Thakur Educational Trust's (Regd.)

THAKUR RAMNARAYAN COLLEGE OF ARTS & COMMERCE

ISO 21001:2018 Certified

PROGRAMME: B.Sc (I.T)

CLASS: S.Y.B.Sc (I.T)

SUBJECT NAME: SOFTWARE ENGINEERING

SEMESTER: IV

FACULTY NAME: Ms. SMRITI DUBEY

UNIT V

Chapter 4 – Distributed Software Engineering

Concepts:

Introduction

Virtually all large computer-based systems are now distributed systems. A distributed system is one involving several computers, in contrast with centralized systems where all of the system components execute on a single computer**. Tanenbaum and Van Steen (2007) define a distributed system to be: ". . .a collection of independent computers that appears to the user as a single coherent system."**

Information processing is distributed over several computers rather than confined to a single machine. Distributed software engineering is therefore very important for enterprise computing systems.

Benefits of distributed systems:

1. Resource sharing - A distributed system allows the sharing of hardware and software resources—such as disks, printers, files, and compilers—that are associated with computers on a network.

2. Openness - Distributed systems are normally open systems, which means that they are designed around standard protocols that allow equipment and software from different vendors to be combined.

3. Concurrency - In a distributed system, several processes may operate at the same time on separate computers on the network. These processes may (but need not) communicate with each other during their normal operation.

4. Scalability - In principle at least, distributed systems are scalable in that the capabilities of the system can be increased by adding new resources to cope with new demands on the system. In practice, the network linking the individual computers in the system may limit the system scalability.

5. Fault tolerance - The availability of several computers and the potential for replicating information means that distributed systems can be tolerant of some hardware and software failures. In most distributed systems, a degraded service can be provided when failures occur; complete loss of service only occurs when there is a network failure.

Distributed System Issues

Distributed systems are more complex than systems that run on a single processor. Complexity arises because different parts of the system are independently managed as is the network. There is no single authority in charge of the system so top-down control is impossible.

The nodes in the system that deliver functionality are often independent systems with no single authority in charge of them. The network connecting these nodes is a separately managed system. It is a complex system in its own right and cannot be controlled by the owners of systems using the network

Some of the most important design issues that have to be considered in distributed systems engineering are:

1. Transparency - To what extent should the distributed system appear to the user as a single system? When it is useful for users to understand that the system is distributed?

2. Openness - Should a system be designed using standard protocols that support interoperability or should more specialized protocols be used that restrict the freedom of the designer?

3. Scalability - How can the system be constructed so that it is scalable? That is, how can the overall system be designed so that its capacity can be increased in response to increasing demands made on the system?

4. Security - How can usable security policies be defined and implemented that apply across a set of independently managed systems?

5. Quality of service - How should the quality of service that is delivered to system users be specified and how should the system be implemented to deliver an acceptable quality of service to all users?

6. Failure management - How can system failures be detected, contained (so that they have minimal effects on other components in the system), and repaired?

In an ideal world, the fact that a system is distributed would be transparent to users. This means that users would see the system as a single system whose behavior is not affected by the way that the system is distributed. In practice, this is impossible to achieve. Central control of a distributed system is impossible and, as a result, individual computers in a system may behave differently at different times.

Client Server Computing

Distributed systems that are accessed over the Internet are normally organized as client–server systems. In a client–server system, the user interacts with a program running on their local computer (e.g., a web browser or phone-based application).

This interacts with another program running on a remote computer (e.g., a web server). The remote computer provides services, such as access to web pages, which are available to external clients.

In a client–server architecture, an application is modeled as a set of services that are provided by servers. Clients may access these services and present results to end users (Orfali and Harkey, 1998). Clients need to be aware of the servers that are available but do not know of the existence of other clients. Clients and servers are separate processes, as shown in Figure.

This illustrates a situation in which there are four servers (s1–s4), that deliver different services. Each service has a set of associated clients that access these services.

