-independence are the supports embedded in such a common shared integration infrastructure. A common shared integration infrastructure is also called middleware.

Middleware [2] is ideally and essentially a computing infrastructure that connects software components or applications in a heterogeneous computing environment, although it might have a scaled-down version used in a specific and homogenous computing setting. This infrastructure consists of a set of services that allows multiple processes running on one or more machines to interact. The technology evolves to provide for greater interoperability in support of the move to coherent distributed architectures, which are used most often to support and simplify complex, distributed applications. Figure 3.7 shows how the supports of code reuse, peer-to-peer architecture, language-independence, and platform independence can be implemented using a middleware.

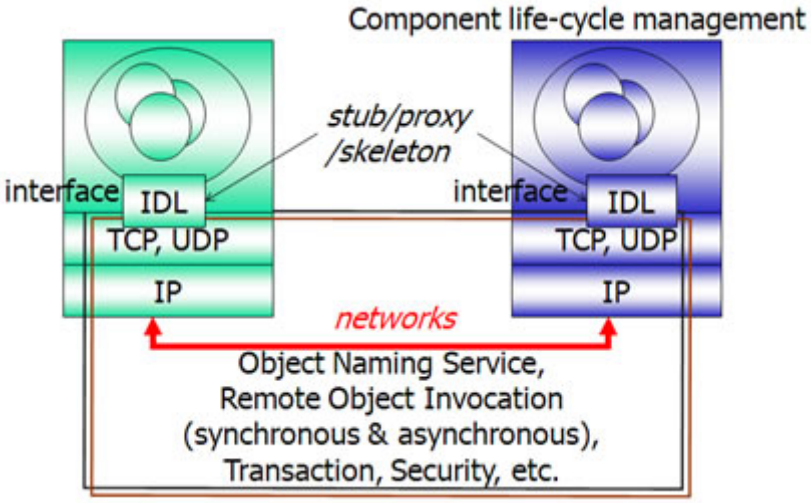

Figure 3.7: Schematic View of a Common Shared Integration Infrastructure

In general, we expect that the middleware would standardize the following supports in managing distributed objects:

- Simplifying software components (or services) distribution by automating
  - Object location & activation
  - Parameter marshaling/unmarshalling
  - Demultiplexing/communication
  - Error handling
- Providing foundation for higher-level services
  - Naming (registration) service
  - Transaction
  - Security
  - Concurrency control
  - Object (service) lifecycle management
- Ensuring better interoperability
  - Language-independence
  - Platform-independence

In the late 90s, three main different kinds of middleware became available in managing distributed objects. The widely applied middleware includes Common Object Request Broker Architecture (CORBA) [2, 4], Microsoft's Distributed Component Object Model (DCOM) [2, 5], and Java's Remote Method Invocation (RMI) [2, 6]. Out of these three, CORBA has been the most popular one used in industry [4].

DCOM was pretty much limited to Microsoft's Windows operating systems. Because of its proprietary technology, Microsoft abandoned its support of DOCM and is focusing on its .Net framework. According to [2], ".NET Remoting is a Microsoft application programming interface (API) for interprocess communication released in 2002 with the 1.0 version of .NET Framework. It is one in a series of Microsoft technologies that began in 1990 with the first version of Object

Linking and Embedding (OLE) for 16-bit Windows. Intermediate steps in the development of these technologies were Component Object Model (COM) released in 1993 and updated in 1995 as COM-95, Distributed Component Object Model (DCOM), released in 1997 (and renamed Active X), and COM+ with its Microsoft Transaction Server (MTS), released in 2000. It is now superseded by Windows Communication Foundation (WCF), which is part of the .NET Framework 3.0." [5]

Although RMI is not constrained by operating systems, it can only be implemented using Java programming language [6]. "Java Remote Method Invocation (Java RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism." [6]

CORBA is widely utilized in industry due to the fact that CORBA has been designed as an open, vendor-independent architecture and infrastructure which different applications rely on to work together over networks. By relying on standard protocols, one CORBA-compliant program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with another CORBA-compliant program from the same or another vendor, on almost any other computer, operating system, programming language, and network [4] (Figure 3.8).

