CHAPTER 1

Intersystem Communications

Intersystem Communications:

The ability of two or more computer systems to share input, output, and storage devices, and to send messages to each other by means of shared input and output channels or by channels that directly connect central processors. It is essential for building complex, distributed applications that leverage the strengths of various components.

Architectures for Integrating Systems

Integrating systems refers to the process by which multiple individual subsystems or sub-components are combined into one all-encompassing larger system thereby allowing the subsystems to function together. In most organizations that use system integration there is a need to improve efficiency and thereby productivity and quality of their operations.

Architectures for integrating systems provide a blueprint for how different systems should interact. They define the communication protocols, data formats, and overall structure of the integration. Here are some key architectures:

i. Point-to-Point Integration:

This is the simplest form of integration, where two systems are directly connected. It's suitable for small-scale integrations with well-defined interfaces. However, it can become complex and difficult to maintain as the number of integrated systems grows. It creates a "spaghetti" architecture, where systems are tightly coupled.

ii. Message-Oriented Middleware (MOM):

MOM uses message queues or topics to decouple systems. Systems communicate by sending and receiving messages, without needing to know the details of each other's implementation. This provides asynchronous communication, improved reliability, and scalability.

Examples: Apache Kafka, RabbitMQ.

iii. Enterprise Service Bus (ESB):

An ESB acts as a central hub for connecting different systems. It provides services such as message routing, transformation, and protocol conversion. ESBs are often used in complex enterprise environments with diverse systems. ESBs can become a central point of failure, and can become overly complex.

iv. Service-Oriented Architecture (SOA):

This is a design paradigm that structures an application as a collection of loosely coupled services. Services are self-contained, reusable, and communicate through well-defined interfaces. SOA promotes interoperability, flexibility, and scalability.

1.1 Middleware Components and Middleware

A ***middleware component*** is software that connects two other separate applications.

For example, there are a number of middleware products that link a database system to a Web server. This allows users to request data from the database using forms displayed on a Web browser, and it enables the Web server to return dynamic Web pages based on the user's requests and profile.

The term ***middleware*** is used to describe separate products that serve as ***the glue*** between two applications. It is, therefore, distinct from import and export features that may be built into one of the applications. Middleware is sometimes called ***plumbing*** because it connects two sides of an application and passes data between them.

To achieve this, Some Distributed object technologies such as Distributed Component Object Model (DCOM), Common Object Request Broker Architecture (CORBA) and Java Remote Method Invocation (Java RMI) have been selected according to the following criteria:

i) Portability
ii) Platform independence
iii) Wide use
iv) Database independence

1.2 Distributed Object Technologies:

A ***distributed system*** is defined as a system in which hardware or software components located at networked computers communicate and co-ordinate their actions only by passing messages.

Computing devices may be connected to a wide range of networks as for instance the Internet, mobile phone networks, corporate networks, home networks or to combinations of these.

Modern program systems like Internet-based and enterprise applications offer multi-tiered, component-based architectures that incorporate middleware for distributing components across heterogeneous platforms. The platforms range from mobile devices like personal digital assistants (PDA), laptops and mobile phones; ubiquitous devices like televisions, refrigerators and cars; to different types of computers like mainframes and PCs.

The three most dominating distributed object technologies or middleware are CORBA, DCOM and Java/RMI. These are extensions of traditional object-oriented systems by allowing objects to be distributed across a heterogeneous network. The objects may reside in their own address space outside of an application or on a different computer than the application and still be referenced as being part of the application.

All three distributed object technologies are based on a client/server approach implemented as network calls operating at the level of bits and bytes transported on network protocols like TCP/IP. In order to avoid the hard and error prone implementation of network calls directly in the client and server {objects}, the distributed technology standards address the complex networking interactions by abstract and hide the networking issues and instead let the programmer concentrate on programming the business logic.

