Module-2: Architectures for Parallel and Distributed Computing

1. Eras of computing

The two fundamental and dominant models of computing are: Sequential and parallel.

The four key elements of computing developed during these eras were:

1. Architectures
2. Compilers
3. Applications and
4. Problem-solving environments

The computing era started with a development in hardware architectures, which enabled the creation of system software – particularly in the area of compilers and operating systems – which supported the management of such systems and the development of applications.

2. Parallel vs. Distributed Computing

The terms parallel computing and distributed computing are often used interchangeably, even though they mean slightly different things.

Parallel implies a tightly coupled system. Parallel computing refers to a model in which the computation is divided among several processors sharing the same memory. The architecture of a parallel computing system is often characterized by the homogeneity of components: each processor is of the same type, and it has the same capability as the others. The shared memory has a single address space, which is accessible to all the processors. Parallel programs are then broken down into several units of execution that can be allocated to different processors and can communicate with each other by means of the shared memory.

Distributed refers to a wider class of system, including those that are tightly coupled. The term distributed computing encompasses any architecture or system that allows the computation to be broken down into units and executed concurrently on different computing elements, whether these are processors on different nodes, processors on the same computer, or cores within the same processor. Therefore, distributed computing includes a wider range of systems and applications than parallel computing.

3. Elements of distributed computing

3.1 General Concepts and Definitions

A distributed system is a collection of independent computers that appears to its users as a single coherent system.

Distributed systems are composed of more than one computer that collaborate together, it is necessary to provide some sort of data and information exchange between them, which generally occurs through the network

A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages.

3.2 Components of a Distributed System

A distributed system is the result of the interaction of several components that traverse the entire computing stack from hardware to software.

It emerges from the collaboration of several elements that by working together give users the illusion of a single coherent system.

Figure 2.10 provides an overview of the different layers that are involved in providing the services of a distributed system.

At the very **bottom layer**, computer and network hardware constitute the physical infrastructure; these components are directly managed by the next layer, operating system, which provides the basic services for interprocess communication (IPC), process scheduling and management, and resource management in terms of file system and local devices. Taken together these two layers become the platform on top of which specialized software is deployed to turn a set of networked computers into a distributed system. At the operating system level, IPC services are implemented on top of standardized communication protocols such Transmission Control Protocol/Internet Protocol (TCP/IP), User Datagram Protocol (UDP) or others.

The **middleware layer** leverages such services to build a uniform environment for the development and deployment of distributed applications. This layer supports the programming paradigms for distributed systems the middleware develops its own protocols, data formats, and programming language or frameworks for the development of distributed applications. All of them constitute a uniform interface to distributed application developers that is completely independent from the underlying operating system and hides all the heterogeneities of the bottom layers.

The **top layer** of the distributed system stack is represented by the applications and services designed and developed to use the middleware. These can serve several purposes and often expose their features in the form of graphical user interfaces (GUIs) accessible locally or through the Internet via a Web browser.

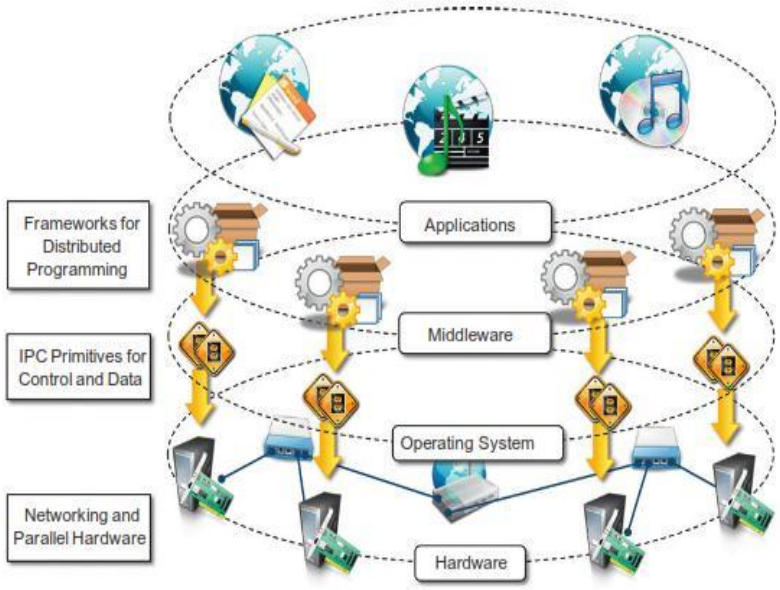


FIGURE 2.10
A layered view of a distributed system.

A very good example is constituted by Infrastructure-as-a-Service (IaaS) providers such as Amazon Web Services (AWS), which provide facilities for creating virtual machines, organizing them together into a cluster, and deploying applications and systems on top.

Figure 2.11 shows an example of how the general reference architecture of a distributed system is contextualized in the case of a cloud computing system. Note that hardware and operating system layers make up the bare-bone infrastructure of one or more datacenters, where racks of servers are deployed and connected together through high-speed connectivity. This infrastructure is managed by the operating system, which provides the basic capability of machine and network management. The core logic is then implemented in the middleware that manages the virtualization layer, which is deployed on the physical infrastructure in order to maximize its utilization and provide a customizable runtime environment for applications.

The middleware provides different facilities to application developers according to the type of services sold to customers. These facilities, offered through Web 2.0-compliant interfaces, range from virtual infrastructure building and deployment to application development and runtime environments.