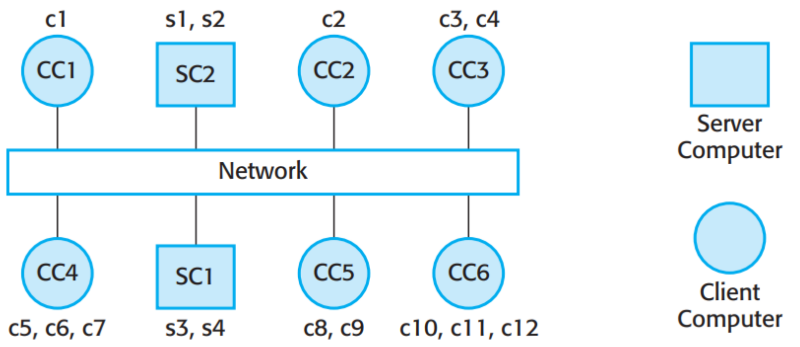

Figure 18.5 Mapping of clients and servers to networked computers

Figure shows client and server processes rather than processors. It is normal for several client processes to run on a single processor. For example, on your PC, you may run a mail client that downloads mail from a remote mail server. You may also run a web browser that interacts with a remote web server and a print client that sends documents to a remote printer.

Figure 18.5 illustrates the situation where the 12 logical clients are running on six computers. The four server processes are mapped onto two physical server computers. Several different server processes may run on the same processor but, often, servers are implemented as multiprocessor systems in which a separate instance of the server process runs on each machine.

Load-balancing software distributes requests for service from clients to different servers so that each server does the same amount of work. This allows a higher volume of transactions with clients to be handled, without degrading the response to individual clients.

Client–server systems depend on there being a clear separation between the presentation of information and the computations that create and process that information.

Figure 18.6 illustrates this model, showing an application structured into four layers:

• **A presentation layer** that is concerned with presenting information to the user and managing all user interaction;

• **A data management** layer that manages the data that is passed to and from the client. This layer may implement checks on the data, generate web pages, etc.;

• **An application processing** layer that is concerned with implementing the logic of the application and so providing the required functionality to end users;

• **A database layer** that stores the data and provides transaction management services, etc.

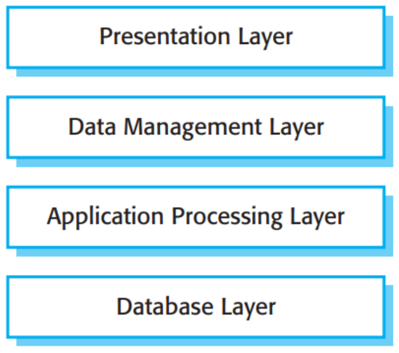

Figure 18.6 Layered architectural model for client–server application

Architectural Patterns for Distributed Systems

There is no universal model of system organization that is appropriate for all circumstances so various distributed architectural styles have emerged. When designing a distributed application, you should choose an architectural style that supports the critical non-functional requirements of your system. Five architectural styles are as follows:

1. Master-slave architecture, which is used in real-time systems in which guaranteed interaction response times are required.

2. Two-tier client–server architecture, which is used for simple client–server systems, and in situations where it is important to centralize the system for security reasons. In such cases, communication between the client and server is normally encrypted.

3. Multitier client–server architecture, which is used when there is a high volume of transactions to be processed by the server.

4. Distributed component architecture, which is used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client–server systems.

5. Peer-to-peer architecture, which is used when clients exchange locally stored information and the role of the server is to introduce clients to each other. It may also be used when a large number of independent computations may have to be made.

Master Slave Architecture

Master-slave architectures for distributed systems are commonly used in real- time systems where there may be separate processors associated with data acquisition from the system's environment, data processing, and computation and actuator management.

For example, an actuator may control a valve and change its state from 'open' to 'closed'. The 'master' process is usually responsible for computation, coordination, and communications and it controls the 'slave' processes.

'Slave' processes are dedicated to specific actions, such as the acquisition of data from an array of sensors. **Figure 18.7** illustrates this architectural model. It is a model of a traffic control system in a city and has three logical processes that run on separate processors.

The master process is the control room process, which communicates with separate slave processes that are responsible for collecting traffic data and managing the operation of traffic lights.