The basic idea behind network abstractions like RPC (Remote Procedure Call) is to replace the local (server) and remote (client) end by stubs. This makes it possible for both client and server to strictly use local calling conventions and thereby be unaware of calling a remote implementation or being called remotely. To accomplish this the client call is handled by a client stub (proxy) that marshals the parameters and sends them by invoking a wire protocol like IIOP (*Internet Inter ORB Protocol)*, ORPC (Object Remote Procedure call) or JRMP(Java RMI Protocol), to the remote end where another stub receives the parameters, unmarshals and calls the true server. The marshaling and unmarshaling actions are responsible for converting data values from their local representation to a network format and on to the remote representation. Format differences like byte ordering and number representations are bridged this way.

The object services offered by all three approaches are defined through interfaces. The interface is for all defined as a collection of named operations, each with a defined signature and optionally a return type. The interface serves as a contract between the server and client.

Distributed Object Technologies or Distributed Object Computing (DOC) integrates heterogeneous applications. DOC extends an object-oriented system by providing a means to distribute objects across a network, allowing each component to interoperate as a unified whole. Objects look "local" to applications, even though they are distributed to different computers throughout a network.

Distributed Object Technologies are useful in many sever situations. A distributed processing is also widely used with a parallel processing approach such as clustering, especially in multi-tier environments.

Remoting:

Many Distributed Object Technologies such as DCOM, CORBA model and Java RMI are used to invoke a remote method, also known as ***remoting.***

To invoke a remote method,

- **(i)** The client makes a call to the client-side proxy or stub.
- **(ii)** The client-side proxy packs the call parameters into a request message and invokes a wire protocol to ship the message to the server.
- **(iii)** At the server side, the wire protocol delivers the message to the server-side stub.
- **(iv)** The server-side stub then unpacks the message and calls the actual method on the object.

In both CORBA and Java RMI, *the server stub is called the skeleton and client stub is called the stub or proxy*. In DCOM, *the client stub is called the proxy and the server stub is called the stub*.

CORBA

CORBA depends on an Object Request Broker (ORB), a central bus over which CORBA objects interact transparently. CORBA uses Internet Inter-ORB Protocol (II-ORB) for remoting objects.
To request a service, A CORBA client acquires an object reference to a CORBA server object. The client then makes method calls on the object reference as if the CORBA server object resided in the client's address space. The ORB finds a CORBA object's implementation, prepares it to receive

requests, communicates the requests, and carries the replies back to the client. CORBA can be used on a range of operating system platforms, from hand-held devices to mainframes.

DCOM

DCOM is as an extension of the *Component Object Model (COM)*, a Microsoft framework that supports program component objects. DCOM supports remoting objects through a protocol named ***Object Remote Procedure Call (ORPC)***. ORPC is a layer that interacts with COM's run- time services. A DCOM server is a body of code capable of serving up particular objects at run- time. Each DCOM server object supports multiple interfaces, each of which represent a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a ***pointer-to-server-object interface***. The client object calls into the server object's exposed methods through the interface pointer, as if the server object resided in the client's address space.

Java RMI

RMI is the Java version generally known as a ***remote procedure call (RPC)***, with the added ability to pass objects along with the request. The objects can include information that change the service performed in the remote computer. This property of RMI is often called ***"moving behavior."***

Java RMI relies on the ***Java Remote Method Protocol and on Java Object Serialization***, which allows objects to be transmitted as a stream. Since Java Object Serialization is specific to Java, both the Java RMI server object and the client object have to be written in Java. Java RMI allows client/server applications to invoke methods across a distributed network of servers running the *Java Virtual Machine*.

Although RMI is considered by many to be weaker than CORBA and DCOM, it offers such unique features as *distributed automatic object management* and has the ability to *pass objects between machines*. The naming mechanism, ***RMIRegistry***, runs on the server machine and holds information about available server objects.