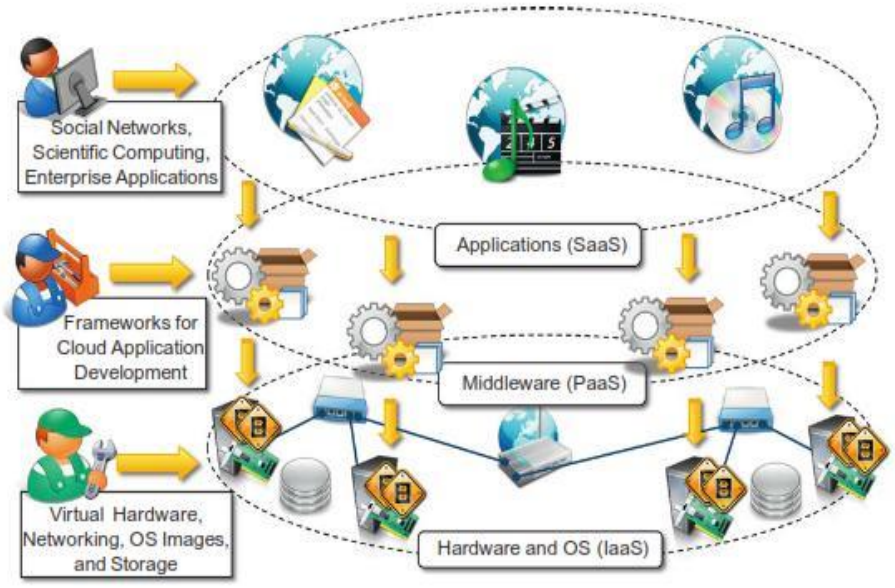


FIGURE 2.11
A cloud computing distributed system.

3.3 Architectural Styles for Distributed Computing

There are many different ways to organize the components that, taken together, constitute such an environment. The interactions among these components and their responsibilities give structure to the middleware and characterize its type or, in other words, define its architecture. Architectural styles aid in understanding and classifying the organization of software systems in general and distributed computing in particular.

Architectural styles are mainly used to determine the vocabulary of components and connectors that are used as instances of the style together with a set of constraints on how they can be combined Architectural styles for distributed systems are helpful in understanding the different roles of components in the system and how they are distributed across multiple machines.

The architectural styles are categorized into two major classes:

- **Software architectural styles:** The style which explains ***logical organization*** of the software
- **System architectural styles:** the styles that describe the ***physical organization*** of distributed software systems in terms of their major components.

1. Components and connectors

A **component** represents a unit of software that encapsulates a function or a feature of the system. Examples of components can be programs, objects, processes, pipes, and filters.

A **connector** is a communication mechanism that allows cooperation and coordination among components. Connectors are not encapsulated in a single entity, but they are implemented in a distributed manner over many system components.

2. Software Architectural Styles

Software architectural styles are based on the logical arrangement of software components. They are helpful because they provide an intuitive view of the whole system, despite its physical deployment.

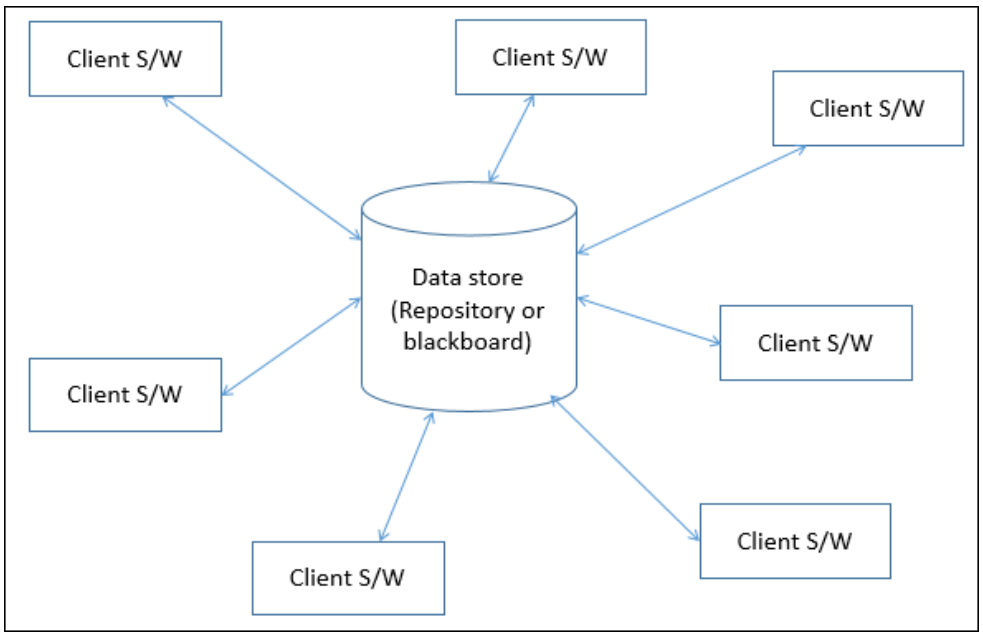
They also identify the main abstractions that are used to shape the components of the system and the expected interaction patterns between them. The software architectural styles are classified as shown in Table 2.2.