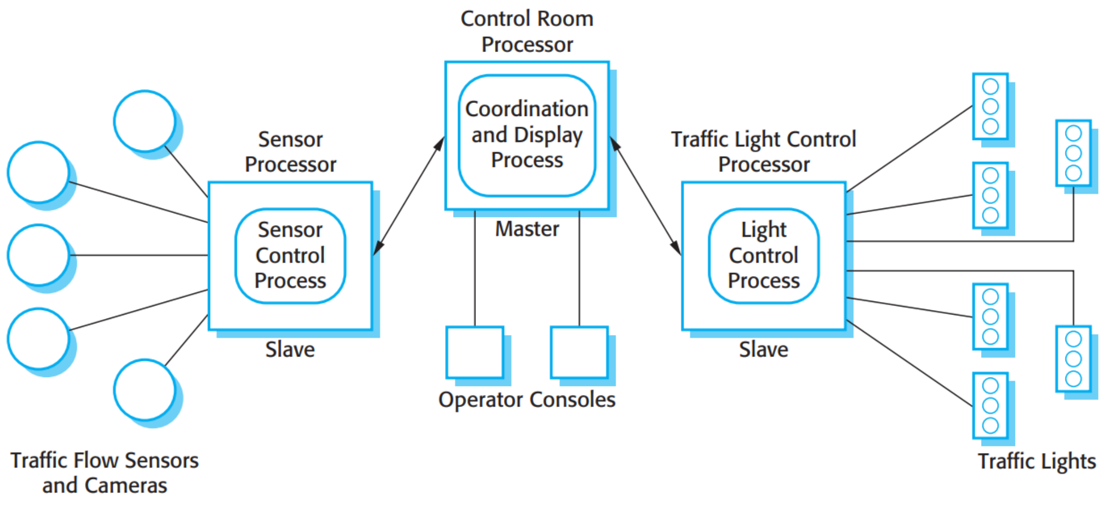

Figure 18.7 – A Traffic control system with Master Slave Architecture

Figure 18.7 illustrates this architectural model. It is a model of a traffic control system in a city and has three logical processes that run on separate processors. The master process is the control room process, which communicates with separate slave processes that are responsible for collecting traffic data and managing the operation of traffic lights.

A set of distributed sensors collects information on the traffic flow. The sensor control process polls the sensors periodically to capture the traffic flow information and collates this information for further processing.

The sensor processor is itself polled periodically for information by the master process that is concerned with displaying traffic status to operators, computing traffic light sequences and accepting operator commands to modify these sequences.

The control room system sends commands to a traffic light control process that converts these into signals to control the traffic light hardware. The master control room system is itself organized as a client–server system, with the client processes running on the operator's consoles.

Two tier Client Server Architecture

A two-tier client–server architecture is the simplest form of client–server architecture. The system is implemented as a single logical server plus an indefinite number of clients that use that server. This is illustrated **in Figure 18.8, which shows two forms of this architectural model:**

1**. A thin-client model**, where the presentation layer is implemented on the client and all other layers (data management, application processing, and database) are implemented on a server. The client software may be a specially written program on the client to handle presentation. More often, however, a web browser on the client computer is used for presentation of the data.

The advantage of the thin-client model is that it is simple to manage the clients. This is a major issue if there are a large number of clients, as it may be difficult and expensive to install new software on all of them. If a web browser is used as the client, there is no need to install any software.

The disadvantage of the thin-client approach, however is that it may place a heavy processing load on both the server and the network. The server is responsible for all computation and this may lead to the generation of significant network traffic between the client and the server

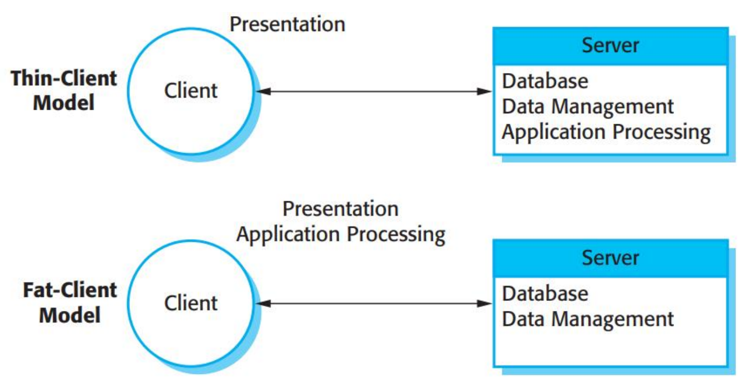

Figure 18.8 Thin- and fat-client architectural models

2. A fat-client model, where some or all of the application processing is carried out on the client. Data management and database functions are implemented on the server.

The fat-client model makes use of available processing power on the computer running the client software, and distributes some or all of the application processing and the presentation to the client. The server is essentially a transaction server that manages all database transactions.

Data management is straightforward as there is no need to manage the interaction between the client and the application processing system.

Of course, **the problem with the fat-client model** is that it requires additional system management to deploy and maintain the software on the client computer.

Multitier Client Server Architecture

The fundamental problem with a two-tier client–server approach is that the logical layers in the system—presentation, application processing, data management, and database—must be mapped onto two computer systems: the client and the server.

