Distributed System

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

• This definition has several important aspects. The first one is that a distributed system consists of components (i.e., computers) that are autonomous. A second aspect is that users (be they people or programs) think they are dealing with a single system. This means that one way or the otherthe autonomous components need to collaborate.

In order to support heterogeneous computers and networks while offering a single-system view, distributed systems are often organized by means of a layer of software-that is, logically placed between a higher-level layer consisting of users and applications, and a layer underneath consisting of operating systems and basic communication facilities, as shown in the figure below Accordingly, such a distributed system is sometimes called middleware.

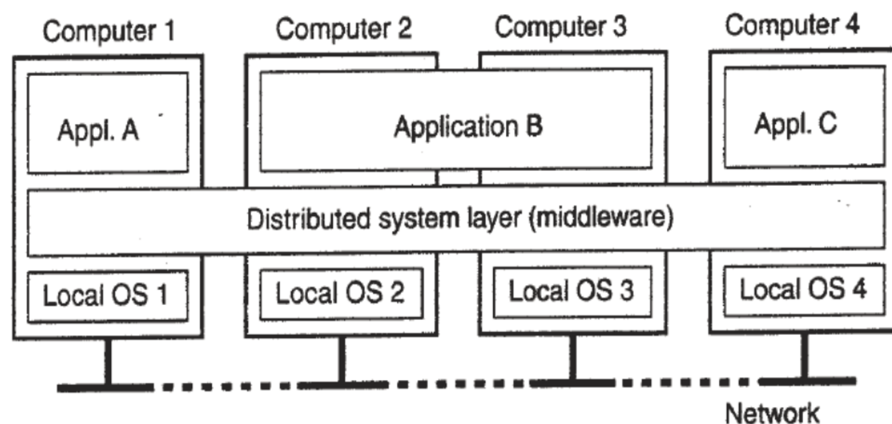


Figure I-I. A distributed system organized as middleware. The middleware layer extends over multiple machines, and offers each application the same interface.

Fig. 1-1 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 reasonable as possible, the differences in hardware and operating systems from each application.

Distributed Systems

Distributed systems goals

• A distributed system should make resources easily accessible; it should reasonably hide the fact that resources are distributed across a network; it should be open; and it should be scalable.

Making Resources Accessible

• The main goal of a distributed system is to make it easy for the users (and applications) to access remote resources, and to share them in a controlled and efficient way. Resources can be just about anything, but typical examples include things like printers, computers, storage facilities, data, files, Web pages, 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 let a printer be shared by several users in a small office than having to buy and maintain a separate printer for each user. Likewise, it makes economic sense to share costly resources such as supercomputers, high-performance storage systems, image setters, and other expensive peripherals.

Distribution Transparency

• An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers. A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent. Let us first take a look at what kinds of transparency exist in distributed systems. After that we will address the more general question whether transparency is always required.

Types of Transparency

The concept of transparency can be applied to several aspects of a distributed system, the most important ones shown in Fig. 1-2.

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

Figure 1-2. Different forms of transparency in a distributed system (ISO, 1995).

Degree of 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.

Openness

Another important goal of distributed systems is openness. An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services. For example, in computer networks, standard rules govern the format, contents, and meaning of messages sent and received. Such rules are formalized in protocols. In distributed systems, services are generally specified through interfaces, which are often described in an Interface Definition Language (IDL). Interface definitions written in an IDL nearly always capture only the syntax of services. In other words, they specify precisely the names of the functions that are available together with types of the parameters, return values, possible exceptions that can be raised, and so on.

What are openness talking about?

Be able to interact with services from other open systems.irrespective of the underlying environment:

- System should confirm to weel defined interfaces
- System should easily interoperate
- System should support portability of application
- System should be easily extensible

Separating Policy from Mechanism

To achieve flexibility in open distributed systems, it is crucial that the system is organized as a collection of relatively small and easily replaceable or adaptable components. This implies that we should provide definitions not only for the highest-level interfaces, that is, those seen by users and applications, but also definitions for interfaces to internal parts of the system and describe how those parts interact. This approach is relatively new. Many older and even contemporary systems are constructed using a monolithic approach in which components are.

Implementing openness :policies

- What level of consistency do we require for client cache data?
- What operations do we allow downloaded code to perform?
- Which QoS requirements do we adjust in the in the face of varying bandwidth?
- What level of secrecy do we require for communication?

Implementing openness : mechanisms

- Allow (dynamic) setting of caching policies.
- Support different level of trust for mobile code.
- Provide adjustable QoS parameters per data stream.
- Offer different encryption algorithm.

Scalability

- • Worldwide connectivity through the Internet is rapidly becoming as common as being able to send a postcard to anyone anywhere around the world. With this in mind, scalability is one of the most important design goals for developers of distributed systems.

