Parallel & Distributed System

Introduction to PDS

Md. Biplob Hosen

Lecturer, IIT-JU

Email: biplob.hosen@juniv.edu

Contents

- Middleware in DS
- DS Design Goals
- Types of DS

Reference Books

- Distributed Systems: Principles and Paradigms, 3rd Edition by Andrew S. Tanenbaum & Maarten van Steen, Publisher: Pearson Prentice Hall.

- Distributed Systems: Concepts and Design, 5th Edition by George Coulouris, Jean Dollimore & Tim Kindberg, Publisher: Addison-Wesley.

Middleware in DS

- To assist the development of distributed applications, distributed systems are often organized to have a separate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system.
- This organization is shown in Figure, leading to what is known as middleware.

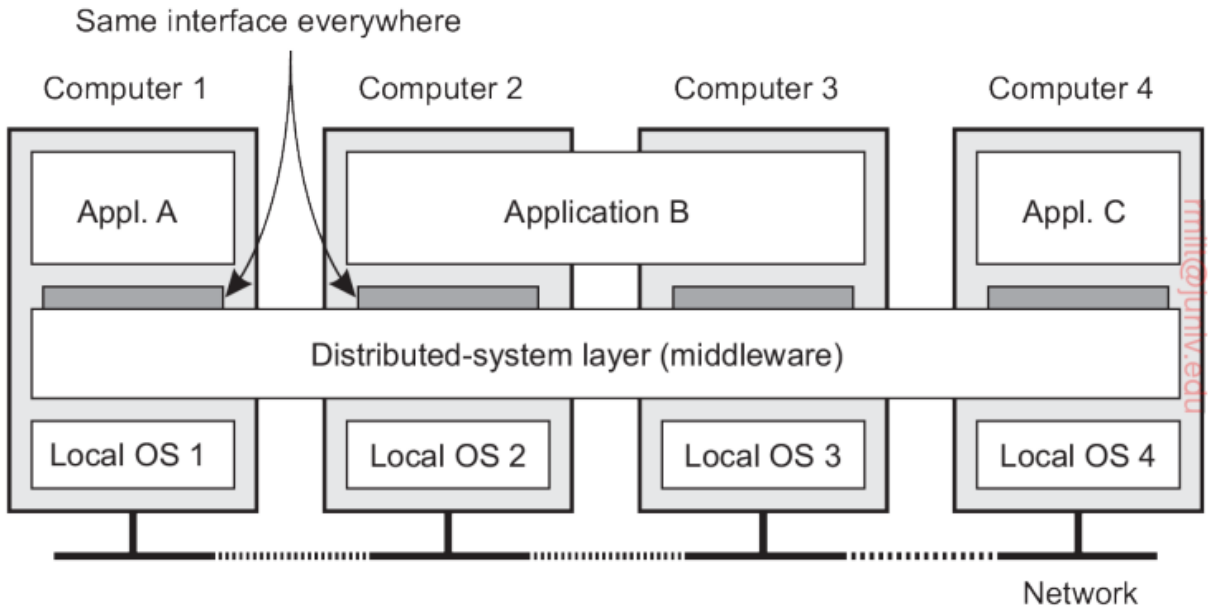


Figure 1.1: A distributed system organized in a middleware layer, which extends over multiple machines, offering each application the same interface.

Middleware in DS

- Figure shows four networked computers and three applications, of which application B is distributed across computers 2 and 3.
- Each application is offered the same interface. The distributed system provides the means for components of a single distributed application to communicate with each other, but also to let different applications communicate.
- At the same time, it hides, as best and reasonably as possible, the differences in hardware and operating systems from each application.
- In a sense, middleware is the same to a distributed system as what an operating system is to a computer: a manager of resources offering its applications to efficiently share and deploy those resources across a network.
- Next to resource management, it offers services that can also be found in most operating systems, including:
    1) Facilities for inter-application communication.
    2) Security services.
    3) Accounting services.
    4) Masking of and recovery from failures.

Middleware in DS

- The main difference with their operating-system equivalents, is that middleware services are offered in a networked environment.
- Note also that most services are useful to many applications. In this sense, middleware can also be viewed as a container of commonly used components and functions that now no longer have to be implemented by applications separately.
- To further illustrate these points, let us briefly consider a few examples of typical middleware services.