Table 2.2 Software Architectural Styles

Category	Most Common Architectural Styles
Data-centered	Repository Blackboard
Data flow	Pipe and filter Batch sequential
Virtual machine	Rule-based system Interpreter
Call and return	Main program and subroutine call/top-down systems Object-oriented systems Layered systems
Independent components	Communicating processes Event systems

a) Data-centered architectures

- In data-centered architecture, the data is centralized and accessed frequently by other components, which modify data.
- The main purpose of this style is to achieve integrity of data.
- The most well-known examples of the data-centered architecture is a database architecture
- For example, a set of related tables with fields and data types in an RDBMS.
- Another example of data-centered architectures is the web architecture which has a common data schema (i.e. meta-structure of the Web)



Types of Components: There are two types of components –

- A **central data** structure or data store or data repository, which is responsible for providing permanent data storage. It represents the current state.
- A **data accessor** or a collection of independent components that operate on the central data store, perform computations, and might put back the results.

Interactions or communication between the data accessors is only through the data store. The data is the only means of communication among clients.

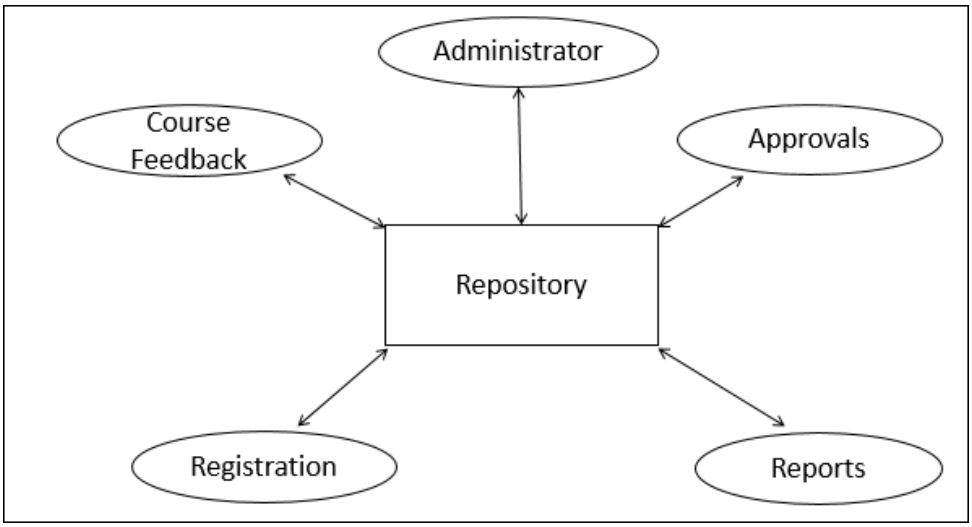
The flow of control differentiates the architecture into two categories –

- Repository Architecture Style
- Blackboard Architecture Style

- **Repository Architecture Style**

In Repository Architecture Style, the **data store is passive and the clients** (software components or agents) **of the data store are active**, which control the logic flow.

- The client sends a request to the system to perform actions (e.g. insert data).
- The computational processes are independent and triggered by incoming requests.
- If the types of transactions in an input stream of transactions trigger selection of processes to execute, then it is traditional database or repository architecture, or passive repository.
- This approach is widely used in DBMS, library information system.



- **Blackboard Architecture Style**

In Blackboard Architecture Style, the **data store is active, and its clients are passive**. Therefore, the logical flow is determined by the current data status in data store.

Parts of Blackboard Model:

The blackboard model is usually presented with three major parts −

1. Knowledge Sources (KS)
2. Blackboard Data Structure
3. Control

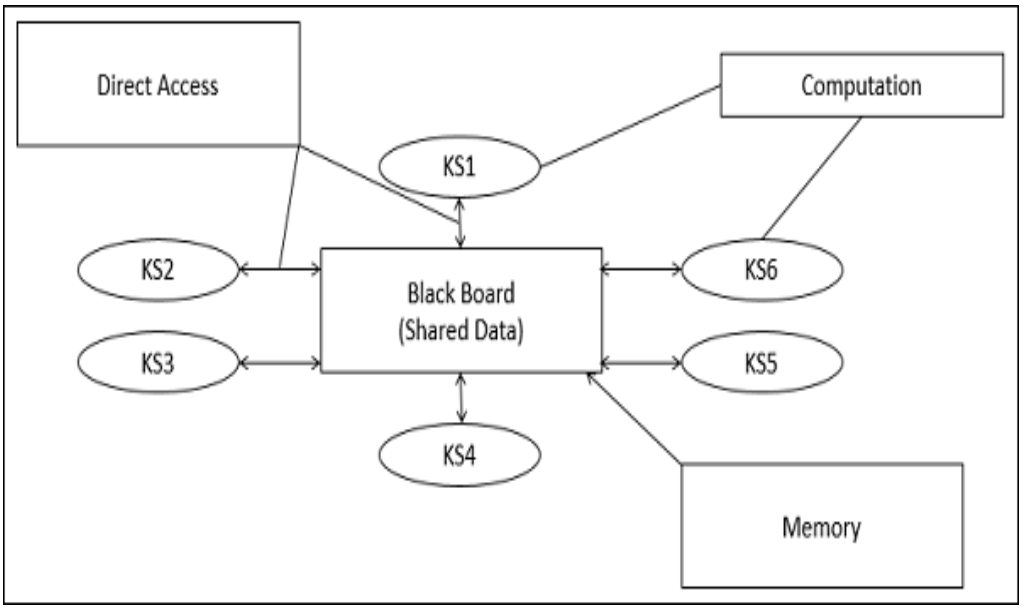
1. Knowledge Sources (KS)

Knowledge Sources, also known as **Listeners** or **Subscribers** are **distinct and independent units.** They solve parts of a problem and aggregate partial result. Interaction among knowledge sources takes place uniquely through the blackboard.