Distributed Computing Systems

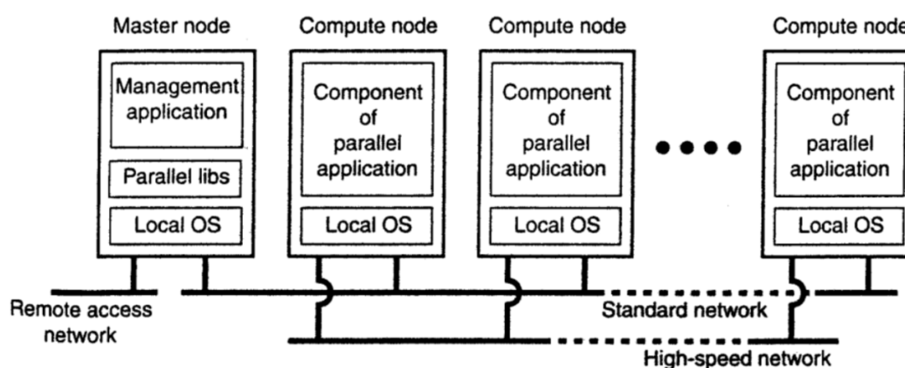
• An important class of distributed systems is the one used for high performance computing tasks. Roughly speaking, one can make a distinction between two subgroups. In cluster computing the underlying hardware consists of a collection of similar workstations or PCs, closely connected by means of a high-speed local-area network. In addition, each node runs the same operating system.

• The situation becomes quite 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.

Cluster Computing Systems

• 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 Systems

• A characteristic feature of cluster computing is its homogeneity. In most cases, the computers in a cluster are largely the same, they all have the same operating system, and are all connected through the same network. In contrast, grid computing systems have a high degree of heterogeneity: no assumptions are made concerning 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 or

institutions. Such a collaboration is realized in the form of a virtual organization.

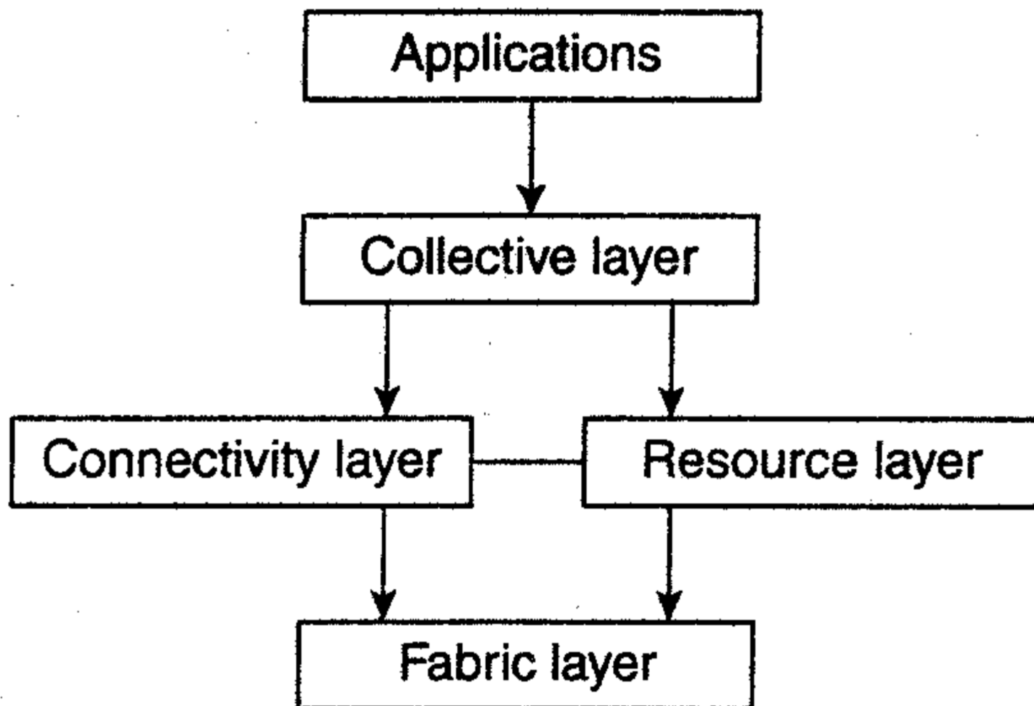


Figure :architecture for grid computing

The layers

- Fabric: provides interfaces to local resources (for querying state and capabilities, locking, etc.)
- Connectivity: communication / transaction protocols, e.g. ,for moving data between resources. Also various authentication protocols.

- Resource: manages a single resource, such as creating process or reading data.
- Collective: handle access to multiple resources: discovery, scheduling, replication.
- Application: contains actual grid applications in a single organization.

