UNIT - II

Communication in Distributed System.

System Model:

Fundamental models are concerned with a more formal description of the properties that are common in all of the architectural models.

Architectural models:

An architectural model of a distributed system first simplifies and abstracts the functions of the individual components of a distributed system and then it considers:

- the placement of the components across a network of computers
- the interrelationships between the components

This classification of processes identifies the responsibilities of each and hence helps us to assess their workloads and to determine the impact of failures in each of them.

Software Layers:

The term software architecture referred originally to the structuring of software as
layers or modules in a single computer and more recently in terms of services offered and requested between processes located in the same or different computers. This process and service-oriented view can be expressed in terms of service layer.

![Service Layer Diagram]

Platform:

These low-level layers provide services to the layers above them, which are implemented independently in each computer, bringing the system's programming interface up to a level that facilitates communication and coordination between processes.

Middleware:

Middleware is represented by processes or objects in a set of computers that interact with each other to implement communication and
Resource sharing support for distributed systems.

They include:

- CORBA
- Java RMI
- Web services
- RCOM
- RM-ODP

System architecture:

The division of responsibilities between system components and the placement of the components on computers in the network is perhaps the most evident aspect of distributed system design.

Client-server:

Server may in turn be clients of other servers. For example, a web server is often a client of a local file server that manages the files on which the web pages are stored.
Peer-to-Peer!

In this architecture, all of the processes involved in a task or activity play similar roles, interacting cooperatively as peers without any distinction between client and server processes or the computers that they run on.

Variations:

Several variations can be derived from the consideration of the following factors:

- The use of multiple servers and caches to increase performance and resilience.
- The use of mobile code and mobile agent
Users need for low cost computers with limited hardware resources that are simple to manage.

The requirement to add and remove mobile devices in a convenient manner.

Interfaces and objects:

The set of functions are available for invocation in a process is specified by one or more interface definitions.

Many objects can be encapsulated in server processes and references to them are passed to other processes so that their methods can be accessed by remote invocation.

This is the approach adopted by CORBA and by Java with its Remote Method Invocation (RMI) mechanism.
Fundamental models:

In this model, based on the fundamental properties that allow us to be more specific about their characteristics and the failures and security risks they might exhibit.

The aspects of distributed systems that we wish to capture in our fundamental models are intended to help us to discuss and reason about:

- Interaction
- Failures
- Security

Interaction model:

The distributed systems are composed of many processes, interacting in complex ways. For example:

1. Multiple server processes may cooperate with one another to provide a service.
2. A set of peer processes may cooperate with one another to achieve a common goal.

Two significant factors affecting interacting processes in a distributed systems:
Communication performance is often a limiting characteristic.

The communication channels in our models are realized in a variety of ways in distributed systems, for example by an implementation of streams or by simple message passing over a computer network.

Communication over a computer network has the following performance characteristics relating to latency, bandwidth, and jitter.

Computer Clusters and Timing Events:

Each computer in a distributed system has its own internal clock, which can be used by local processes to obtain the value of the current time. Therefore, two processes running on different computers can associate timestamps with their events.

Two variants of the interaction model:

In a distributed system it is hard to set time limits on the time taken for processes...
execution, message delivery or clock drift. Two opposing extreme positions provide a pair of simple models, the first has a strong assumption of time and the second makes no assumptions about time.

* Synchronous distributed systems
* Asynchronous distributed systems

**Event ordering:**

The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks.

**Failure Models:**

In a distributed system both processes and communication channel may fail—that is they may depart from what is considered to be correct or desirable behaviour. The failure model defines the way in which failure may occur in order to provide an understanding of the effects of failure.

**Omission failures:**

The faults classified as omission failures refer to cases when a process or
communication channels fails to perform actions that it is supposed to do.

* Process omission failures
* Communication omission failures.

**Arbitrary Failures**

The term arbitrary failure is used to describe the worst possible failure semantics, in which any type of error may occur. For example, a process may set wrong values in its data items, or it may return a wrong value in response to a invocation.

**Timing Failure**

Timing failures are applicable in synchronous distributed systems where time limits are set on process execution time, message delivery time, and clock drift rate.

**Masking Failures**

A service masks a failure, either by hiding it altogether or by converting it into
a more acceptable type of failures. For an example of the latter, check sums are used to mask corrupted messages - effectively converting an arbitrary failure into an omission failure.

Security model:

The security of distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.
Interprocess communication.