1) **Communication:** A common communication service is the so-called Remote Procedure Call (RPC). An RPC service allows an application to invoke a function that is implemented and executed on a remote computer as if it was locally available.

- To this end, a developer need merely specify the function header expressed in a special programming language, from which the RPC subsystem can then generate the necessary code that establishes remote invocations.

Middleware in DS

2) **Transactions:** Many applications make use of multiple services that are distributed among several computers. Middleware generally offers special support for executing such services in an all-or-nothing fashion, commonly referred to as an atomic transaction. In this case, the application developer need only specify the remote services involved, and by following a standardized protocol, the middleware makes sure that every service is invoked, or none at all.

3) **Service composition:** It is becoming increasingly common to develop new applications by taking existing programs and gluing them together. This is notably the case for many Web-based applications, in particular those known as Web services. Web-based middleware can help by standardizing the way Web services are accessed and providing the means to generate their functions in a specific order. A simple example of how service composition is deployed is formed by mashups: Web pages that combine and aggregate data from different sources. Well-known mashups are those based on Google maps in which maps are enhanced with extra information such as trip planners or real-time weather forecasts.

4) **Reliability:** As a last example, there has been a wealth of research on providing enhanced functions for building reliable distributed applications. The Horus toolkit allows a developer to build an application as a group of processes such that any message sent by one process is guaranteed to be received by all or no other process. As it turns out, such guarantees can greatly simplify developing distributed applications and are typically implemented as part of the middleware.

DS Design Goals

What do we want to achieve?

- Sharing of resources: Distributed system should make resources easily accessible.
- Distribution transparency: It should hide the fact that resources are distributed across a network.
- Openness: It should be open.
- Scalability: It should be scalable.

Sharing Resources

- An important goal of a distributed system is to make it easy for users to access and share remote resources.
- Resources can be virtually anything, but typical examples include peripherals, storage facilities, data, files, services, and networks, to name just a few.
- There are many reasons for wanting to share resources. One obvious reason is that of economics.
- For example, it is cheaper to have a single high-end reliable storage facility be shared than having to buy and maintain storage for each user separately.
- Connecting users and resources also makes it easier to collaborate and exchange information, as is illustrated by the success of the Internet with its simple protocols for exchanging files, mail, documents, audio, and video.
- The connectivity of the Internet has allowed geographically widely dispersed groups of people to work together by means of all kinds of groupware, that is, software for collaborative editing, teleconferencing, and so on, as is illustrated by multinational software-development companies that have outsourced much of their code production to Asia.

Distribution Transparency

- Distribution should be hidden from the user as much as possible.
- The concept of transparency can be applied to several aspects of a distributed system. The term object to mean either a process or a resource.

Transparency	Description
Access	Hide differences in data representation and how an object is accessed.
Location	Hide where an object is located.
Relocation	Hide that an object may be moved to another location while in use.
Migration	Hide that an object may move to another location.
Replication	Hide that an object is replicated.
Concurrency	Hide that an object may be shared by several independent users.
Failure	Hide the failure and recovery of an object.

Degree of Distribution Transparency

- Although distribution transparency is generally considered preferable for any distributed system, there are situations in which attempting to blindly hide all distribution aspects from users is not a good idea.
- A simple example is requesting your electronic newspaper to appear in your mailbox before 7 AM local time, as usual, while you are currently at the other end of the world living in a different time zone. Your morning paper will not be the morning paper you are used to.
- Likewise, a wide-area distributed system that connects a process in San Francisco to a process in Amsterdam cannot be expected to hide the fact that Mother Nature will not allow it to send a message from one process to the other in less than approximately 35 milliseconds. Practice shows that it actually takes several hundred milliseconds using a computer network. Signal transmission is not only limited by the speed of light, but also by limited processing capacities and delays in the intermediate switches.