Distributed Information Systems

• Another important class of distributed systems is found in organizations that were confronted with a wealth of networked applications, but for which interoperability turned out to be a painful experience. Many of the existing middleware solutions are the result of working with an infrastructure in which it was easier to integrate applications into an enterprise-wide information system We can distinguish several levels at which integration took place.

. In many cases, a networked application simply consisted of a server running that application (often including a database) and making it available to remote programs, called clients. Such clients could send a request to the server for executing a specific operation, after which a response would be sent back. Integration at the lowest level would allow clients to wrap a number of requests, possibly for different servers, into a single larger request and have it executed as a distributed transaction. The key idea was that all, or none of the requests would be executed.

Distributed Information Systems

• Another important class of distributed systems is found in organizations that were confronted with a wealth of networked applications, but for which interoperability turned out to be a painful experience. Many of the existing middleware solutions are the result of working with an infrastructure in which it was easier to integrate applications into an enterprise-wide information system We can distinguish several levels at which integration took place.

. In many cases, a networked application simply consisted of a server running that application (often including a database) and making it available to remote programs, called clients. Such clients could send a request to the server for executing a specific operation, after which a response would be sent back. Integration at the lowest level would allow clients to wrap a number of requests, possibly for different servers, into a single larger request and have it executed as a distributed transaction. The key idea was that all, or none of the requests would be executed.

Examples of distributed information systems: Transaction Processing Systems

• To clarify our discussion, let us concentrate on database applications. In practice, operations on a database are usually carried out in the form of transactions. Programming using transactions requires special primitives that must either be supplied by the underlying distributed system or by the language runtime system. Typical examples of transaction primitives are shown in the figure below:

Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Transactions properties

1. Atomic: To the outside world, the transaction happens indivisibly.

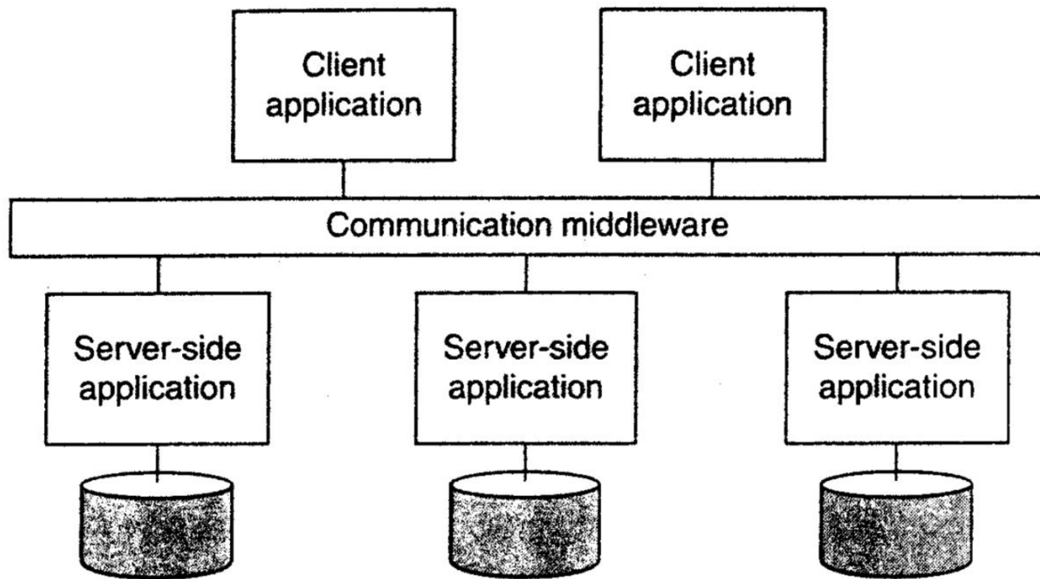
2. Consistent: The transaction does not violate system invariants.

3. Isolated: Concurrent transactions do not interfere with each other.

4. Durable: Once a transaction commits, the changes are permanent.

Enterprise Application Integration

• As mentioned, the more applications became decoupled from the databases they were built upon, the more evident it became that facilities were needed to integrate applications independent from their databases. In particular, application components should be able to communicate directly with each other and not merely by means of the request/reply behavior that was supported by transaction processing systems. This need for inter-application communication led to many different communication models, which we will discuss in detail in this book (and for which reason we shall keep it brief for now). The main idea was that existing applications could directly exchange information, as shown in the figure below:


Client application
Client application
Communication middleware
Server-side application
Server-side application
Server-side application

Distributed Pervasive Systems

• The distributed systems we have been discussing so far are largely characterized by their stability: nodes are fixed and have a more or less permanent and high-quality connection to a network. To a certain extent, this stability has been realized through the various techniques that are discussed in this book and which aim at achieving distribution transparency. For example, the wealth of techniques. for masking failures and recovery will give the impression that only occasionally things may go wrong. Likewise, we have been able to hide aspects related to the actual network location of a node, effectively allowing users and applications to believe that nodes stay put.