The Java API for interprocess communication in the Internet provides both Datagram and Stream communication. These are presented, together with a discussion of their failure models. They provide alternative building blocks for communication protocols.

The application program interface to UDP provides a message passing abstraction — the simplest form of interprocess communication. This enables a sending process to transmit a single message to receiving process. The independent packets containing these messages are called datagrams.

The application program interface to TCP provides the abstraction of a two-way stream between pairs of processes. The information communicated consists of a stream of data items with no message boundaries. Stream provide a building block for producer-consumer communication.
The API for the Internet protocols:

The characteristics of inter-process communication:

Message passing between a pair of processes can be supported by two message communication operations:

* Send
* Receive

Synchronous and Asynchronous communication

In synchronous form of communication, the sending and receiving process synchronize at every message. In this case, both send and receive are blocking operations.

In asynchronous form of communication, the use of send and receive operation is non-blocking so that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process.

Message destinations:

In the Internet protocols, messages are sent to (Internet address, local port) pairs. A local port message destination within a computer
specified as an integer. A port has exactly one receiver, but can have many senders.

Reliability:

Reliable communication in terms of validity and integrity

Ordering:

Some applications require that messages be delivered in sender order — that is the order in which they were transmitted by the sender. The delivery of messages out of sender order is regarded as a failure by such applications.

Sockets:

Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process. Processes may use the same socket for sending and receiving messages.

Ex:

Inet Address aComputer = Inet Address. getByName("bruno.des.qmul.ac.uk");
The Programming Model

The programming model offered by message queue is very simple. It offers an approach to communication in distributed systems through queues. Three styles of receive are generally supported:

* Blocking receive
* Non-blocking receive
* Notify operation

Implementation issues:

The key implementation issues for message queuing system is the choice between centralized and distributed implementations of the concept. Some implementations are centralized, with one or more message queues managed by queue manager located at a given node.
UDP datagram communication:

A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or relates. If a failure occurs, the message may not arrive. A datagram is transmitted between the processes when one process sends it and another receives it.

Some issues relating to datagram communication:

- Message size
- Blocking
- Timeouts
- Receive from any

Failure Model:

It presents a failure model for communication channels and defines reliable communication it having two properties: 1. Integrity and 2. Validity. It suffers the following failures:

- Omission failures
- Duplication
- Ordering
Top-stream communication.

It provides the abstraction of a stream of bytes to which data may be written and from which data may be read. The following characteristics of the network are hidden by the stream abstraction:

* message size
* lost message
* flow control
* message duplication and ordering
* message destination.

External data representation and multicast communication:

The information stored in running programs is represented as data structure whereas the information in messages consist of sequences of bytes. Irrespective of the form of communication used, the data structure must be flattened before transmission and rebuilt on arrival.

One of the following methods can be used to enable two computers to exchange data values.
The values are converted to an agreed external
format before transmission and converted.

The values are transmitted in the sender format,
together with an indication of the format used,
and the recipient converts the values if necessary.

External data representation.

To support RPC or RMI, any data type
that can be passed as an argument or returned as
a result must be able to be flattened and the
individual primitive data values represented
in an agreed format. An agreed standard for the
representation of data structures and primitives
value is called external data representation.

Marshalling is the process of taking a collection
of data items into a form suitable for transmission in a
message.

Unmarshalling is the process of disassembling them
on arrival to produce an equivalent collection of
data items at destination.

Three alternative approaches to external data
representation.
1. CORBA's Common Data Representation (CDR)

It consists of 15 primitive types that include:
- short (16-bit)
- long (32-bit)
- unsigned short
- unsigned long
- float (32-bit)
- double (64-bit)
- char
- boolean (true/false)
- octet (8-bit)

Any constructed type can be built upon these primitive types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Length followed by element in order</td>
</tr>
<tr>
<td>String</td>
<td>Length followed by characters in order</td>
</tr>
<tr>
<td>array</td>
<td>Array elements in order</td>
</tr>
<tr>
<td>struct</td>
<td>In the order of declaration of the components</td>
</tr>
<tr>
<td>enumerated</td>
<td>Unsigned long</td>
</tr>
<tr>
<td>Union</td>
<td>Type tag followed by the selected number</td>
</tr>
</tbody>
</table>
Marshalling in CORBA

Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message. The types of these data structures and the types of the basic data items are described in CORBA IDL, which provides a notation for describing the types of the arguments and results of RMI methods.

Example:

```java
struct person {
    string name;
    string place;
    unsigned long year;
};
```

Java object serialization:

In Java RMI, both object and primitive data values may be passed as arguments and results of method invocation. An object is an instance of a Java class.