2. Blackboard Data Structure

The problem-solving state data is organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.

3. Control: Control manages tasks and checks the work state.



b) Data flow architectures

- Data Flow Architecture is transformed input data by a **series of computational** or manipulative components into output data.

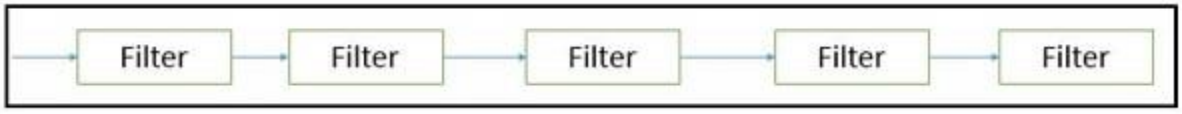
There are Two types of execution sequences between modules:

1. Batch Sequential style
2. Pipe and Filter style

1. Batch Sequential style

The batch sequential style is characterized by an ordered sequence of separate programs executing one after the other.

These programs are chained together by providing as input for the next program the output generated by the last program after its completion, which is most likely in the form of a file.



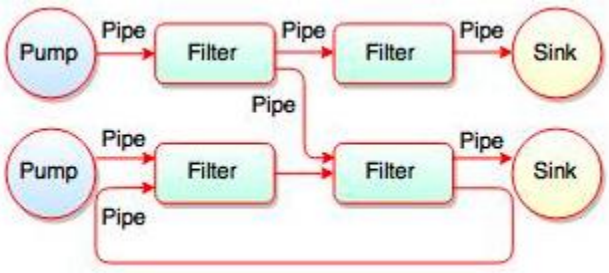
The above diagram shows the flow of batch sequential architecture. It provides simpler divisions on subsystems and each subsystem can be an independent program working on input data and produces output data.

For example, many distributed applications for scientific computing are defined by jobs expressed as sequence of programs that, for **Example**, pre-filter, analyse, and post process data. It is very common to compose these phases using the batch sequential style.

2. Pipe and Filter style

What is meant by Pipe?

- Pipe is a connector which passes the data from one filter to the next.
- Pipe is a directional sequence of data transformations of data implemented by a data buffer to store all data, until the next filter has time to process it.
- It transfers the data from one data source to one data sink.
- Pipes are the stateless data stream.



What are the Filters?

- Filter is a component.
- It has interfaces from which a set of inputs can flow in, and a set of outputs can flow out.
- It transforms and refines input data.
- Filters are the independent entities.

c) Virtual Machine Architectures

- It is an abstract execution environment (generally referred as a virtual machine) that simulates features that are not available in the hardware or software.
- Applications and systems are implemented on top of this layer and become portable over different hardware and software environments.
- The general interaction flow for systems implementing this pattern is – the program (or the application) defines its operations and state in an abstract format, which is interpreted by the virtual machine engine.

Architectural styles Virtual Machines

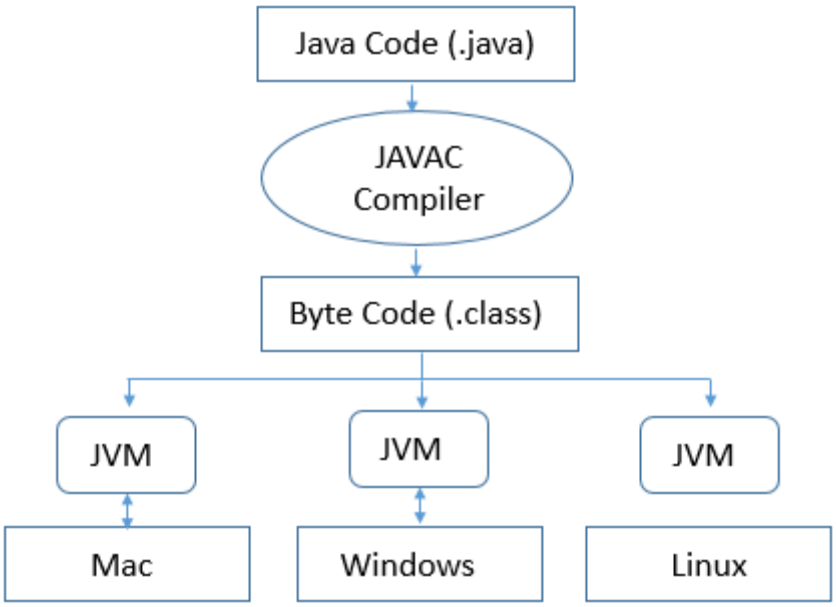


Fig: Role of Java Virtual Machine

Popular examples within this category are

1. Rule Based Systems and
2. Interpreters

1. Rule Based Systems

- This architecture is characterized by representing the abstract execution environment as an **inference engine**.
- Programs are expressed in the form of **rules or predicates** that hold **true**.
- The input data for applications is generally represented by a set of assertions or facts that the inference engine uses to activate rules or to apply predicates, thus transforming data.
- The examples of rule-based systems can be found in the networking domain: **Network Intrusion Detection Systems (NIDS)** often rely on a set of rules to identify abnormal behaviours connected to possible intrusion in computing systems.