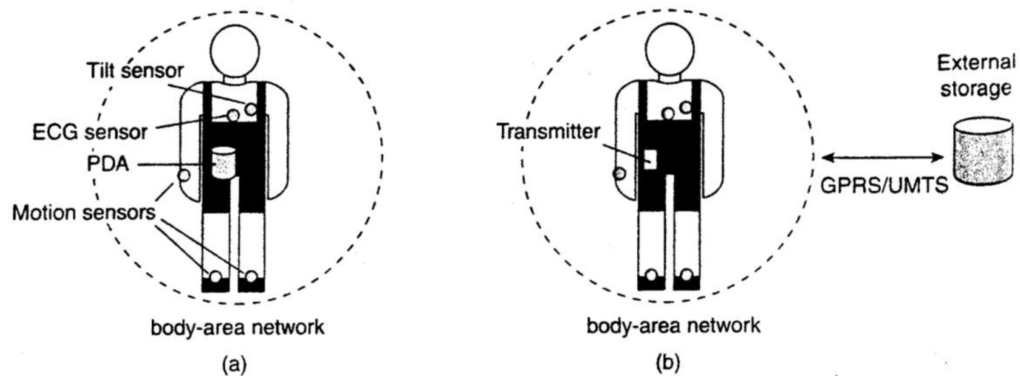
Home Systems

• An increasingly popular type of pervasive system, but which may perhaps be the least constrained, are systems built around home networks. These systems generally consist of one or more personal computers, but more importantly integrate typical consumer electronics such as TVs, audio and video equipment. Gaming devices, (smart) phones, PDAs, and other personal wearables into a single

system. In addition, we can expect that all kinds of devices such as kitchen appliances, surveillance cameras, clocks, controllers for lighting, and so on, will all be hooked up into a single distributed system.

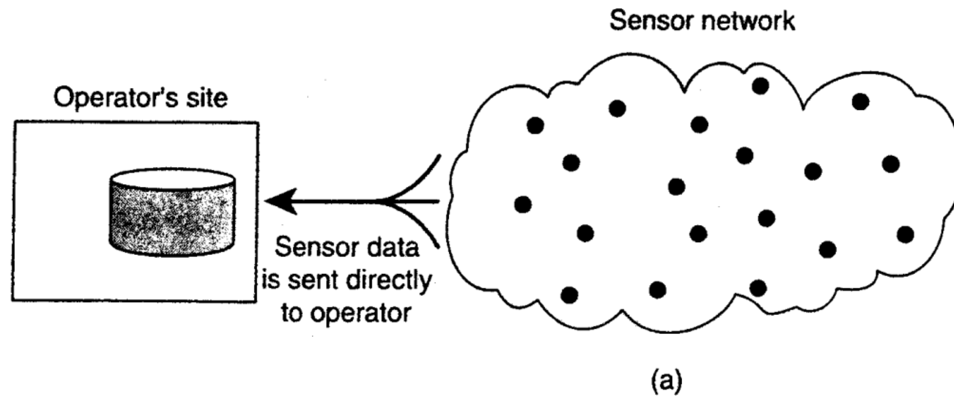
Electronic Health Care Systems

• Another important and upcoming class of pervasive systems are those related to (personal) electronic health care. With the increasing cost of medical treatment, new devices are being developed to monitor the wellbeing of individuals and to automatically contact physicians when needed. In many of these systems, a major goal is to prevent people from being hospitalized. Personal health care systems are often equipped with various sensors organized in a (preferably wireless) body-area network (BAN). An important issue is that such a network should at worst only minimally hinder a person. To this end, the network should be able to operate while a person is moving, with no strings (i.e., wires) attached to immobile devices. 8 Lecture 3- Distributed systems types ..