Example:

```java
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        Place = aPlace;
        Year = aYear;
    }
    // Accessing instance variables
}
```
Extensible Markup Language (XML)

XML is used to enable clients to communicate with web services for defining the interface and other properties of web services. However, XML is also used in many other ways. It is used in archiving and retrieval systems — although an XML archive may be larger than a binary one, it has the advantage of being readable on any computer.

Ex:

```
  <person id="123456">
    <name>Smith</name>
    <place>Chennai</place>
    <year>2018</year>
    <!-- a comment -->
  </person>
```

XML elements and attributes:

Elements

An element in XML consists of portions of character data surrounded by matching start and end tags.

Attributes

A start tag may optionally include pairs of associated names and values, such as

```
person id="123456".
```
Remote object references

A Remote Object Reference is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked.

Representation of Remote Object Reference

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Address</td>
<td>port Number</td>
<td>time</td>
<td>Object Number</td>
</tr>
<tr>
<td>Interface of Remote Object</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To allow remote objects to be reallocated in a different process on a different computer, the remote object reference should not be used as the address of the remote object.

Remote Method Invocation and objects

The term RMI refers to remote method invocation in a generic way. This should not be confused with particular examples of remote method invocation such as Java RMI. A large portion of current distributed systems software is written in object-oriented languages and RPC can be understood in relation to RMI.
Middleware

Software that provides a programming model above the basic building blocks of processes and message passing is called middleware.

<table>
<thead>
<tr>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI, RPC and Events</td>
</tr>
<tr>
<td>Request-Reply protocol</td>
</tr>
<tr>
<td>External Data Representation</td>
</tr>
</tbody>
</table>

Some form of middleware is the provision to allow middleware the separate components to be written in different programming languages.

* Location transparency
* Communication protocols
* Computer Hardware
* Operating System
* Use of Programming Language
Interface definition language (IDL)

IDLs are designed to allow object implementations in different languages to interoperate one another. An IDL provides a notation for defining an interface in which each of the parameters of a method may be described as from input or output in addition to having its type specified.

Ex:

```java
struct Person {
    string name;
    string place;
    long year;
}

interface PersonList {
    readonly attribute string listName;
    void addPerson (in Person p);
    void getPerson (in string name, out Person p);
    long number();
}
```

Request-reply protocols:

Request-reply communication is synchronous because the client process blocks until the reply arrives from the server. It can also be reliable because the reply from the server is effectively an acknowledgement to the client.
Case Study: WebSphere MQ

WebSphere MQ is middleware developed by IBM based on the concept of message queue, offering an indirect connection between senders and receivers of messages.

Queue, WebSphere MQ are managed by queue managers which host and manage queues and allow applications to access queues through Message Queue Interface (MQI).

MQ introduces the concept of a message channel as a unidirectional connection between two queue managers that is used to forward messages asynchronously from one queue to another.

Two agents are responsible for establishing and maintaining the channel, including an initial negotiation to agree on the properties of the channel.
Case study: Java Messaging Service (JMS)

JMS is a specification of a standardized way for distributed Java programs to communicate indirectly.

JMS distinguishes between the following key roles:

* **JMS consumer** is a program that receives and consumes messages.
* **JMS provider** is any of the multiple systems that implement the JMS specification.
* **JMS message** is an object that is used to communicate information between JMS clients.
* **JMS destination** is an object supporting indirect communication in JMS.

**Programming with JMS**

```
Connection Factory

Connection

Message Producer

Destination People Queue

Session

Message

Message Consumer

Destination People Queue

--> creates

--> communicates
```
Connection can be used to create one or more sessions. A session is a series of operations involving the creation, production, and consumption of messages related to a logical task. The resultant session object also supports operations to create transactions, supporting all-or-nothing execution of a series of operations.

**Shared Memory Approach**

**Distributed Shared Memory**

Distributed Shared Memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. Processes access DSM by reading and updating what appears to be ordinary memory within their address space.

The main point of DSM is that it spares the programmer the concern of message passing when writing applications that might otherwise have to use it.
DMM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly.

**Message Passing versus DMM:**

The DSM and message-passing approaches to programming contrast as follows:

**Service offered:**

Under the message passing model, variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process.