Degree of Distribution Transparency

- There is also a trade-off between a high degree of transparency and the performance of a system. For example, many Internet applications repeatedly try to contact a server before finally giving up. Consequently, attempting to mask a transient server failure before trying another one may slow down the system as a whole. In such a case, it may have been better to give up earlier, or at least let the user cancel the attempts to make contact.
- Another example is where we need to guarantee that several replicas, located on different continents, must be consistent all the time. In other words, if one copy is changed, that change should be propagated to all copies before allowing any other operation. It is clear that a single update operation may now even take seconds to complete, something that cannot be hidden from users.

Degree of Distribution Transparency

- There are also other arguments against distribution transparency. Recognizing that full distribution transparency is simply impossible, we should ask ourselves whether it is even wise to pretend that we can achieve it. It may be much better to make distribution explicit so that the user and application developer are never tricked into believing that there is such a thing as transparency. The result will be that users will much better understand the behavior of a distributed system, and are thus much better prepared to deal with this behavior.
- The conclusion is that aiming for distribution transparency may be a nice goal when designing and implementing distributed systems, but that it should be considered together with other issues such as performance and comprehensibility. The price for achieving full transparency may be surprisingly high.

Being Open

- Another important goal of distributed systems is openness. An open distributed system is essentially a system that offers components that can easily be used by, or integrated into other systems.
- At the same time, an open distributed system itself will often consist of components that originate from elsewhere.
- **Interoperability** characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard.
- **Portability** characterizes to what extent an application developed for a distributed system A can be executed, without modification, on a different distributed system B that implements the same interfaces as A.
- Another important goal for an open distributed system is that it should be easy to configure the system out of different components (possibly from different developers). Also, it should be easy to add new components or replace existing ones without affecting those components that stay in place. In other words, an open distributed system should also be **extensible**.

Being Scalable

- Scalability of a system can be measured along at least three different dimensions:
- **First**, a system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system
- **Second**, a geographically scalable system is one in which the users and resources may lie far apart.
- **Third**, a system can be administratively scalable, meaning that it can still be easy to manage even if it spans many independent administrative organizations.
- Scalable distributed systems operate effectively and efficiently at many different scales, ranging from a small Intranet to the Internet.
- Scalable distributed systems remain effective when there is a significant increase in the number of resources and the number of users.

Being Scalable

- Challenges of designing scalable distributed systems are:

1) **Controlling the cost of physical resources:**

- Cost should linearly increase with the system size.
- For example, the frequency with which files are accessed in an intranet is likely to grow as the number of users and computer increases. It must be possible to add server computers to avoid performance bottleneck that would arise if a single file server had to handle all file access requests.

2) **Controlling the performance loss:**

- Consider a table with the correspondence between the domain names of computers and their IP address held by the DNS, which is used mainly to look up DNS names.
- Algorithm that use hierarchic structures scale better than those that use linear structures.
- But even in hierarchic structures an increase in size will result in some loss in performance.
- For example, in hierarchically structured data, search performance loss due to data growth but should not be beyond O(log n), where n is the size of data.
- For a system to be scalable, the maximum performance loss should not be more than that.

Being Scalable

3) **Preventing software resources running out:**

- An example is the numbers used as Internet addresses (IP)(32 bit->128-bit)

4) **Avoiding performance bottlenecks:**

- Using decentralized algorithms to avoid having performance bottlenecks.

Developing DS: Pitfalls

Observation

- Many distributed systems are needlessly complex caused by mistakes that required patching later on. Like:

1. The network is reliable
2. The network is secure
3. The network is homogeneous
4. The topology does not change
5. Latency is zero
6. Bandwidth is infinite
7. Transport cost is zero
8. There is one administrator