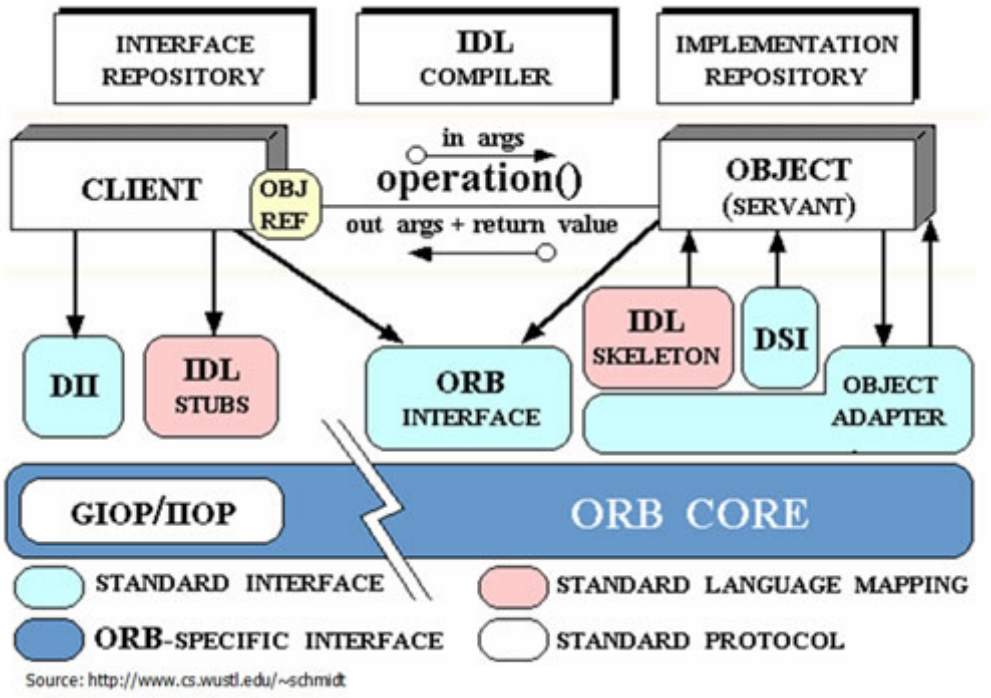

Figure 3.8: Overview of CORBA ORB Architecture [Source: http://www.cs.wustl.edu/~schmidt (http://www.cs.wustl.edu/~schmidt)]

Overview of CORBA

As we discussed earlier several times, **code reuse** to support data marshalling/unmarshalling and network communication programming, a **peer-to-peer** rather than point-to-point relationship between applications, **language-independence**, and **platform-independence** are the core supports necessary for an excellent common shared integration infrastructure. Object Management Group (OMG) aims at enabling the best middleware technology to manage distributed objects in enterprise integration. Three pillars of standard supports enabled to address these needs are [7-11]:

- The Object Request Broker (ORB), defined as a core connectivity support that focuses on standardize and facilitate effective communication between CORBA objects
- Interface Definition Language (IDL), which specifies interfaces between CORBA objects
    - IDL ensures CORBA's language independence because interfaces described in IDL can be theoretically mapped into any programming languages
- GIOP/IIOP Protocols, i.e., General InterORB Protocol and Internet InterORB Protocol. The GIOP specification consists of the following fundamental elements:
- *The Common Data Representation (CDR) definition*. CDR is a transfer syntax mapping OMG IDL data types into a bicanonical low-level representation for "on-the-wire" transfer between ORBs and Inter-ORB bridges (agents).
- *GIOP Message Formats*. GIOP messages are exchanged between agents to facilitate object requests, locate object implementations, and manage communication channels.
- *GIOP Transport Assumptions*. The GIOP specification describes general assumptions made concerning any network transport layer that may be used to transfer GIOP messages. The specification also describes how connections may be managed, and constraints on GIOP message ordering.
- The IIOP specification adds the following element to the GIOP specification:
    - *Internet IOP Message Transport*. The IIOP specification describes how agents open TCP/IP connections and use them to transfer GIOP messages.
    - The IIOP is not a separate specification; it is a specialization, or mapping, of the GIOP to a specific transport (TCP/IP). The GIOP specification (without the transport-specific IIOP element) may be considered as a separate conformance point for future mappings to other transport layers.