2. Interpreters

- The core feature of the interpreter style is the presence of an engine that is used to interpret a pseudo-program expressed in a format acceptable for the interpreter.
- The interpretation of the pseudo-program constitutes the execution of the program itself.
- Systems modelled according to this style exhibit **four main components**:
  - The interpretation engine that executes the core activity of this style.
  - An internal memory that contains the pseudo-code to be interpreted.
  - A representation of the current state of the engine and
  - A representation of the current state of the program being executed.

This model is quite useful in designing virtual machines for high-level programming (Java, C#) and scripting languages (Awk, PERL, and so on).

d) Call and return architectures

- This identifies all systems that are organized into components mostly **connected together by method calls**.
- The activity of systems modelled in this way is characterized by a chain of method calls whose overall execution and composition identify the execution one or more operations.

There are three categories in this

1. Top-down Style

2. Object Oriented Style

3. Layered Style

1. Top-down Style

- It leads to a divide-and-conquer approach to problem resolution.
- Systems developed according to this style are composed of one large main program that accomplishes its tasks by invoking subprograms or procedures.
- The **calling program passes information with parameters** and receives data from return values or parameters.
- Method calls can also extend beyond the boundary of a single process by leveraging techniques for remote method invocation, such as remote procedure call (RPC) and all its descendants.

2. Object Oriented Style

- Systems are specified in terms of **classes** and implemented in terms of **objects**. Classes define the type of components by specifying the data that represent their state and the operations that can be done over these data.

3. Layered Style

- The layered system style allows the design and implementation of software systems in terms of layers, which provide a different level of abstraction of the system.
- Each layer generally operates with at most **two layers**: the one that provides a **lower abstraction** level and the one that provides a **higher abstraction layer**. Specific protocols and interfaces define how adjacent layers interact.
- A **user or client** generally **interacts** with the layer at the **highest abstraction**, which in order to carry its activity, interacts and uses the services of the **lower layer**.
- The **advantage** of the layered style is that it **allows to different levels of abstractions** by encapsulating together all the operations that belong to a specific level.

e) Architectural styles based on independent components

- Independent components that have their own life cycles, which interact with each other to perform their activities.

There are two major categories within this class:

1. Communicating processes and
2. Event systems

1. Communicating processes

- Components are represented by independent processes that leverage IPC facilities for coordination management. This is an abstraction that is quite suitable to modelling distributed systems that, being distributed over a network of computing nodes, are necessarily composed of several concurrent processes.

Example: client-server and peer-to-peer communication

2. Event systems

- The components of the system are loosely coupled and connected.
- Each component also publishes (or announces) a collection of events with which other components can register.
- **In general**, other components provide a callback that will be executed when the event is activated. During the activity of a component, a specific runtime condition can activate one of the exposed events, thus triggering the execution of the callbacks registered with it.

3. System Architectural Styles

System architectural styles cover the physical organization of components and processes over a distributed infrastructure.

There are Two fundamental reference style

1. Client/Server and

2. Peer-to-Peer

1. Client / Server

The client/server model features two major components: a server and a client. These two components **interact with each other through a network connection using a given protocol**. **The communication is unidirectional**: The **client issues a request** to the server, and after processing the request the **server returns a response**. There could be multiple client components issuing requests to a server that is passively waiting for them. Hence, the important operations in the client-server paradigm are request, **accept** (client side), and **listen** and response (server side).

For the client design, we identify two major models:

a. Thin-client model.

b. Fat-client model.

a. Thin-client model. In this model, the load of data processing and transformation is put on the server side, and the client has a light implementation that is mostly concerned with retrieving and returning the data it is being asked for, with no considerable further processing.

b. Fat-client model. In this model, the client component is also responsible for processing and transforming the data before returning it to the user, whereas the server features a relatively light implementation that is mostly concerned with the management of access to the data.