Types of DS

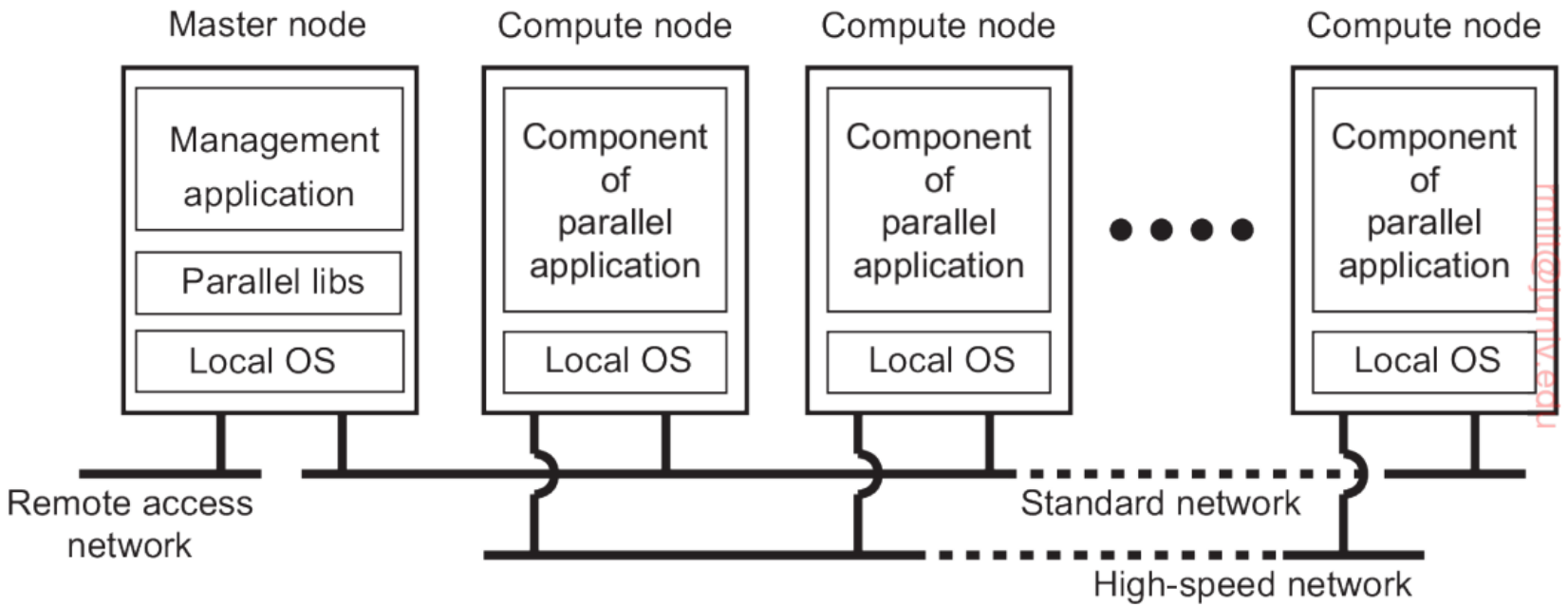
1. High performance distributed computing systems
2. Distributed information systems
3. Distributed systems for pervasive computing

High Performance DS

- In **cluster computing** the underlying hardware consists of a collection of similar workstations or PCs, closely connected by means of a high-speed local-area network. In addition, each node runs the same operating system.
- The situation becomes very different in the case of **grid computing**. This subgroup consists of distributed systems that are often constructed as a federation of computer systems, where each system may fall under a different administrative domain, and may be very different when it comes to hardware, software, and deployed network technology.
- From the perspective of grid computing, a next logical step is to simply
- outsource the entire infrastructure that is needed for compute-intensive applications. In essence, this is what **cloud computing** is all about: providing the facilities to dynamically construct an infrastructure and compose what is needed from available services. Unlike grid computing, which is strongly associated with high-performance computing, cloud computing is much more than just providing lots of resources.

Cluster Computing

- Cluster computing systems became popular when the price/performance ratio of personal computers and workstations improved.
- At a certain point, it became financially and technically attractive to build a supercomputer using off-the-shelf technology by simply hooking up a collection of relatively simple computers in a high-speed network.
- In virtually all cases, cluster computing is used for parallel programming in which a single (compute intensive) program is run in parallel on multiple machines.