This may lead to problems with scalability and performance if the thin-client model is chosen, or problems of system management if the fat-client model is used. **To avoid some of these problems, a 'multi-tier client–server' architecture can be used.**

In this architecture, the different layers of the system, namely presentation, data management, application processing, and database, are separate processes that may execute on different processors

An Internet banking system **(Figure 18.10)** is an example of a multi-tier client–server architecture, where there are three tiers in the system. The bank's customer database (usually hosted on a mainframe computer as discussed above) provides database services.

A web server provides data management services such as web page generation and some application services. Application services such as facilities to transfer cash, generate statements, pay bills, and so on are implemented in the web server and as scripts that are executed by the client.

The user's own computer with an Internet browser is the client. This system is scalable because it is relatively easy to add servers (scale out) as the number of customers increase.

In this case, the use of a three-tier architecture allows the information transfer between the web server and the database server to be optimized. The communications between these systems can use fast, low-level data exchange protocols. Efficient middleware that supports database queries in SQL (Structured Query Language) is used to handle information retrieval from the database.

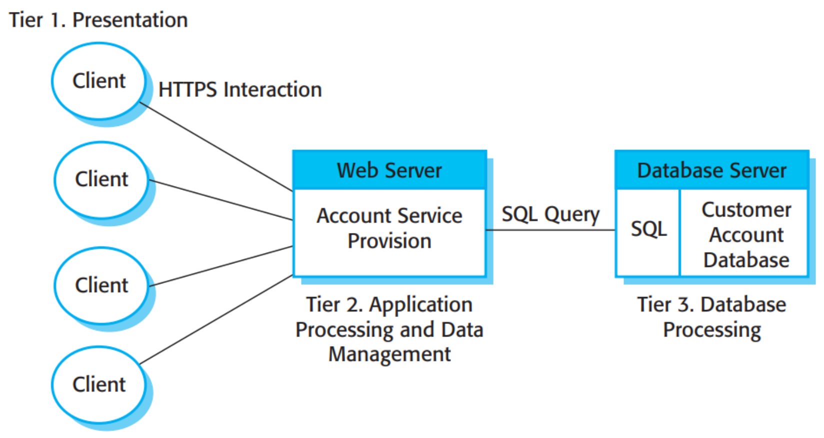
Tier 1. Presentation
Client
HTTPS Interaction
Client
Client
Client
Web Server
Account Service Provision
SQL Query
Database Server
SQL
Customer Account Database
Tier 2. Application Processing and Data Management
Tier 3. Database Processing

Figure 18.10 Three-tier architecture for an Internet banking system

Distributed component architectures

A more general approach to distributed system design is to design the system as a set of services, without attempting to allocate these services to layers in the system. Each service, or group of related services, is implemented using a separate component. In a distributed component architecture (Figure 18.12) the system is organized as a set of interacting components or objects.

These components provide an interface to a set of services that they provide. Other components call on these services through middleware, using remote procedure or method calls.

Distributed component systems are reliant on middleware, which manages component interactions, reconciles differences between types of the parameters passed between components, and provides a set of common services that application components can use.

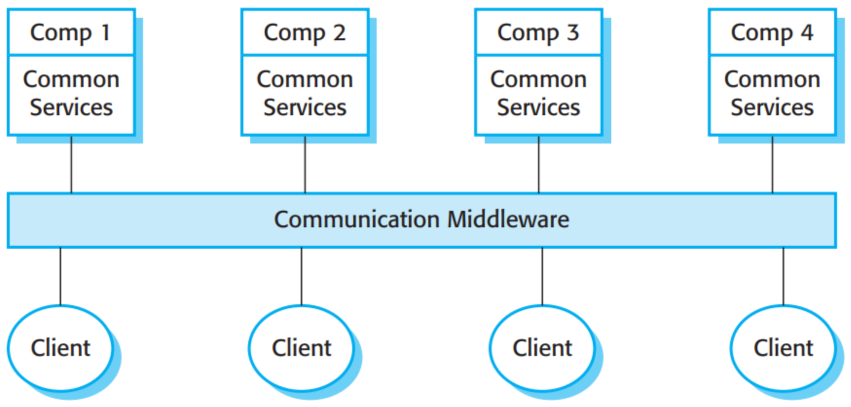

Figure 18.12 A distributed component architecture

The benefits of using a distributed component model for implementing distributed systems are the following:

1. It allows the system designer to delay decisions on where and how services should be provided. Service-providing components may execute on any node of the network. There is no need to decide in advance whether a service is part of a data management layer, an application layer, etc.