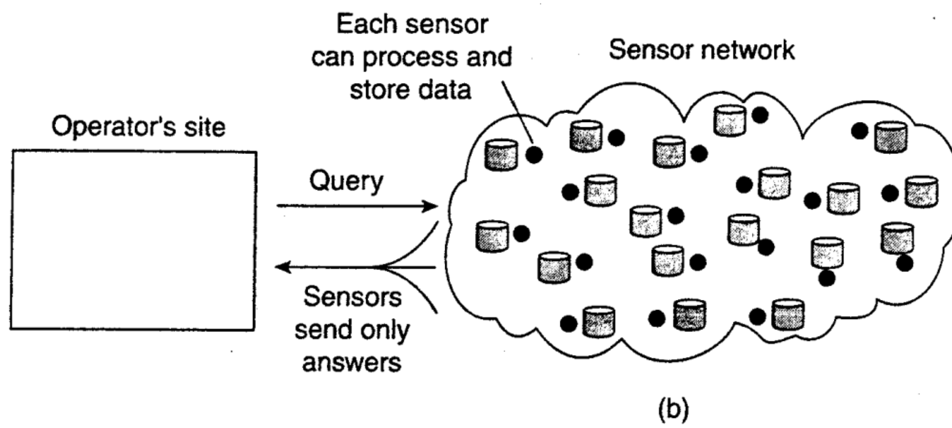


Sensor Networks

• Sensor network typically consists of tens to hundreds or thousands of relatively small nodes, each equipped with a sensing device. Most sensor networks use wireless communication, and the nodes are often battery powered. Their limited resources, restricted communication capabilities, and constrained power consumption demand that efficiency be high on the list of design criteria. The relation with distributed systems can be made clear by considering sensor networks as distributed databases.



(a)



(b)

Architectural Styles

• The style is formulated in terms of components, the way that components are connected to each other, the data exchanged between components. And finally how these elements are jointly configured into a system. A component is a modular unit with well-defined required and provided interfaces that is replaceable within its environment.

• Using components and connectors, we can come to various configurations, which, in turn have been classified into architectural styles. Several styles have by now been identified, of which the most important ones for distributed systems are:

– Layered architectures

– Object-based architectures

– Data-centered architectures

– Event-based architectures

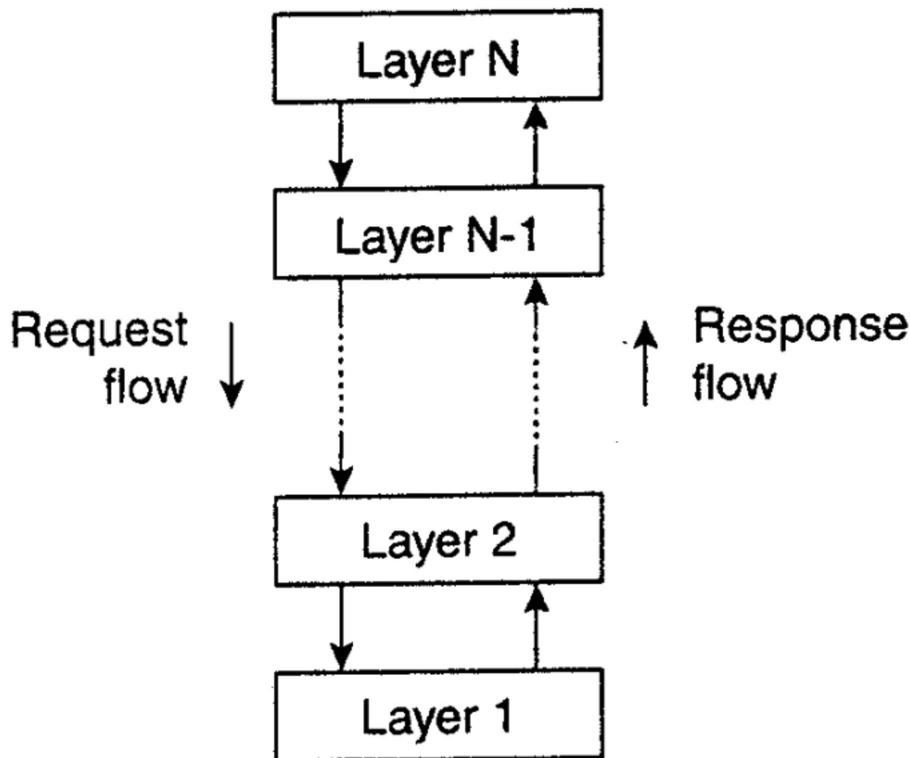
1. Layered Architecture:

In Layered architecture, different components are organised in layers. Each layer communicates with its adjacent layer by sending requests and getting responses. The layered architecture separates components into units. It is an

efficient way of communication. Any layer can not directly communicate with another layer. A layer can only communicate with its neighbouring layer and then the next layer transfers information to another layer and so on the process goes on.

In some cases, layered architecture is in cross-layer coordination. In a cross-layer, any adjacent layer can be skipped until it fulfils the request and provides better performance results. Request flow from top to bottom(downwards) and response flow from bottom to top(upwards). The advantage of layered architecture is that each layer can be modified independently without affecting the whole system. This type of architecture is used in Open System Interconnection (OSI) model.