A Java RMI client acquires a reference to a Java RMI server object by looking up a server object reference and invoking methods on the server object, as if the Java RMI server object resided on the client. These server objects are named using ***Universal Resource Locators (URL)***. A client acquires a reference by specifying the server object's URL, just as you would specify the URL to an HTML page.

While CORBA, DCOM, and Java RMI all provide similar mechanisms for transparently accessing remote distributed objects,

- DCOM is a proprietary solution that works best in Microsoft environments. For an organization that has adopted a Microsoft-centered strategy, DCOM is an excellent choice. However, if any other operating systems are required in the application architecture, DCOM is probably not the correct solution.
- Because of its easy-to-use native-JAVA model, RMI is the simplest and fastest way to implement distributed object architecture. It's a good choice for small applications implemented completely in Java. Since RMI's native-transport protocol, JRMP, can only communicate with other Java RMI objects, it's not a good choice for heterogeneous applications.
- CORBA and DCOM are similar in capability, but DCOM doesn't yet support operating system interoperability, which may discount it as a single solution. At the moment, CORBA is the logical choice for building enterprise wide, open-architecture, distributed object applications.

1.3 DCOM Architecture

DCOM – Distributed Component Object Model, is a standard developed by Microsoft and it is a distributed extension of the COM standard [COM] COM to fit into network environments. The COM standard is based on the development of compound document technology to integrate document parts (for instance spread-sheets, pictures, presentations, word processors etc.) created by different Windows applications. COM is the world most widely used component software model and dominates the desktop market. Fig. 1.1 shows the COM/DCOM architecture.

A client that wants to access a server object calls a COM ***runtime through COM interfaces***, which checks the client's permissions by invoking a ***security provider***. If the client has the appropriate permissions, the COM runtime invokes the ***Distributed Computing Environment Remote Procedure Call (DCE RPC)*** that uses the underlying protocol stack as one of communication methods. Before invoking a DCE RPC, a client side COM runtime performs a **marshalling through a proxy** that converts parameters into the form that could be transferred over the designated protocol.

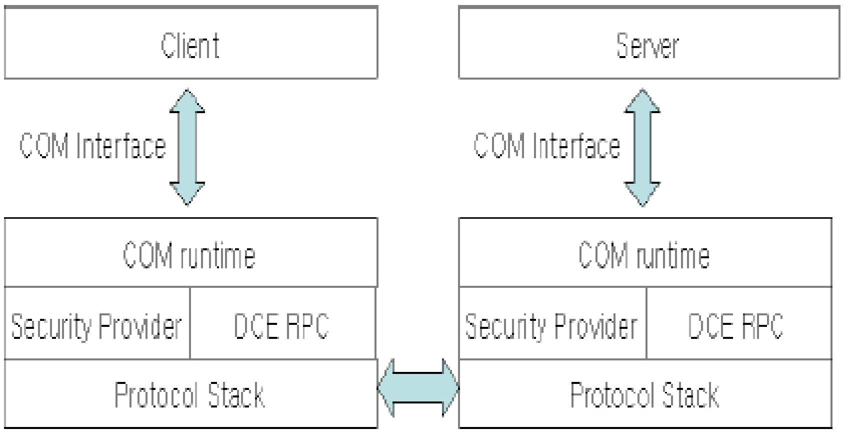

Figure 1.1 COM/DCOM architecture

The protocol stack on the server side receives the request, delivers it to the COM runtime, which performs an ***unmarshalling through a stub*** that converts network packets into parameters. The COM runtime activates the ***server object that calls the object method and processes the request***. A server object has the **3 types such as an *inproc*, a *local*, and a *service* type**. An inproc type takes the form of a *Dynamic Linking Library (DLL)*. An inproc type is the fastest type because it is loaded directly into a client process when activated. A local type takes an *executable form*. When activated, it is executed as a separate process. Lastly, a service type is also similar to the local type in that both of them take executable forms, but different from the local type in that the service type is always resident in a memory, and receives a request from a client. In order to export the methods of a server object, a server object should describe itself by using an ***Interface Description Language (IDL)*.**

For methods that are commonly used in the COM, COM previously creates a set of common interfaces such as *IUnknown*, *IDispatch*, and *IClassFactory*. Although Windows operating system incorporates it, both COM and DCOM are supported only by limited operating systems, and they can't be used in most of web environments due to protection policies.