2. It is a very open system architecture that allows new resources to be added as required. New system services can be added easily without major disruption to the existing system.

3. The system is flexible and scalable. New components or replicated components can be added as the load on the system increases, without disrupting other parts of the system.

4. It is possible to reconfigure the system dynamically with components migrating across the network as required. This may be important where there are fluctuating patterns of demand on services. A service-providing component can migrate to the same processor as service-requesting objects, thus improving the performance of the system.

A distributed component architecture can be used as a logical model that allows you to structure and organize the system. In this case, you think about how to provide application functionality solely in terms of services and combinations of services.

You then work out how to provide these services using a set of distributed components. For example, in a retail application there may be application components concerned with stock control, customer communications, goods ordering, and so on.

Data mining systems are a good example of a type of system in which a distributed component architecture is the best architectural pattern to use. A data mining system looks for relationships between the data that is stored in a number of databases (Figure 18.13). Data mining systems usually pull in information from several separate databases, carry out computationally intensive processing, and display their results graphically.

An example of such a data mining application might be a system for a retail business that sells food and books. The marketing department wants to find relationships between a customer's food and book purchases.

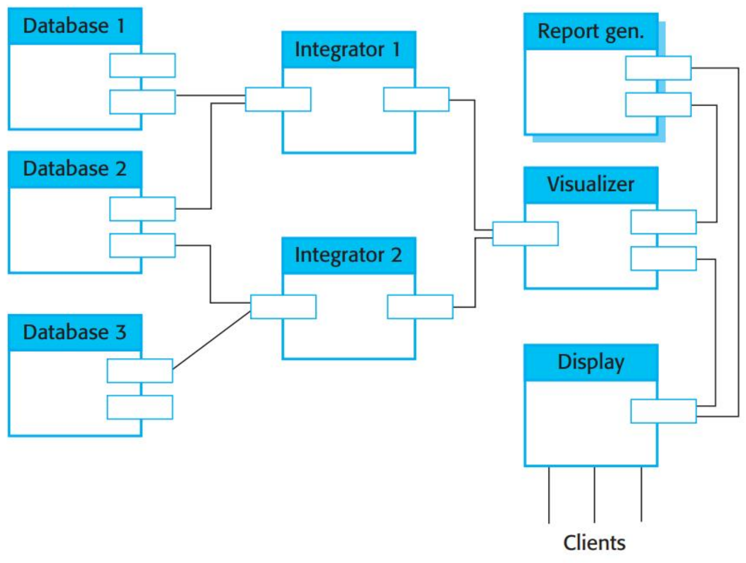

Figure 18.13 A distributed component architecture for a data mining system

Distributed component architectures suffer from two major disadvantages:

- They are more complex to design than client-server systems. Distributed component architectures are difficult for people to visualize and understand.
- Standardized middleware for distributed component systems has never been accepted by the community. Different vendors, such as Microsoft and Sun, have developed different, incompatible middleware.

As a result of these problems, service-oriented architectures are replacing distributed component architectures in many situations.

Peer to Peer Architecture

Peer-to-peer (p2p) systems are decentralized systems in which computations may be carried out by any node on the network. In principle at least, no distinctions are made between clients and servers.

In peer-to-peer applications, the overall system is designed to take advantage of the computational power and storage available across a potentially huge network of computers. The standards and protocols that enable communications across the nodes are embedded in the application itself and each node must run a copy of that application.

P2P architectures are used when

- A system is computationally-intensive and it is possible to separate the processing required into a large number of independent computations.
- A system primarily involves the exchange of information between individual computers on a network and there is no need for this information to be centrally-stored or managed.

In principle, every node in a p2p network could be aware of every other node. Nodes could connect to and exchange data directly with any other node in the network. In practice, of course, this is impossible, so nodes are organized into 'localities' with some nodes acting as bridges to other node localities.

Figure 18.14 shows this decentralized p2p architecture. In a decentralized architecture, the nodes in the network are not simply functional elements but are also communications switches that can route data and control signals from one node to another.

For example, assume that Figure 18.14 represents a decentralized, document-management system. This system is used by a consortium of researchers to share documents, and each member of the consortium maintains his or her own document store. However, when a document is retrieved, the node retrieving that document also makes it available to other nodes.