Prof. Douglas C. Schmidt (http://www.cs.wustl.edu/~schmidt/) maintains an excellent website on distributed object computing with CORBA middleware. Here uses his descriptions to overview these components in Figure 3.8 [7]:

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

distributed programming by decoupling the client from the details of the method invocations. This makes client requests appear to be local procedure calls. When a client invokes an operation, the ORB is responsible for finding the object implementation, transparently activating it if necessary, delivering the request to the object, and returning any response to the caller.

- ***ORB Interface*** *-- An ORB is a logical entity that may be implemented in various ways (such as one or more processes or a set of libraries). To decouple applications from implementation details, the CORBA specification defines an abstract interface for an ORB. This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.*
- ***CORBA IDL stubs and skeletons*** *-- CORBA IDL stubs and skeletons serve as the "glue" between the client and server applications, respectively, and the ORB. The transformation between CORBA IDL definitions and the target programming language is automated by a CORBA IDL compiler. The use of a compiler reduces the potential for inconsistencies between client stubs and server skeletons and increases opportunities for automated compiler optimizations.*
- ***Dynamic Invocation Interface (DII)*** *-- This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in. Unlike IDL stubs (which only allow RPC-style requests), the DII also allows clients to make non-blocking deferred synchronous (separate send and receive operations) and one way (send-only) calls.*
- ***Dynamic Skeleton Interface (DSI)*** *-- This is the server side's analogue to the client side's DII. The DSI allows an ORB to deliver requests to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. The client making the request has no idea whether the implementation is using the type-specific IDL skeletons or is using the dynamic skeletons.*
- ***Object Adapter*** *-- This assists the ORB with delivering requests to the object and with activating the object. More importantly, an object adapter associates object implementations with the ORB. Object adapters can be specialized to provide support for certain object implementation styles (such as OODB object adapters for persistence and library object adapters for non-remote objects).*

Highlight 1: Interface Definition Language

IDL is just a declarative language, independent of programming language. CORBA uses language mapping which determines how IDL features are mapped to the facilities of a given programming languages. OMG has standardized mappings from IDL to C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript [4].

This separation of interface from implementation is the essence of CORBA as it enables interoperability between objects residing in applications written in different programming languages. Although the interface to each object is defined very strictly, the implementation of an object - its running code, and its data - is hidden from the rest of the system. Clients access remote objects only through their published interface, invoking only those operations/methods thatthat the object exposes through its IDL interface, with only those parameters (input and output) that are included in the invocation [4].

Highlight 2: Invocation Models

Figure 3.9 illustrates how two objects could communication through remote object invocations. Instead of client-server relationship between two applications, this support clearly shows how CORBA moves away from client/server to peer-to-peer relationship for communication within distributed applications.

CORBA Examples

Here shows only a very simple example extracted from JacORB (http://www.jacorb.org/download.html (http://www.jacorb.org/download.html)). There are many other free tools allowing you to try this integration approach [7, 8, 10, and 12]. Please download one based on your background (or interest), and then try to implement some simples examples. (You will have a lot of fun if you like programming although you might have to spend a lot of time to make it right).

The following steps are usually used to implement a typical CORBA-based integration [13]:

Listing 3.1 Step 1: Defining IDL

```
//grid.idl
// IDL defintion of a 2–D grid:

module demo
{
module grid
{
    interface MyServer
    {
        typedef fixed <5,2> fixedT;

        readonly attribute short height; // height of the grid
        readonly attribute short width; // width of the grid

        // set the element (n,m) of the grid, to value:
        void set(in short n, in short m, in fixedT value);

        // return element [n,m] of the grid:
        fixedT get(in short n, in short m);

        exeption MyException
        {
            string why;
        };

        short opWithException() raises ( myException );
    };

 };
 };
```