This is done based on

- Programming model
- Efficiency

**Implementation approach to DSM**

The distributed shared memory is implemented with the help of:

1. Specialized Hardware

   The shared memory multiprocessor architecture employs the hardware to make the processors along with the shared memory

   Ex: Load, Store
Paged Virtual memory:

The DSM can be implemented as a virtual memory region by most of the system like

Ex: Ivy, Munin, Mirage, cloud etc

Middleware

DSM may be supported by some of the languages and middleware. It is not implemented with help of hardware or paged virtual memory in this case.

* clients
* servers

Distributed Object

Middleware based on distributed object is designed to provide a programming model based on object oriented principles and therefore to bring the benefits of the object oriented approach to distributed programming.

There are three stand of activity

* distributed systems
* programming languages
* software engineering
<table>
<thead>
<tr>
<th>Objects</th>
<th>Distributed Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Reference</td>
<td>Remote Object Reference</td>
</tr>
<tr>
<td>Interfaces</td>
<td>Remote Interface</td>
</tr>
<tr>
<td>Actions</td>
<td>Distributed Actions</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Distributed Exception</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>Distributed Garbage collection</td>
</tr>
</tbody>
</table>

The added complexities

The associated distributed object middleware must provide additional functionality, as summarized below:

* Inter-object communication
* Life cycle management
* Activation and deactivation
* Persistence
* Additional services

![Diagram showing process accessing DSM, DSM appears as memory to address space of process, mappings between physical memories]
Case Study: Enterprise JavaBeans

Enterprise JavaBeans (EJB)

EJB is a specification of a server-side, managed components architecture and a major element of the Java platform, Enterprise Edition (J2EE).

A set of specifications for client-server programming.

The goal of EJB is to maintain a strong specification of concerns between the various roles involved in developing distributed applications.

The EJB specifications identifies the following key roles:

* bean provider
* Application assembler
* deployer
* Service provider
* Persistence provider
* Container provider
* System administrator

The EJB component model:

A given bean is represented by the set of remote and local business interface together.
with an associated bean class that implements the interface.

Two main styles of beans are supported in the EJB 3.0 specifications:

- Session Beans
- Message-driven Beans

**Enterprise Java Bean Interception**

The Enterprise JavaBeans specification enables the programmer to intercept two ways of operation on beans in order to alter their default behaviour.

- Method calls associated with a business interface
- Life cycle events

**Interception of lifecycle events**

A similar mechanism can be used to intercept and react to lifecycle events associated with a component.

- @ Post Construct
- @ Pre Destroy

The annotations are associated with given methods in the bean class, with the effect that these methods will be called when the associated lifecycle events happen.
* `public byte[] doOperation (RemoteRef s, int operation) Id,
  byte[] arguments)`

  → Sends a request message to the remote object and returns the reply.

* `public byte[] getRequest()`

  → Acquires a client request via the server port.

* `public void sendReply (byte[] reply, InetAddress
  client Host, int client port)`

  → Sends the reply message reply to the client and its Internet address and port.

**Request-reply message structure**

<table>
<thead>
<tr>
<th>Message Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request ID</td>
</tr>
<tr>
<td>Object Reference</td>
</tr>
<tr>
<td>Method ID</td>
</tr>
<tr>
<td>Arguments</td>
</tr>
</tbody>
</table>
Message Identifier

A message identifier consists of two parts:

i) a request ID, which is chosen from an increasing sequence of integers by the sender process.
ii) an identifier of the sender process, e.g., port no. and internet address.

Failure model:

They suffer from the same communication failures.

That is:

i) they suffer from omission failures.
ii) messages are not guaranteed to be delivered in sender order.

Timeouts:

This is not the usual approach – the timeout may have been due to the request or reply message getting lost – and in the latter case, the operation will have been performed.

RPC Exchange protocols:

It produces differing behaviours in the presence of communication failures, are used for implementing various types of RPC.

* The Request Protocol (R)
* The Request-Reply Protocol (RR)
* The Request-Reply-Acknowledge (RRR) protocol
Remote procedure call

The concept of a remote procedure call (RPC) represents a major intellectual breakthrough in distributed computing, with the goal of making the programming of distributed systems look similar, if not identical, to conventional programming—that is, achieving a high level of distribution transparency.

Design issues for RPC

There are three issues that are important in understanding this concept:

i) the style of programming promoted by RPC

ii) the call semantics associated with RPC

iii) the key issue of transparency and how it relates to RPC.

RPC call semantics

The main choices are:

i) Retry—request message

Control whether to retransmit the request message until either a reply is received or the server is assumed to have failed.

ii) Duplicate filtering

Controls when retransmissions are used and whether to filter out duplicate requests at the server.
NO Retransmission of results

Controls whether to keep a history of result messages to enable last results to be retransmitted without re-executing the operations at the server.