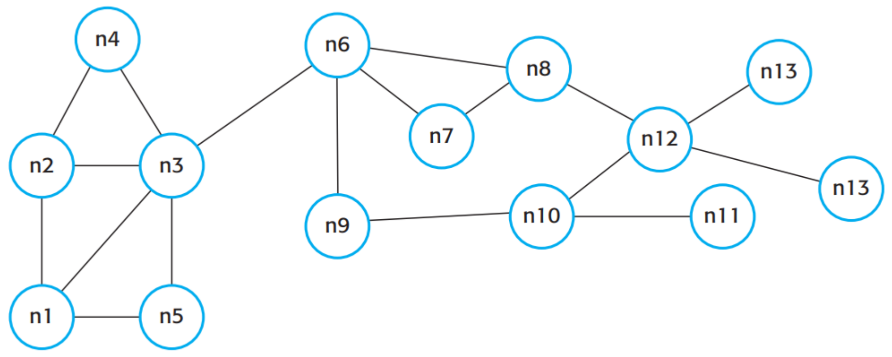

Figure 18.14
A decentralized p2p architecture

If someone needs a document that is stored somewhere on the network, they issue a search command, which is sent to nodes in their ‘locality’. These nodes check whether they have the document and, if so, return it to the requestor. If they do not have it, they route the search to other nodes. Therefore, if n1 issues a search for a document that is stored at n10, this search is routed through nodes n3, n6, and n9 to n10. When the document is finally discovered, the node holding the document then sends it to the requesting node directly by making a peer-to-peer connection.

An alternative p2p architectural model, which departs from a pure p2p architecture, is a semi centralized architecture where, within the network, one or more nodes act as servers to facilitate node communications. This reduces the amount of traffic between nodes.

Figure 18.15 illustrates this model. In a semi centralized architecture, the role of the server (sometimes called a super peer) is to help establish contact between peers in the network, or to coordinate the results of a computation. For example, if Figure 18.15 represents an instant messaging system, then network nodes communicate with the server (indicated by dashed lines) to find out what other nodes are available.

Once these nodes are discovered, direct communications can be established and the connection to the server is unnecessary. Therefore, nodes n2, n3, n5, and n6 are in direct communication.

In a computational p2p system, where a processor-intensive computation is distributed across a large number of nodes, it is normal for some nodes to be super peers. Their role is to distribute work to other nodes and to collate and check the results of the computation.

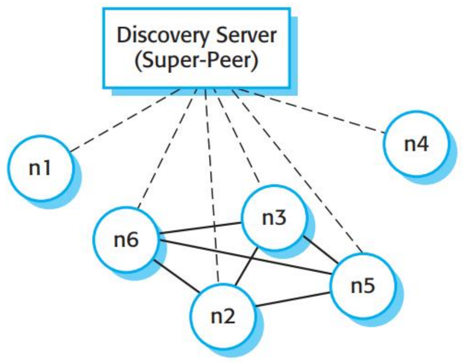

Figure 18.15
A semicentralized p2p architecture

Software as a Service (SaaS)

Software as a service (SaaS) involves hosting the software remotely and providing access to it over the Internet. Software is deployed on a server (or more commonly a number of servers) and is accessed through a web browser.

It is not deployed on a local PC. The software is owned and managed by a software provider, rather than the organizations using the software. Users may pay for the software according to the amount of use they make of it or through an annual or monthly subscription. Example: Google Docs. Key elements of SaaS include:

- Software is deployed on a server (or more commonly a number of servers) and is accessed through a web browser. It is not deployed on a local PC.
- The software is owned and managed by a software provider, rather than the organizations using the software.
- Users may pay for the software according to the amount of use they make of it or through an annual or monthly subscription. Sometimes, the software is free for anyone to use but users must then agree to accept advertisements, which fund the software service.

Software as a service (SaaS) and service-oriented architectures (SOA) are related, but they are not the same. Software as a service is a way of providing functionality on a remote server with client access through a web browser.

The server maintains the user's data and state during an interaction session. Transactions are usually long transactions e.g., editing a document. Service-oriented architecture is an approach to structuring a software system as a set of separate, stateless services. These may be provided by multiple providers and may be distributed. Typically, transactions are short transactions where a service is called, does something then returns a result.

When you are implementing SaaS, you have to take into account that you may have users of the software from several different organizations. You have to take three factors into account:

1. Configurability How do you configure the software for the specific requirements of each organization?

2. Multi-tenancy How do you present each user of the software with the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?

3. Scalability How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?