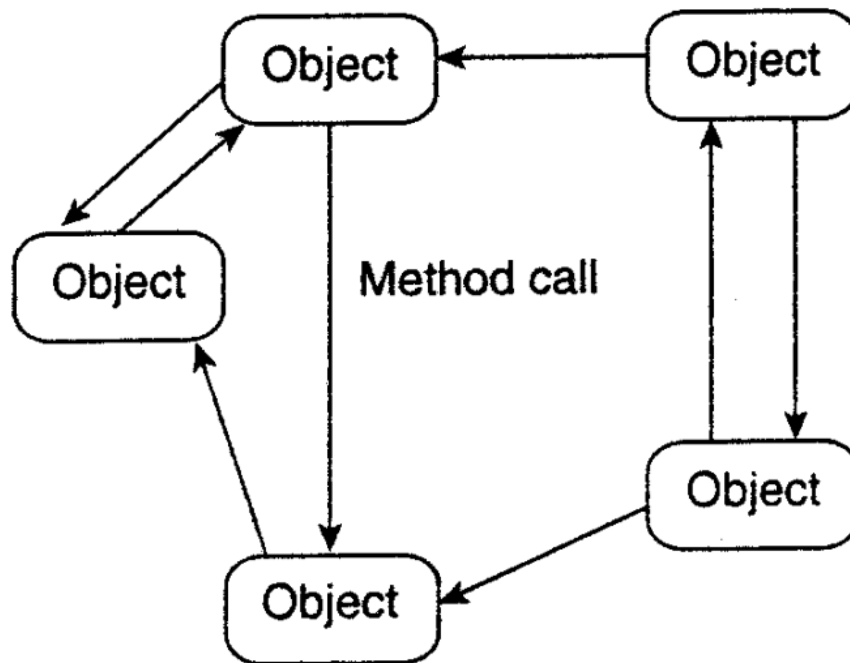
To the layers on top, the layers at the bottom offer a service. While the response is transmitted from bottom to top, the request is sent from top to bottom. This method has the advantage that calls always follow a predetermined path and that each layer is simple to replace or modify without affecting the architecture as a whole.



Object-Oriented Architecture:

In this type of architecture, components are treated as objects which convey information to each other. Object-Oriented Architecture contains an arrangement of loosely coupled objects. Objects can interact with each other through method calls. Objects are connected to each other through the Remote Procedure Call (RPC) mechanism or Remote Method Invocation (RMI) mechanism. Web Services and REST API are examples of object-oriented architecture. Invocations of methods are how objects communicate with one another. Typically, these are referred to as Remote Procedure Calls (RPC). REST API Calls, Web

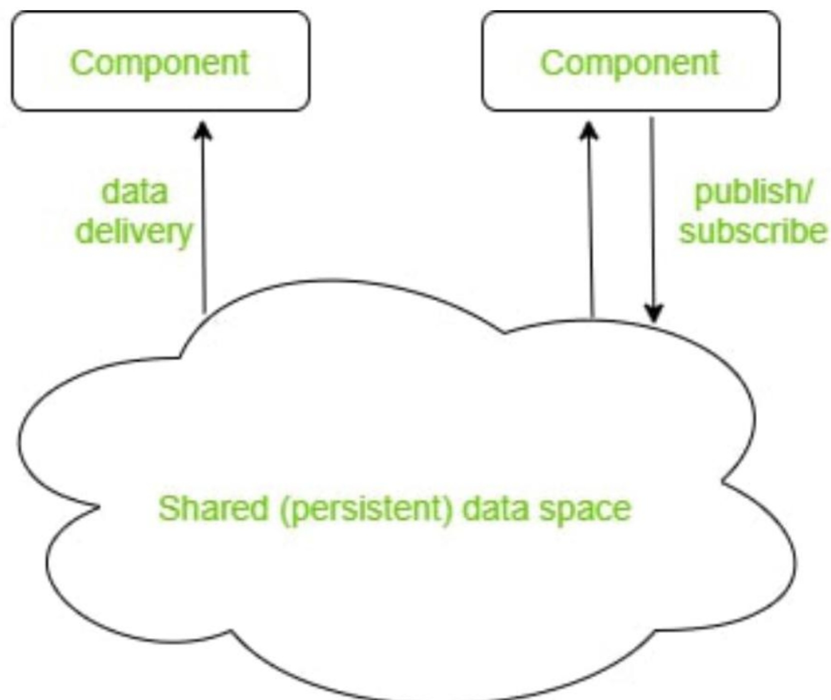
Services, and Java RMI are a few well-known examples. These characteristics apply to this.



Data Centered Architecture:

Data Centered Architecture is a type of architecture in which a common data space is present at the centre. It contains all the required data in one place a shared data space. All the components are connected to this data space and they follow publish/subscribe type of communication. It has a central data repository at the centre. Required data is then delivered to the components. Distributed file systems, producer-consumer systems, and web-based data services are a few well-known examples.