Benefits of DCOM

- Large User Base and Component Market
- Binary Software Integration
  - Large-scale software reuse (no source code)
  - Cross-language software reuse
- On-line software update

    - Allows updating a component in an application without recompilation, re-linking or even restarting
- Multiple interfaces per object
- Wide selection of programming tools available, most of which provide automation of standard code

1.4 CORBA Architecture

CORBA – Common Object Request Broker Architecture, is an open distributed object computing infrastructure standardized by the Object Management Group (OMG) and is a specification based on technologies proposed (and partly provided) by the software industry [CORBA]. CORBA is the most used middleware standard in the non-Windows market. The OMG was founded in 1989 to promote the adoption of object-oriented technology and reusable software components. Fig. 1.2 shows the CORBA architecture.

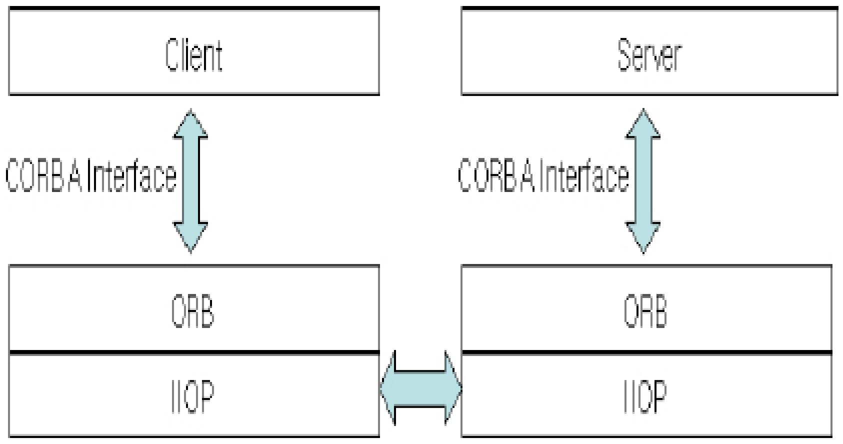

Figure 1.2. CORBA architecture

A client that wants to access a server object should first bind it through a CORBA interface. The client makes a bind request to a local ***Object Request Broker (ORB) through CORBA interfaces***. If the object exists at the local machine, the ORB activates locally the object that processes the request. Otherwise, the ORB makes a request to the other ORBs connected by the network using the ***Internet Inter ORB Protocol (IIOP) over TCP/IP***. After binding the appropriate object, the client side ORB sends the requested method call to the server side ORB. Before sending and receiving parameters, a *stub* performs marshalling and a *skeleton* performs unmarshalling at the CORBA. CORBA has the 2 server types such as a ***Basic Object Adapter (BOA)*** **and a** ***Portable Object Adapter (POA).*** A BOA type was the early model, and its ambiguity caused many different implementations at an early stage. A POA type as developed to solve this incompatibility problem, and came to be a standard. A CORBA also has the ***IDL that is similar to the COM IDL.***

Although CORBA is supported by various operating systems, its users should also choose their own CORBA vendors because various vendors implementing CORBA specification exist, and they can't be used in most of web environments due to protection policies.

Benefits of CORBA

- Programming-language independent interface
- Legacy integration
- Rich distributed object infrastructure
- Location transparency
- Network transparency
- Direct object communication
- Dynamic Invocation Interface

1.5 Java/RMI – Java/Remote Method Invocation

Java/RMI – Java/Remote Method Invocation, is a standard developed by JavaSoft. Java has grown from a programming language to three basic and completely compatible platforms; J2SE (Java 2 Standard Edition), J2EE (Java 2 Enterprise Edition) and J2ME (Java 2 Micro Edition). J2SE is the foundational programming language and tool-set for coding and component development. J2EE supplements J2SE and is a set of technologies and components for enterprise and Internet development. J2ME is used for creating software for embedded, mobile, consumer and other small devices like Personal Digital Assistants (PDA) and mobile phones.