Implementation of RPC

The client that accesses a service includes one stub procedure for each procedure in the service interface. The role of a stub procedure is similar to that of a proxy method. It behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifiers and arguments into a request message, which it sends via its communication module to the server.

The client and server stub procedures and the dispatcher can be generated by an interface compiler from the interface definition of the service.
Remote Method Invocation

Remote Method Invocation (RMI) is closely related to RPC but extended into the world of distributed objects. In RMI, a calling object can invoke a method in a potentially remote object.

The commonalities between RMI and RPC are as follows:

1) They both support programming with interfaces with the resultant benefits that stem from this approach.
2) They are both typically constructed on top of request reply protocols and can offer a range of call semantics such as at least once and at most once.
3) They both offer a similar level of transparency.

Design Issues for RMI

The key related design issues related to the object model and, in particular, achieving the transition from objects to distributed objects.
The object model

An object oriented program, for example C++, Java consists of a collection of interacting objects, each of which consists of a set of data and a set of methods. An object communicates with other objects by invoking their methods, generally passing arguments and receiving results.

Object reference

Objects can be accessed via object reference. For example, in Java, a variable that appears to hold an object actually holds a reference to that object.

Interfaces

An interface defines a specification of the signatures of a set of methods without specifying their implementation.

Actions

Action in an object-oriented program is initiated by an object involving a method in another object.

Exceptions

Program can encounter many sorts of errors and unexpected conditions of varying seriousness.
Garbage collection:

It is necessary to provide a means of freeing the space occupied by objects when they are no longer needed. A language such as Java, that can detect automatically when an object is no longer accessible, recovers the space and makes it available for allocation to other objects. This process is called Garbage collection.

Distributed objects

The state of an object consists of the values of its instance variables. In the object-based paradigm the state of a program is partitioned into separate parts, each of which is associated with an object.
Implementation of RMI

Communication module

The two cooperating communication modules carry out the request-reply protocol, which transmits request and reply messages between the client and server.

---

The communication module in the server selects the dispatcher for the class of the object to be invoked, passing on its local reference, which it gets from the remote reference module in return for the remote object identifier in the request message.

Remote reference module:

A remote reference module is responsible for translating between local and remote object references and for creating remote object references.
Servant

A servant is an instance of a class that provides the body of a remote object. Servants live within a server process.

Case Study: Java RMI

Java RMI extends the Java Object model to provide support for distributed objects in the Java language. It allows objects to invoke methods on remote objects using the same syntax as for local invocations.

The programming of distributed applications in Java RMI should be relatively simple because it is a single-language system - remote interfaces are defined in the Java language.

Remote Interfaces in Java RMI

Remote interfaces are defined by extending an interface called Remote provided in the Java.rmi package. The method must be thrown. Remote Exception, but application-specific exceptions may also be thrown.
Parameter and result passing:

The parameters of a method are assumed to be:

- Input parameters and the result of a method is a
  single output parameter.

- Passing remote objects
- Passing non-remote objects

RMI Registry:

The RMI registry is the backend for Java RMI. An
instance of RMI Registry must run on every server computer
that hosts remote objects. It maintains a table
mapping textual, URL-style names to references to
remote objects hosted on that computer.

- void rebuild (String name, Remote Obj)
- void build (String name, Remote Obj)
- void unbind (String name, Remote Obj)
- Remote lookup (String name)
- String list()
Design and Implementation of Java RMI

The original Java RMI system used all of the components as used. The reflection facilities were used to make a generic dispatcher and to avoid the need for skeletons.

- Use of reflection
- Java classes supporting RMI

Group Communication

Group communication offers a service whereby a message is sent to a group and then that message is delivered to all members of the group. Group communication represents an abstraction over multicast communication and may be implemented over IP multicast or an equivalent overlay network, adding significant extra value in terms of managing group membership, detecting failures and providing reliability and ordering guarantees.
The Programming Model

1. The central concept is that of a group with associated group membership.
2. The group communication implements multicast communication, in which a message is sent to all the members of the group by a single operation.

Process groups and object groups

Most work on group services focuses on the concept of process group, that is, groups where the communicating entities are processes.

- Messages are delivered to processes and no further support for object-oriented groups is provided.
- Messages are typically unstructured byte arrays with no support for marshalling of complex data types.