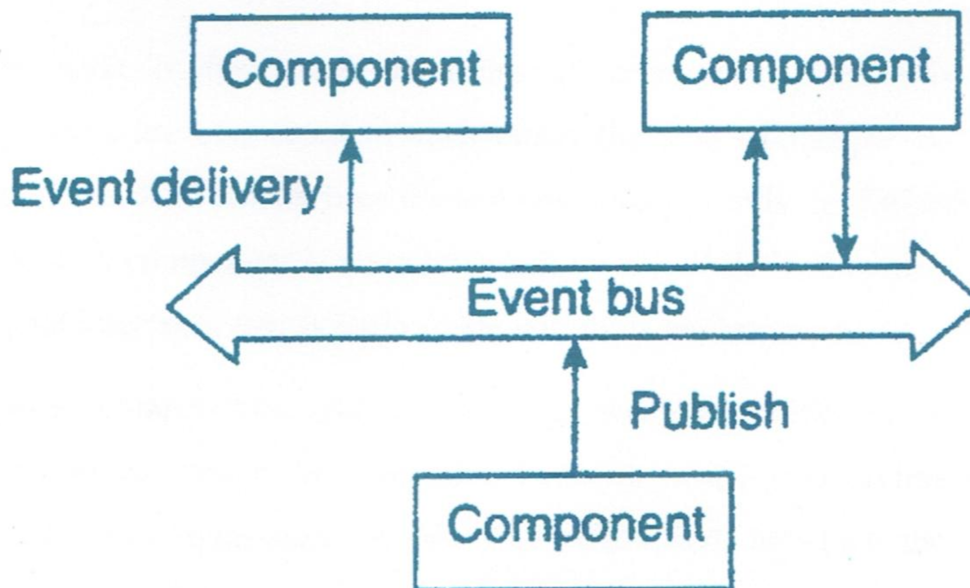
For example Producer-Consumer system. The producer produces data in common data space and consumers request data.



Event-Based Architecture:

Event-Based Architecture is almost similar to Data centered architecture just the difference is that in this architecture events are present instead of data. Events are present at the centre in the Event bus and delivered to the required component whenever needed. In this architecture, the entire communication is done through events. When an event occurs, the system, as well as the receiver, get notified. Data,

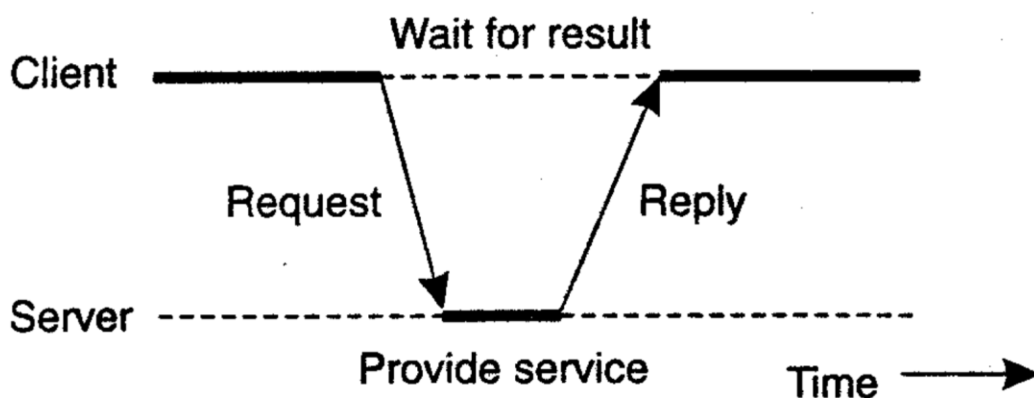
URLs etc are transmitted through events. The components of this system are loosely coupled that's why it is easy to add, remove and modify them. Heterogeneous components can communicate through the bus. One significant benefit is that these heterogeneous components can communicate with the bus using any protocol. However, a specific bus or an ESB has the ability to handle any kind of incoming request and respond appropriately.



Decentralized system:

• Despite the lack of consensus on many distributed systems issues, there is one issue that many researchers and practitioners agree upon: thinking in terms of clients that request services from servers helps us understand and manage the complexity of distributed systems and that is a good thing. In the basic client-server model, processes in a distributed system are divided into two (possibly overlapping) groups.

• A server is a process implementing a specific service, for example, a file system service or a database service. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply. This client-server interaction, also known as request reply behavior is shown in Fig. 2-3.



Decentralized Architecture

• Multitier client-server architectures are a direct consequence of dividing applications into a user-interface, processing components, and a data level. The different tiers correspond directly with the logical organization of applications. In many business environments, distributed processing is equivalent to organizing a client-server application as a multitiered architecture. We refer to this type of distribution as vertical distribution. The characteristic feature of vertical distribution is that it is achieved by placing logically different components on different machines.