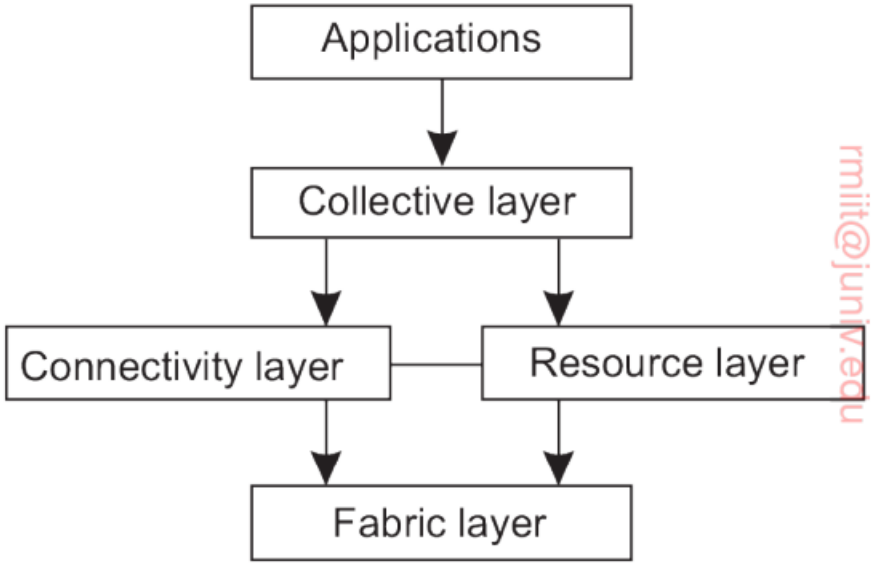


Grid Computing

- A characteristic feature of traditional cluster computing is its homogeneity.
- In most cases, the computers in a cluster are largely the same, have the same operating system, and are all connected through the same network.
- However, as we just discussed, there has been a trend towards more hybrid architectures in which nodes are specifically configured for certain tasks.
- This diversity is even more prevalent in grid-computing systems: no assumptions are made concerning similarity of hardware, operating systems, networks, administrative domains, security policies, etc.
- A key issue in a grid-computing system is that resources from different organizations are brought together to allow the collaboration of a group of people from different institutions, indeed forming a federation of systems.
- Such a collaboration is realized in the form of a virtual organization. The processes belonging to the same virtual organization have access rights to the resources that are provided to that organization. Typically, resources consist of compute servers (including supercomputers, possibly implemented as cluster computers), storage facilities, and databases.
- In addition, special networked devices such as telescopes, sensors, etc., can be provided as well.

Grid Computing

- The architecture consists of four layers.
- The lowest **fabric layer** provides interfaces to local resources at a specific site. Note that these interfaces are tailored to allow sharing of resources within a virtual organization. Typically, they will provide functions for querying the state and capabilities of a resource, along with functions for actual resource management (e.g., locking resources).



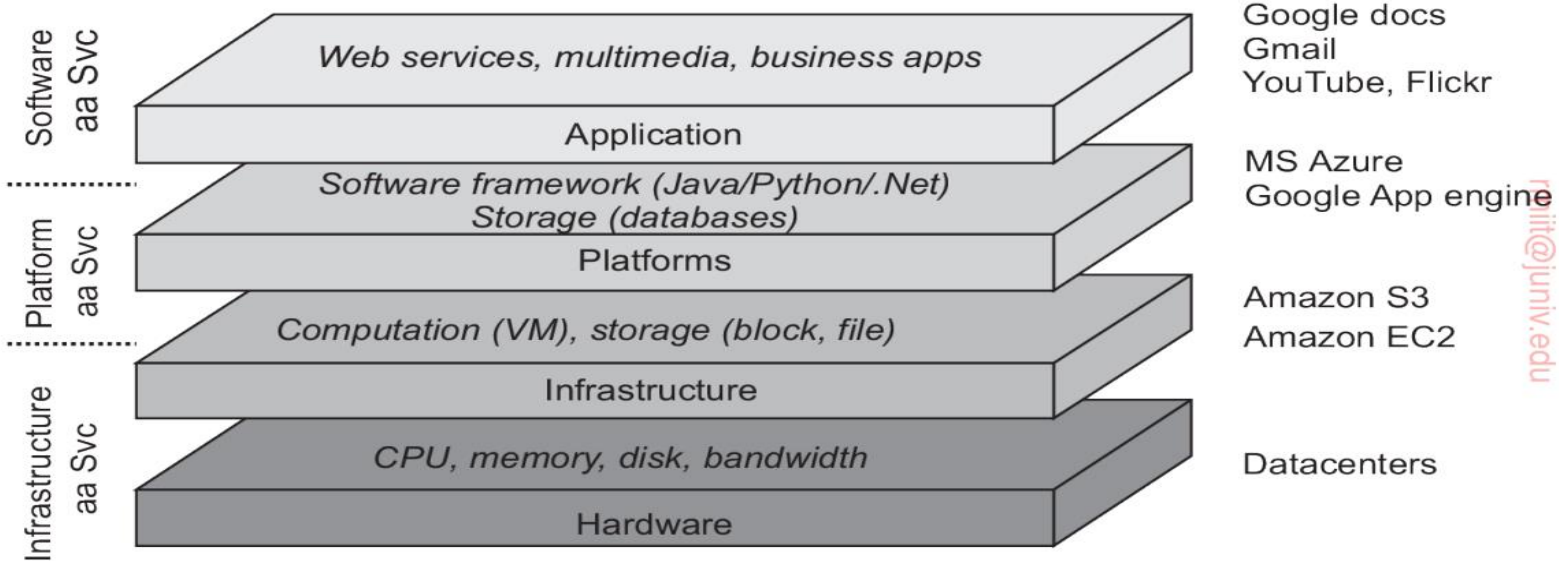
A layered architecture for grid computing systems.