Listing 3.2 Step 3: Writing Implementation Logic

```
package demo. grid;

/**
*A very simple implementation of a 2-D grid
*/

import demo.grid.MyServerPackage.MyException;

public class gridImpl
    extends MyServerPOA
{
    protected short height = 31;
    protected short width = 14;
    protected java.math.BigDecimal[] [] mygrid;

    public gridImpl ()
    {
        mygrid new java.math.BigDecimal[heightj [width]:
        for( short h = 0; h < height; h++ )
        {
            for(Short w = 0; w < width; w++ )
            {
                mygrid[h][w] = new java.math.BigÐecima1("O.21");
            }
        }
    }

    public java.math.BigDecimal get(short n, short m)
    {
        if(( n <= height ) && ( m <= width ))
            return mygrid[n][m];
        else
            return new java.math.BigDecimal("O.O1");
    }

    public short height()
    {
        return height;
    }

    public void set (short n, short m, java.math.BigDecimal value)
    {
        if( ( n <= height ) && ( m <= width ) )
        mygrid[n] [m) value;
    }

    public short width()
    {
        return width;
    }

    public short opWithException()
        throws demo.grid.MyServerPackage.MyException
```

```
    {
        throw new demo.grid.MyServerPackage.MyException("This is only a test
        exception, no harm done :–)");
    }
}
```

Listing 3.3 Step 4: Writing Your Server Main

```
package demo.grid;

import java.io.;
import org.omg.CosNaming.*;

public class Server
{
    public static void main( String[] args )
    {
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
        try
        {
            org.omg.PortableServer..POA poa=

            org. omg . PortableServer . POAHelper . narrow (orb. resolve_initial_references ("Ro
            OA") );

            poa.the_POAManager().activate();

            org.omg.CORBA.Object o =
                poa.servant_to_reference(new gridImpl());

            if( args.length == 1 )
            {
                // write the object reference to args[0]

                PrintWriter ps =
                    new PrintWriter(new FileoutputStream(new File( args[O] )));
                ps.println( orb.object_to_string( o ) );
                ps. close();
            }
            else
            {
                // use the naming service

                NamingContextExt nc =

                    NamingContextExtHelper.narrow(orb.resolve initial references("NameService"))
                nc.bind( nc.to_name("grid.example"), o);
            }

                orb.run ;
        }
        catch ( Exception e )
        {
            e.printStackTrace();
        }
    }
}
```

Listing 3.4 Step 5: Writing Your Client Invocation

```
package demo. grid;

import org.orng.CosNaming.*;
import java.io.*;
public class Client
{
    public static void main(String args[])
    {
    try
    {
        MyServer grid;

        org.omg.CORBA.ORB orb = org.orag.CORBA.ORB.init(args, null);

        if( args.length == 1 )
        {
            // args[O] is an IOR-string
            grid = MyServerHelper.narrow(orb.string_to_object(args(O]));
        }
        else
        {
            NamingContextExt nc =
                NamingContextExtflelper.narrow(
                    orb.resolve_initial_references( "NameService" ));

                nc.to_name ("grid.example");

                org.omg.CORBA.Object o =
                    nc.resolve(nc.to_name("grid.example"));

            grid MyServerHelper.narrow(o);
        }

        short x = grid.height();
        System.out.println("Height = " + X);

        short y = grid.width();
        System.out.println("Width = " + y);

        X -= 1;
        y -= 1;

        System.out.println("O1d value at (" + x + "," + y +"): " +
            grid.get(x,y));

        System.out.println("Setting (" + x + "," + y +") to 470.11);

        grid.set( x, y, new java.math.BigDecimal("470.11"));

        System.out.println("New value at (" + x + "," + y +"): " +
            grid.get( xy));

        try
```

```
        {
            grid.opWithException();
        }
        catch (demo.grid.MyServerPackage.MyException ex)
        {
            System.out.println("MyException, reason: " + ex.why);
        }

        orb. shutdown (true);
        System.out.println("done. );

    }
    catch (Exception e);
    {
        e.printStackTrace();
    }
  }
}
```