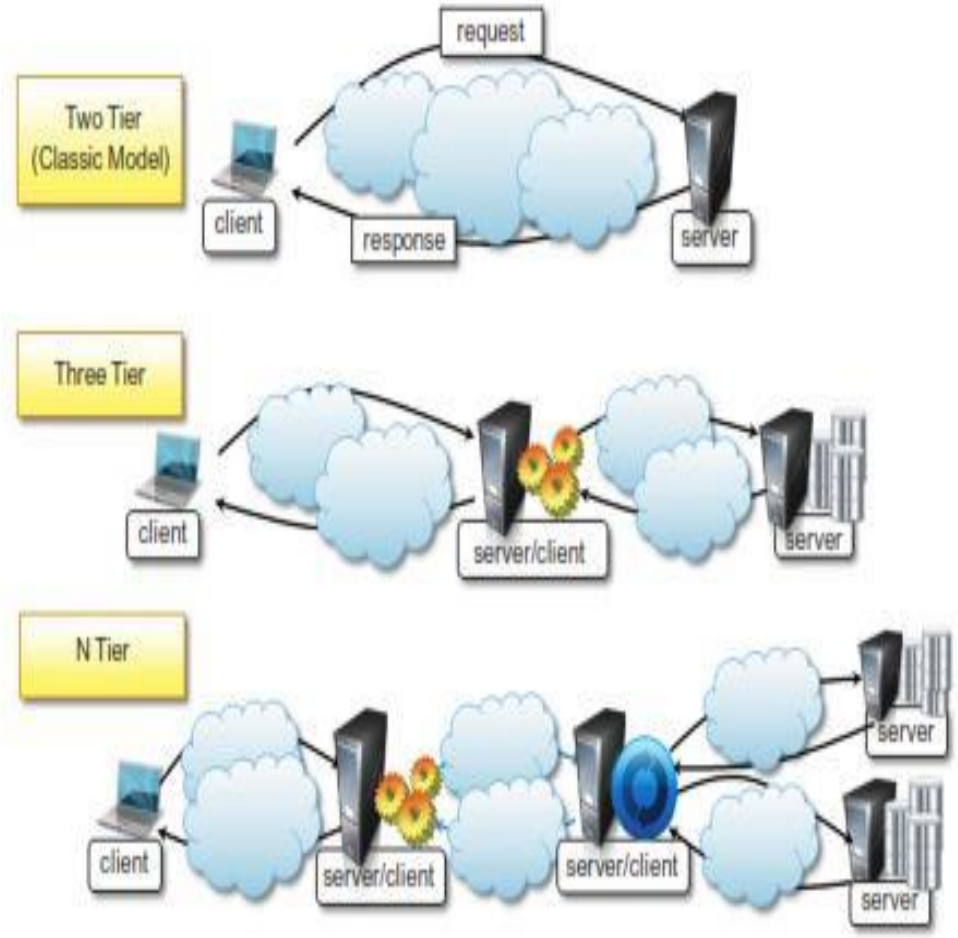


FIGURE 2.12

Client/server architectural styles.

1) Two-tier architecture

This architecture partitions the systems into two tiers, which are located one in the client component and the other on the server. The client is responsible for the presentation tier by providing a user interface; the server concentrates the application logic and the data store into a single tier. The server component is generally deployed on a powerful machine that is capable of processing user requests, accessing data, and executing the application logic to provide a client with a response. This architecture is suitable for systems of limited size and suffers from scalability issues.

2) Three-tier architecture/N-tier architecture

The three-tier architecture separates the presentation of data, the application logic, and the data storage into three tiers. This architecture is generalized into an N-tier model in case it is necessary to further divide the stages composing the application logic and storage tiers. This model is generally more scalable than the two-tier one because it is possible to distribute the tiers into several computing nodes, thus isolating the performance bottlenecks.

2. Peer-to-Peer

The peer-to-peer model, depicted in Figure 2.13, introduces a symmetric architecture in which all the components called peers, play the same role, and incorporate both client and server capabilities of the client/server model.

Each peer acts as a server when it processes requests from other peers and as a client when it issues requests to other peers.

With respect to the client/server model that partitions the responsibilities of the IPC between server and clients, the peer-to peer model attributes the same responsibilities to each component. Therefore, this model is quite suitable for highly decentralized architecture, which can scale better along the dimension of the number of peers.

FIGURE 2.13
Peer-to-peer architectural style.

3.4 Models for Inter-Process Communication

- Distributed systems are composed of a collection of concurrent processes interacting with each other by means of a network connection.
- IPC is a fundamental aspect of distributed systems design and implementation.
- IPC is used to either exchange data and information or coordinate the activity of processes.
- IPC is what ties together the different components of a distributed system, thus making them act as a single system.
- There are several different models in which processes can interact with each other – these maps to different abstractions for IPC.
- Among the most relevant that we can mention are shared memory, remote procedure call (RPC), and **message passing**.

There are two ways for Inter-Process communications

1. Message-based communication
2. Models for Message-based Communication

1. Message-based communication

- The abstraction of message has played an important role in the evolution of the model and technologies enabling distributed computing.
- The definition of distributed computing – is the one in which components located at networked computers communicate and coordinate their actions **only by-passing messages**. The term messages, in this case, identifies any discrete amount of information that is passed from one entity to another. It encompasses any form of data representation that is limited in size and time, whereas this is an invocation to a remote procedure or a serialized object instance or a generic message.
- The term message-based communication model can be used to refer to any model for IPC.
- Several distributed programming paradigms eventually use message-based communication despite the abstractions that are presented to developers for programming the interactions of distributed components.

Most popular and important message passing techniques are:

1) Message Passing: This paradigm introduces the concept of a message as the main abstraction of the model. *The entities exchanging information explicitly encode in the form of a message the data to be exchanged*. The structure and the content of a message vary according to the model. **Examples** of this model are the 1. Message-Passing-Interface (MPI) and 2. openMP (open multi-processing)

2) Remote Procedure Call (RPC): This paradigm extends the concept of procedure call beyond the boundaries of a single process, thus triggering the execution of code in remote processes.

3) Distributed Objects: This is an implementation of the RPC model for the object-oriented paradigm and contextualizes this feature for the remote invocation of methods exposed by objects. **Examples** of distributed object infrastructures are Common Object Request Broker Architecture (CORBA), Component Object Model (COM, DCOM, and COM+), Java Remote Method Invocation (RMI), and .NET Remoting.

4) Distributed agents and active Objects: Programming paradigms based on agents and active objects involve by definition the presence of instances, whether they are agents of objects, despite the existence of requests.

5) Web Service: An implementation of the RPC concept over HTTP; thus, allowing the interaction of components that are developed with different technologies. A Web service is exposed as a remote object hosted on a Web Server, and method invocation are transformed in HTTP requests, using specific protocols such as Simple Object Access Protocol (SOAP) or Representational State Transfer (REST).