Grid Computing

- The **connectivity layer** consists of communication protocols for supporting grid transactions that span the usage of multiple resources. For example, protocols are needed to transfer data between resources, or to simply access a resource from a remote location. In addition, the connectivity layer will contain security protocols to authenticate users and resources.
- The **resource layer** is responsible for managing a single resource. It uses the functions provided by the connectivity layer and calls directly the interfaces made available by the fabric layer. For example, this layer will offer functions for obtaining configuration information on a specific resource, or, in general, to perform specific operations such as creating a process or reading data. The resource layer is thus seen to be responsible for access control, and hence will rely on the authentication performed as part of the connectivity layer.
- The next layer in the hierarchy is the **collective layer**. It deals with handling access to multiple resources and typically consists of services for resource discovery, allocation and scheduling of tasks onto multiple resources, data replication, and so on. Unlike the connectivity and resource layer, each consisting of a relatively small, standard collection of protocols, the collective layer may consist of many different protocols reflecting the broad spectrum of services it may offer to a virtual organization.

Cloud Computing

- Cloud computing is characterized by an easily usable and accessible pool of virtualized resources.
- Which and how resources are used can be configured dynamically, providing the basis for scalability: if more work needs to be done, a customer can simply acquire more resources.
- The link to utility computing is formed by the fact that cloud computing is generally based on a pay-per-use model in which guarantees are offered by means of customized service-level agreements (SLAs).



Cloud Computing

- **Hardware:** The lowest layer is formed by the means to manage the necessary hardware: processors, routers, but also power and cooling systems. It is generally implemented at data centers and contains the resources that customers normally never get to see directly.
- **Infrastructure:** This is an important layer forming the backbone for most cloud computing platforms. It deploys virtualization techniques to provide customers an infrastructure consisting of virtual storage and computing resources. Indeed, nothing is what it seems: cloud computing evolves around allocating and managing virtual storage devices and virtual servers.
- **Platform:** One could argue that the platform layer provides to a cloud computing customer what an operating system provides to application developers, namely the means to easily develop and deploy applications that need to run in a cloud. In practice, an application developer is offered a vendor-specific API, which includes calls to uploading and executing a program in that vendor's cloud.

Cloud Computing

- **Application:** Actual applications run in this layer and are offered to users for further customization. Well-known examples include those found in office suites (text processors, spreadsheet applications, presentation applications, and so on). It is important to realize that these applications are again executed in the vendor's cloud. As before, they can be compared to the traditional suite of applications that are shipped when installing an operating system.
- Cloud-computing providers offer three different types of services:
- **Infrastructure-as-a-Service (IaaS)** covering the hardware and infrastructure layer.
- **Platform-as-a-Service (PaaS)** covering the platform layer.
- **Software-as-a-Service (SaaS)** in which their applications are covered.