RMI supports remote objects by running on a protocol called the *Java Remote Method Protocol (JRMP).* Object serialisation is heavily used to marshal and unmarshal objects as streams. Both client and server have to be written in Java to be able to use the object serialisation. The Java server object defines interfaces that can be used to access the object outside the current Java virtual machine (JVM) from another JVM on for instance a different machine. A RMI registry on the server machine holds information of the available server objects and provides a naming service for RMI. A client acquires a server object reference through the RMI registry on the server and invokes methods on the server object as if the object resided in the client address space. The server objects are named using URLs and the client acquires the server object reference by specifying the URL.

When a Java/RMI client requests a service from the Java/RMI server, it does the following [Java/RMI]:

(i) initiates a connection with the remote JVM containing the remote object,

(ii) marshals the parameters to the remote JVM,

(iii) waits for the result of the method invocation,

(iv) unmarshals the return value or exception returned, and

(v) returns the value to the caller.

By using serialization of the objects, both data and code can be passed between a server and a client – this allows different instances of an object to run on both client and server machines. To insure that code is downloaded or uploaded safely, RMI provides extra security.

To declare remote access to server objects in Java, every server object must implement the java.rmi.Remote interface. java.rmi.server.RemoteObject and its subclasses, java.rmi.server.RemoteServer, java.rmi.server.UnicastRemoteObject and java.rmi.activation.Activatable provide RMI server functions. The class java.rmi.server.RemoteObject provides implementations for the java.lang.Object methods, hashCode, equals, and toString that are sensible for remote objects. The classes UnicastRemoteObject and Activatable provide the methods needed to create remote objects and make them available to remote clients. The subclasses identify the semantics of the remote reference, for example whether the server is a simple remote object or is an activatable remote object (one that executes when invoked) [Java/RMI]. The java.rmi.server.UnicastRemoteObject class defines a singleton (unicast) remote object whose references are valid only while the server process is alive. The class java.rmi.activation.Activatable is an abstract class that defines an activatable remote object that starts executing when its remote methods are invoked and can shut itself down when necessary [Java/RMI]. Java/RMI can be used on a diversity of platforms and operating systems as long as there is a JVM implementation on the platform.

Benefits of RMI:

- Support seamless remote invocation on objects in different virtual machines
- Support callbacks from servers to applets
- Integrate the distributed object model into the Java language in a natural way while retaining most of the Java language's object semantics
- Make differences between the distributed object model and local Java object model apparent
- Make writing reliable distributed applications as simple as possible

- Preserve the safety provided by the Java runtime environment
- Single language
- Free

1.6 Architecture comparison

The following table 1.1 shows the similarities and dissimilarities among three architectures

Table 1.1 Architecture comparison (CORBA, DCOM, Java/RMI)

	CORBA	**DCOM**	**Java/RMI**
Similarity1: Component based methodology	To maximize reusability and rapid application development, component based approach is applied. Component based approach heavily depends on interface definition. OMG IDL, COM IDL, and so called Java IDL (simply mapping for Java to the OMG IDL) are used.		
Similarity2: Distributed Network	Components can be distributed over network.		
Similarity3: Naming and locating service	All services provide some sort of registry or repository to help locate the corresponding service.		
Dissimilarity1: Vendor and players	Multiple vendor support	Microsoft with some third parties	Sun microsystem with other vendors
Dissimilarity2: Language	A variety of languages	Some languages such as VB/ VC++ / MS-Java	Java
Dissimilarity3: Platform	Platform independent with specification	Win32	Platform independent by JVM