Other key distinctions:

A wide range of group communication services has been developed, and they vary in the assumptions they make:

- Closed and open groups
- Overlapping and non-overlapping groups
- Synchronous and asynchronous systems
Implementation issues.

Reliability and ordering in multicast

Reliability in one-to-one communication was defined in terms of two properties:

i) Integrity

ii) Validity

Group communication services offer ordered multicast, with the option of one or more of the following properties:

* FIFO ordering
* Causal ordering
* Total ordering

Reliability and ordering are examples of co-ordination and agreement in distributed systems.
Group membership management

The important role of group membership management in maintaining an accurate view of the current membership, given that entities may join, leave or instead fail. In more detail, a group membership service has four main tasks.

i) Providing an interface for group membership changes
ii) Failure detection
iii) Notifying members of group membership changes
iv) Performing group address expansion.

Publish-subscribe systems

A public subscribe system where publishers publish structured events to an event service and subscribers express interest in particular events through subscriptions which can be arbitrary patterns over the structured events.

Applications of publish-subscribe systems

* Financial information systems
* Live feeds of real-time data
* Support for cooperative learning
Support for ubiquitous computing

- A broad set of monitoring applications

Dealing Room System

A simple dealing room system whose task is to allow dealers using computers to see the latest information about the market prices of the stocks they deal in.
Characteristics of Publish-Subscribe Systems:

Publish-Subscribe systems have two main characteristics:

1) Heterogeneity
2) Asynchronicity

The Programming Model:

The programming model in publish-subscribe system is based on a small set of operations. Publishers disseminate an event \( e \) through publish(e) operation and subscribes express an interest in a set of events through subscriptions.
The expressiveness of publish-subscribe systems is determined by the subscription model, with a number of schemes defined and considered here in increasing order of sophistication:

* Channel-based
* Topic-based
* Content-based
* Type-based

Message Queue

Message queues are a further important category of indirect communication systems. Whereas groups and publish-subscribe provide a one-to-many style of communication, message queues provide a point-to-point service using the concept of a message queue as an indirection, thus achieving the desired properties of space and time uncoupling.
For example, in the code fragment below from the eShop, TidyUp will be called just before the component is destroyed.

```java
@Stateful
public class eShop implements Orders {

...

public void MakeOrder (...) {

...}

...}

@preDestroy void TidyUp

...}

...}

From Object to Components

The techniques incorporated in CORBA and related platforms have proved to be successful in tackling many of the key issues related to distributed computing, especially related to resolving heterogeneity and enabling capacity, portability and interoperability of distributed systems software.
Issues with object-oriented middleware:

Implicit dependencies:

A distributed system offers a contract to the outside world in terms of the interface, it offers to the distributed environment. The contract represents a binding agreement between the provider of the object and users of that object in terms of its expected behaviour.

Interaction with middleware:

Despite the goals of transparency, it is clear that in using distributed object middleware the programmer is exposed to many relatively low-level details associated with the middleware architecture.

Lack of separation of distributed concerns:

Programmers using distributed object middleware also have deal to explicitly with non-functional concerns related to issues such as security, transactions, coordination and replication.

* Programmer must have an intimate knowledge of the full details of all the associated distributed system services.

* The implementation for a given object will contain application code alongside calls to distributed system services and to the underlying middleware interfaces.
No support for deployment:

While technologies such as CORBA and Java RMI make it possible to develop arbitrary distributed configuration of objects, there is no support for the deployment of such configuration.

Essence of components:

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only.

The dependencies are also represented as interfaces. More specifically, a component is specified in terms of a contract, which includes:

- A set of provided interface
- A set of required interface

![Diagram of software components](image-url)
Components and distributed systems:

A range of component-based middleware technologies have emerged, including Enterprise JavaBeans and the CORBA Component Model (CCM), an evolution of CORBA from an object-based to a component-based platform.

**Containers**

The concept of containers is absolutely central to component-based middleware, which consists of:

* a Front-end client
* a container holding one or more components that implement the application or business logic
* a system service that manages the associated data in persistent storage

**Support for deployment:**

Component-based middleware provides support for the deployment of component configuration; releases of software are packaged as software architecture together with deployment descriptors that fully describe how the configurations should be deployed in a distributed environment.

Deployment descriptors are typically written in XML and include sufficient information to ensure that
components are correctly connected using appropriate protocols and distributed middleware.

* The underlying middleware and platform are configured to provide the right level of support to the component configuration.

* The associated distributed system services are set up to provide the right level of security, transaction support and so on.