2. Models for Message-based Communication

1) Point-to-Point message model:

-direct addressing between two components and address of component is essential.

2) Publish – and – Subscribe message model

Two roles are present: publisher and subscriber

Subscriber should register with **Publisher** to send and receive messages

Two strategies for dispatching events or messages to subscribers from publisher

i. Push Strategy: publisher notifies all subscribers, then subscribers communicate.

ii. Pull Strategy: publisher will be open with an event/message, subscriber should always be checking whether any message present.

3) Request- reply message model: Request sent by component to other component, if request accepted then communication is established.

4. Technologies for distributed computing

4.1 Remote Procedure Call (RPC)

4.2 Distributed Object Frameworks

4.3 Service Oriented Computing

4.1 Remote Procedure Call (RPC)

- RPC is the fundamental abstraction enabling the execution procedures on clients' request.
- RPC allows extending the concept of a procedure call beyond the boundaries of a process and a single memory address space.
- The called procedure and calling procedure may be on the same system or they may be on different systems.
- The important aspect of RPC is marshalling and unmarshalling.
- Marshalling is the process of *transforming the memory representation of an object to a data format* suitable for the storage and transmission. Unmarshalling refers to the *process of transforming a representation of an object that is used for storage or transmission to a representation of the object* that is executable.

Below diagram illustrates the major components that enable an RPC system. The system is based on a client/server model. The server process maintains a registry of all the available procedures that can be remotely invoked and listens for requests from clients that specify which procedure to invoke, together with the values of the parameters required by the procedure.

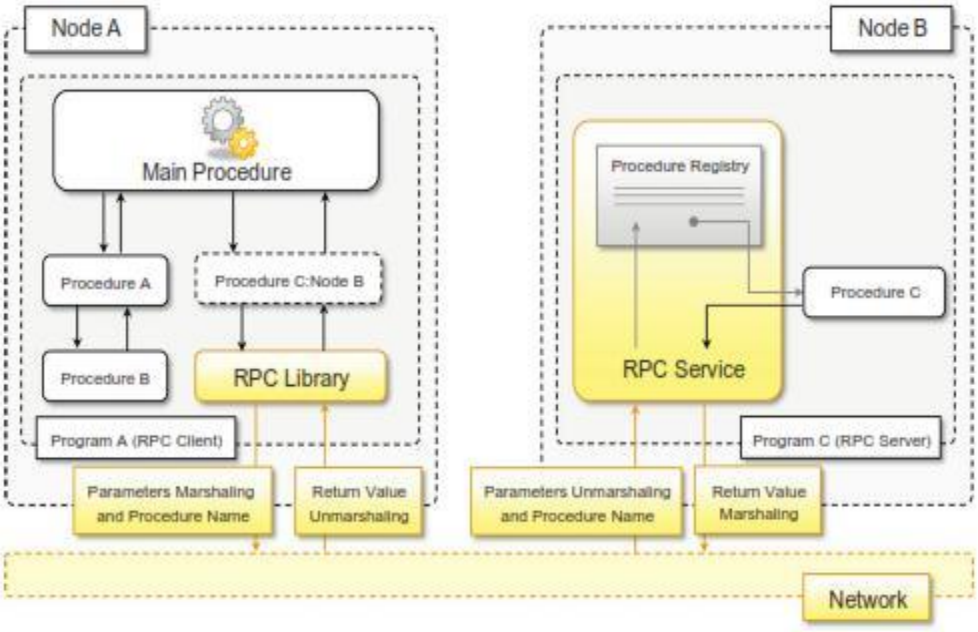


FIGURE 2.14
The RPC reference model.

4.2 Distributed Object Frameworks

Distributed object frameworks extend object-oriented programming systems by allowing objects to be distributed across a heterogeneous network and provide facilities so that they can coherently act as though they were in the same address space.

Distributed object frameworks leverage the basic mechanism introduced with RPC and extend it to enable the remote invocation of object methods and to keep track of references to objects made available through a network connection.

With respect to the RPC model, the infrastructure manages instances that are exposed through well-known interfaces instead of procedures.

Therefore, the common interaction pattern is the following:

1. The server process maintains a registry of active objects that are made available to other processes. According to the specific implementation, active objects can be published using interface definitions or class definitions.

2. The client process, by using a given addressing scheme, obtains a reference to the active remote object. This reference is represented by a pointer to an instance that is of a shared type of interface and class definition.

3. The client process invokes the methods on the active object by calling them through the reference previously obtained. Parameters and return values are marshaled as happens in the case of RPC.