Messaging Systems for Enterprise Integration (11 of 14)

Messaging Systems for Enterprise Integration

Moving away from procedural languages and related traditional programming models into the realm of OOD/OOP and standard approaches to managing distributed objects over networks becomes obviously advantageous. As enterprise IT systems continue to increase functionality and complexity, two other important functionality sharing issues unresolved yet in distributed objects require solutions. These two issues are:

- Synchronous interaction between integrated applications leads to strong coupling, resulting in a lack of necessary scalability in the long run.
- RPC- and ORB-based communication is not reliable as there is no guarantee that the innovation and return data will be delivered to the intended destination.

Messaging has been introduced to promote decoupling between applications; the asynchronous messaging approach essentially can address these two issues very well. Because of the introduction of this decoupling mechanism, high transaction volumes of interactions to meet the needs of data and functionality sharing in real time between distributed applications can be realized [14].

Overview of Asynchronous Messaging (12 of 14)

Overview of Asynchronous Messaging

In asynchronous messaging, two popular programming models, point-to-point and publish-to-subscribe, are used. In the point-to-point messaging model (Figure 3.10), a message is sent to a specified queue from a sender (or client), then it is up to the receiver (or server) to retrieve the message stored in the queue. In the publish-to-subscribe model (Figure 3.11), multiple subscribers (or clients) subscribe an interested topic and a publisher (or server) publishes a message to the topic. When these subscribers are connected to the topic, the message will be delivered to connected subscribers.

Asynchronous messaging allows the client application to continue to perform its computation after it sends out a message. The server will execute the requested computational service and could send the result back to the client using the same model but a reverse client/server relationship. Apparently, the model decouples the relationship between applications that require data/functionality sharing, leading to a much more scalable solution.