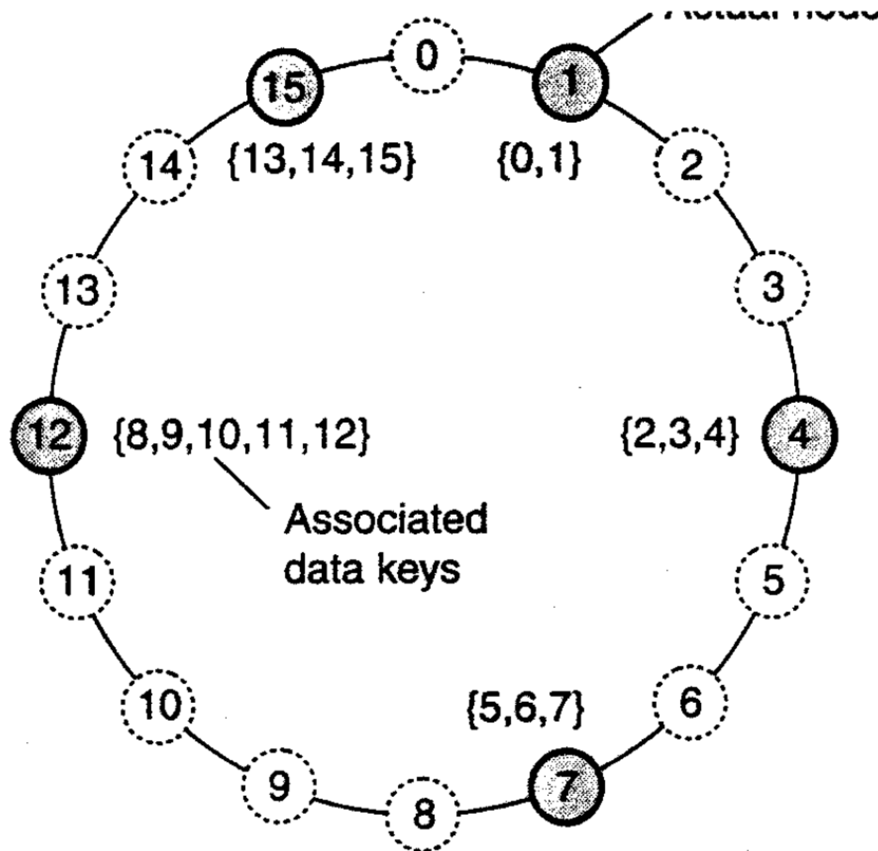
• In a structured peer-to-peer architecture, the overlay network is constructed using a deterministic procedure. By far the most-used procedure is to organize the processes through a distributed hash table (DHT). In a DHT -based system, data items are assigned a random key from a large identifier space, such as a 128-bit or 160-bit identifier.

Likewise, nodes in the system are also assigned a random number from the same identifier space.

• The crux of every DHT-based system is then to implement an efficient and deterministic scheme that uniquely maps the key of a data item to the identifier of a node based on some distance metric. Most importantly, when looking up a data item, the network address of the node responsible for that data item is returned. Effectively, this is accomplished by routing a request for a data item to the responsible node.

• For example, in the Chord system the nodes are logically organized in a ring such that a data item with key k is mapped to the node with the smallest identifier id ~ k. This node is referred to as the successor of key k and denoted as succ(k), as shown in Fig. 2-7. To actually look up the data item, an application running on an arbitrary node would then call the function LOOKUP(k).

• which would subsequently return the network address of succ(k). At that point, the application can contact the node to obtain a copy of the data item.



The mapping of data items onto nodes in Chord

If we concentrate on how nodes organize themselves into an overlay network, or, in other words, membership management. In the following, it is important to realize that looking up a key does not follow the logical organization of nodes in the ring from Fig. 2-7. Rather, each node will maintain shortcuts to other nodes in such a way that lookups can generally be done in O(log (N) number of steps, where N is the number of nodes participating in the overlay.

Peer-to- Peer Architectures

• Unstructured peer-to-peer systems largely rely on randomized algorithms for constructing an overlay network. The main idea is that each node maintains a list of neighbors, but that this list is constructed in a more or less random way. Likewise, data items are assumed to be randomly placed on nodes. As a consequence, when a node needs to locate a specific data item, the only thing it can effectively do is flood the network with a search query. • One of the goals of many unstructured peer-to-peer systems is to construct an overlay network that resembles a random graph. The basic model is that each node maintains a list of c neighbors, where, ideally, each of these neighbors represents a randomly chosen live node from the current set of nodes. The list of neighbors is also referred to as a partial view. There are many ways to construct such a partial view.

perpeers

• Notably in unstructured peer-to-peer systems, locating relevant data items can become problematic as the network grows. The reason for this scalability problem is simple: as there is no deterministic way of routing a lookup request to a

specific data item, essentially the only technique a node can resort to is flooding the request. There are various ways in which flooding can be dammed, but as an alternative many peer-to-peer systems have proposed to make use of special nodes that maintain an index of data items. There are other situations in which abandoning the symmetric nature of peer-to-peer systems is sensible. Consider a collaboration of nodes that offer resources.