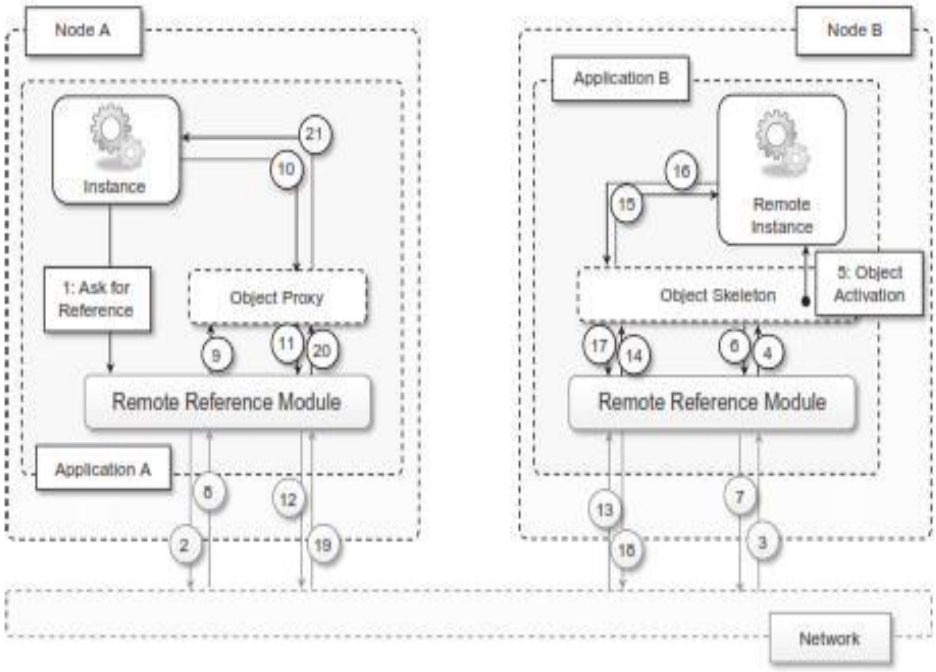


FIGURE 2.15

The distributed object programming model.

Examples of distributed Object frameworks

- Common Object Request Broker Architecture (CORBA): cross platform and cross language interoperability among distributed components.
- Distributed Component Object Model (DCOM/COM+): Microsoft technology for distributed object programming before the introduction of .NET technology.
- Java Remote Method Invocation (RMI): technology provided by Java for enabling RPC among distributed Java objects.
- .NET Remoting: IPC among .NET applications, a uniform platform for accessing remote objects from within any application developed in any of the languages supported by .NET.

4.3 Service Oriented Computing

- Service
- Service-Oriented Architecture (SOA)
- Web Services

- **Service**

A service encapsulates a software component that provides a set of coherent and related functionalities that can be reused and integrated into bigger and more complex applications. Service oriented computing organizes distributed systems in terms of services, which represent the major abstraction for building systems.

Service orientation expresses applications and software systems as an aggregation of services that are coordinated within a service-oriented architecture (SOA).

Even though there is no designed technology for the development of service-oriented software systems, web services are the de facto approach for developing SOA.

Web services, the fundamental component enabling Cloud computing systems, leverage the

Internet as the main interaction channel between users and the system.

The Four major characteristics that identify a service:

1. **Boundaries are explicit:** A service-oriented application is generally composed of services that are spread across different domains, trust authorities, and execution environments.
2. **Services are autonomous:** Services are components that exist to offer functionality and are aggregated and coordinated to build more complex system.
3. **Services share schema and contracts, not class or interface definitions**: Services are not expressed in terms of classes or interfaces, as happens in object-oriented systems, but they define themselves in terms of schemas and contracts.
4. **Services compatibility is determined based on policy:** Service orientation separates structural compatibility from semantic compatibility.

- **Service-Oriented Architecture (SOA)**

SOA is an architectural style supporting service orientation. It organizes a software system into a collection of interacting services.

SOA encompasses a set of design principles that structure system development and provide means for integrating components into a coherent and decentralized system.

SOA based computing packages functionalities into a set of interoperable services, which can be integrated into different software systems belonging to separate business domains.

There are two major roles within SOA:

- Service Provider
- Service Consumer

The following guiding principles, which characterize SOA platforms, are winning features within an enterprise context:

- **Standardized service contract**. Services adhere to a given communication agreement, which is specified through one or more service description documents.

- **Loose coupling**. Services are designed as self-contained components, maintain relationships that minimize dependencies on other services, and only require being aware of each other. Service contracts will enforce the required interaction among services. This simplifies the flexible aggregation of services and enables a more agile design strategy that supports the evolution of the enterprise business.

- **Abstraction**. A service is completely defined by service contracts and description documents. They hide their logic, which is encapsulated within their implementation. The use of service description documents and contracts removes the need to consider the technical implementation details and provides a more intuitive framework to define software systems within a business context.

- **Reusability**. Designed as components, services can be reused more effectively, thus reducing development time and the associated costs. Reusability allows for a more agile design and cost- effective system implementation and deployment. Therefore, it is possible to leverage third- party services to deliver required functionality by paying an appropriate fee rather developing the same capability in-house.

- **Autonomy**. Services have control over the logic they encapsulate and, from a service consumer point of view, there is no need to know about their implementation.

- **Lack of state**. By providing a stateless interaction pattern (at least in principle), services increase the chance of being reused and aggregated, especially in a scenario in which a single service is used by multiple consumers that belong to different administrative and business domains.

- **Discoverability**. Services are defined by description documents that constitute supplemental metadata through which they can be effectively discovered. Service discovery provides an effective means for utilizing third-party resources.

- **Composability**. Using services as building blocks, sophisticated and complex operations can be implemented. Service orchestration and choreography provide a solid support for composing services and achieving business goals.

- **Web Services**
  - Web Services are the prominent technology for implementing SOA systems and applications. They leverage Internet technologies and standards for building distributed systems.
  - Several aspects make Web Services the technology of choice for SOA.
  - First, they allow for interoperability across different platforms and programming languages. Second, they are based on well-known and vendor-independent standards such as HTTP, SOAP, and WSDL.
  - Third, they provide an intuitive and simple way to connect heterogeneous software systems, enabling quick composition of services in distributed environment.

FIGURE 2.16

A Web services interaction reference scenario.