Point-to-Point Messaging

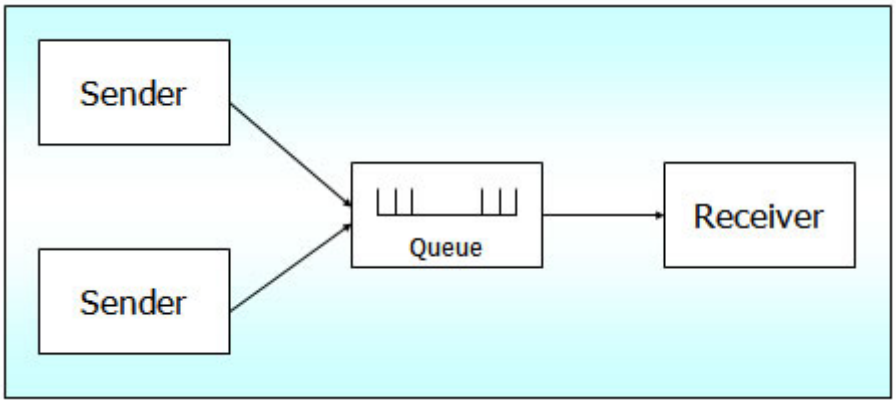

Figure 3.10: Point-to-Point Messaging

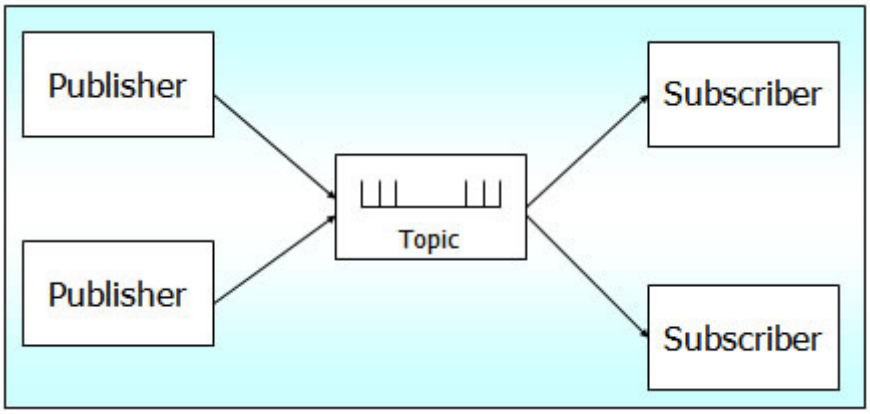

Figure 3.11: Publish-to-Subscribe Messaging

As an implementation example, Java Message Service (JMS) [15] defines the concept of a destination as the target for a message. This destination can be a Queue or a Topic. **Queues** are used for point-to-point messaging; **Topics** are used for publish-subscribe messaging. JMS also defines the concept of **connection** factories that create the connections to a JMS provider. Point-to-point and publish-subscribe messaging are implemented and defined by separate interfaces so that a provider doesn't have to support both.

In JMS publish-subscribe messaging, each message may have multiple consumers. There is also a timing dependency between publishers and subscribers, because a client that subscribes to a topic can consume only messages published after the client has created a subscription, and the

subscriber must continue to be active in order for it to consume messages. Durable subscriptions can receive messages sent while the subscribers are not active. The messaging approach provides a much more reliable mechanism for enterprise integration [14, 15].

References

[1] Roshen, W. 2009. SOA-based Enterprise Integration: A Step-by-Step Guide to Services-based Application Integration. McGraw-Hill, New York, USA.

[2] Wikipedia:

- History of Programming Languages: http://en.wikipedia.org/wiki/History_of_programming_languages,
- Timeline of Programming Languages: http://en.wikipedia.org/wiki/Timeline_of_programming_languages,
- Object-oriented Design: http://en.wikipedia.org/wiki/Object-oriented_design,
- Object-oriented Programming: http://en.wikipedia.org/wiki/Object-oriented_programming,
- Peer-to-peer: http://en.wikipedia.org/wiki/Peer-to-peer,
- Distributed Object: http://en.wikipedia.org/wiki/Distributed_object,
- Distributed Computing: http://en.wikipedia.org/wiki/Distributed_computing,
- Middleware: http://en.wikipedia.org/wiki/Middleware,
- DCOM: http://en.wikipedia.org/wiki/Distributed_Component_Object_Model,
- .Net Remoting: http://en.wikipedia.org/wiki/.NET_Remoting,
- Windows Communication Foundation: http://en.wikipedia.org/wiki/Windows_Communication_Foundation, Java RMI: http://en.wikipedia.org/wiki/Java_remote_method_invocation,
- GIOP/IIOP: http://en.wikipedia.org/wiki/General_Inter-ORB_Protocol,
- Message Broker: http://en.wikipedia.org/wiki/Message_broker,
- Message-oriented Middleware: http://en.wikipedia.org/wiki/Message-oriented_middleware,

[3] IBM Rational: http://www-01.ibm.com/software/rational/

[4] OMG: http://www.omg.org/

[5] Microsoft Solutions to Distributed Objects:

- DCOM: http://msdn.microsoft.com/en-us/library/ms878122.aspx
- .Net Remoting: http://msdn.microsoft.com/en-us/library/kwdt6w2k(VS.71).aspx
- Windows Communication Foundation: http://msdn.microsoft.com/en-us/netframework/aa663324.aspx. http://en.wikipedia.org/wiki/Windows_Communication_Foundation

[6] RMI: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[7] Distributed Object Computing with CORBA Middleware: http://www.cs.wustl.edu/~schmidt/corba.html

[8] A Simple C++ Client/Server in CORBA: http://www.codeproject.com/KB/IP/corba.aspx

[9] CORBA Explained Simply: http://www.ciaranmchale.com/corba-explained-simply/

[10] Orbacus Package: http://web.progress.com/en/orbacus/downloads.html

[11] GIOP/IIOP: http://www2.informatik.hu-berlin.de/~obecker/Lehre/SS2001/CORBA/specs/01-02-51.pdf

[12] JacORB: http://www.jacorb.org/download.html.

[13] JacORB Programming Guide: http://www.jacorb.org/documentation.html.

[14] Patterns and Best Practices for Enterprise Integration: http://www.eaipatterns.com/index.html.

[15] Java Message Service: http://java.sun.com/products/jms/