Distributed Information Systems

- Distributed information systems (DIS) are designed to store, manage, and process information across multiple nodes or machines in a distributed environment.
- These systems distribute data and processing tasks to achieve scalability, fault tolerance, and high availability.
- Here are some key components and characteristics of distributed information systems:
- **Distributed Databases:** Distributed information systems often rely on distributed databases to store and manage data across multiple nodes. These databases can be horizontally partitioned, with each partition residing on different nodes, or replicated, where copies of data are stored on multiple nodes for redundancy and fault tolerance.
- **Replication and Consistency:** Replication is commonly used in distributed information systems to ensure data availability and fault tolerance. Replicated copies of data are stored on multiple nodes, and consistency protocols are employed to maintain data consistency across these replicas. Consistency models like eventual consistency or strong consistency are used depending on the system requirements.
- **Distributed File Systems:** Distributed file systems are used to manage and store large files across a cluster of machines. These systems provide a scalable and fault-tolerant way to store and access files, distributing the file blocks across multiple nodes for efficient storage and retrieval.

Distributed Information Systems

- **Distributed Query Processing:** Distributed information systems often involve executing queries that span multiple nodes or partitions. Distributed query processing involves optimizing query execution plans, partitioning data across nodes, and coordinating the execution of queries to efficiently retrieve and process data from distributed sources.
- **Fault Tolerance and Replication:** Distributed information systems incorporate fault tolerance mechanisms to ensure system reliability even in the presence of failures. This can include replication of data and services, failure detection and recovery mechanisms, and load balancing techniques to ensure that the system can continue operating despite individual node failures.
- **Distributed Coordination:** Distributed systems often require coordination among multiple nodes to achieve consistency, synchronization, and distributed transactions. Distributed coordination frameworks, such as Apache ZooKeeper, provide distributed locking, leader election, and coordination services to ensure proper synchronization and consistency in distributed environments.

Distributed Pervasive Computing

- Pervasive systems, also known as ubiquitous computing systems, refer to environments where computing technologies are seamlessly integrated into everyday life, becoming an inherent part of the physical environment and human activities.
- Pervasive systems aim to create an environment where computing is pervasive, unobtrusive, and transparent to users.
- These systems typically involve a network of interconnected devices, sensors, actuators, and services that work together to provide context-aware and personalized experiences.
- Here are some key characteristics and examples of pervasive systems:

1. Ubiquitous Connectivity: Pervasive systems rely on pervasive connectivity to facilitate communication and data exchange between devices and services. This includes wired and wireless networks, such as Wi-Fi, Bluetooth, and cellular networks, which enable devices to be interconnected and accessible from anywhere.
2. Sensor Networks: Pervasive systems often leverage sensor networks to gather contextual information about the environment. These sensors can include temperature sensors, motion sensors, light sensors, GPS modules, and more. Sensor data is used to understand the context, make informed decisions, and provide personalized services.

Distributed Pervasive Computing

3. Context Awareness: Pervasive systems aim to be context-aware by utilizing data from sensors and other sources to understand the user's context, such as location, activity, preferences, and social interactions. This context is used to adapt services, provide relevant information, and anticipate user needs.
4. Intelligent Adaptation: Pervasive systems employ intelligent algorithms and techniques to adapt to changing contexts and user requirements. They can dynamically adjust their behavior, interface, and services based on the current situation. For example, a smart home system may adjust lighting, temperature, and security settings based on the occupants' presence and preferences.
5. User Interaction: Pervasive systems focus on natural and intuitive user interaction, aiming to minimize the cognitive load required to interact with technology. Interaction mechanisms include voice commands, gestures, touch interfaces, and even context-aware automation where devices anticipate user needs and act accordingly.

Distributed Pervasive Computing

6. Security and Privacy: Pervasive systems face challenges related to security and privacy, as they deal with sensitive data and interconnected devices. Ensuring secure communication, data protection, and user privacy is crucial to maintain trust in these systems.

- Examples of pervasive systems include smart homes with interconnected devices, wearable fitness trackers, smart cities with sensor networks for monitoring and managing resources, and intelligent transportation systems that optimize traffic flow and provide real-time information to drivers.

Practice Tasks

- From the book of Coulouris-
  - **Exercise:** 1.1, 1.2, 1.3, 1.4, 1.5

Thank You ☺