Unit 2: Architecture

Contents:

2.1 Architectural Styles

The organization of distributed systems is mostly about the software components that constitute the system. These software architectures tell us how the various software components are to be organized and how they should interact.

An important goal of distributed systems is to separate applications from underlying platforms by providing a middleware layer. Adopting such a layer is an important architectural decision, and its main purpose is to provide distribution transparency.

The actual realization of a distributed system requires that we instantiate and place software components on real machines. There are many different choices that can be made in doing so. The final instantiation of a software architecture is also referred to as a system architecture.

The logical organization of a distributed system into software components, also referred to as its software architecture is crucial for the successful development of large software systems.

The notion of an architectural style is important. It 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. A connector, is generally described as a mechanism that mediates communication, coordination, or cooperation among components. 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
- Resource-centered architectures
- Publish-subscribe architectures

Layered architectures:

The basic idea for the layered style is simple: components are organized in a layered fashion where a component at layer Lj can make a downcall to a component at a lower-level layer Li (with i < j) and generally expects a response. Only in exceptional cases will an upcall be made to a higher-level component. The three common cases are shown in Figure below:

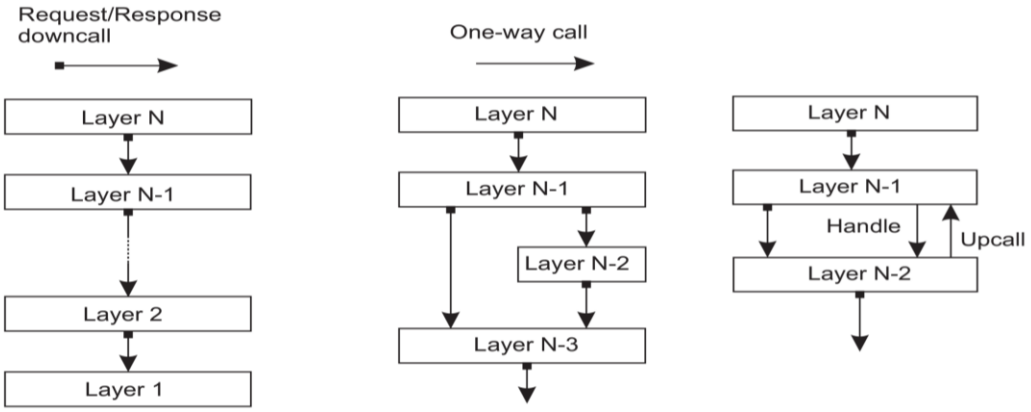


Figure: Pure layered organization, Mixed layered organization and Layered organization with Upcall

The first figure (Pure layered organization) shows a standard organization in which only downcalls to the next lower layer are made. This organization is commonly deployed in the case of network communication.

In many situations we also encounter the organization shown in second figure (Mixed layered organization). Consider, for example, an application A that makes use of a library LOS to interface to an operating system. At the same time, the application uses a specialized mathematical library Lmath that has been implemented by also making use of LOS. In this case, A is implemented at layer N − 1, Lmath at layer N − 2, and LOS which is common to both of them, at layer N − 3.

Finally, a special situation is shown in third figure (Layered organization with Upcall). In some cases, it is convenient to have a lower layer do an upcall to its next higher layer. A typical example is when an operating system signals the occurrence of an event, to which end it calls a user-defined operation for which an application had previously passed a reference (typically referred to as a handle).

Layered communication protocols

A well-known and ubiquitously applied layered architecture is that of so called communication-protocol stacks. In communication-protocol stacks, each layer implements one or several communication services allowing data to be sent from a destination to one or several targets. To this end, each layer offers an interface specifying the functions that can be called.

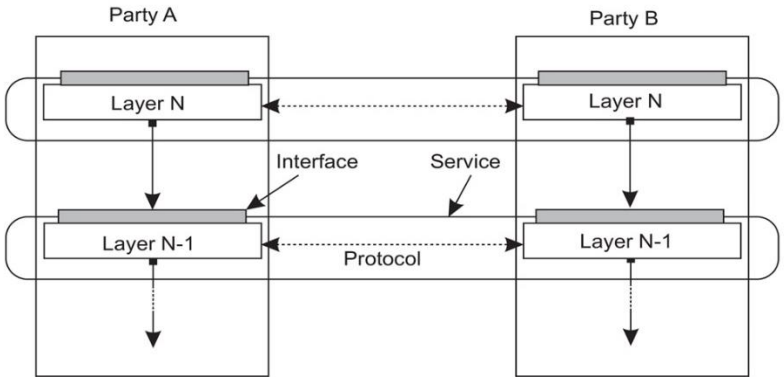


Figure: A layered communication-protocol stack, showing the difference between a service, its interface and the protocol it deploys.

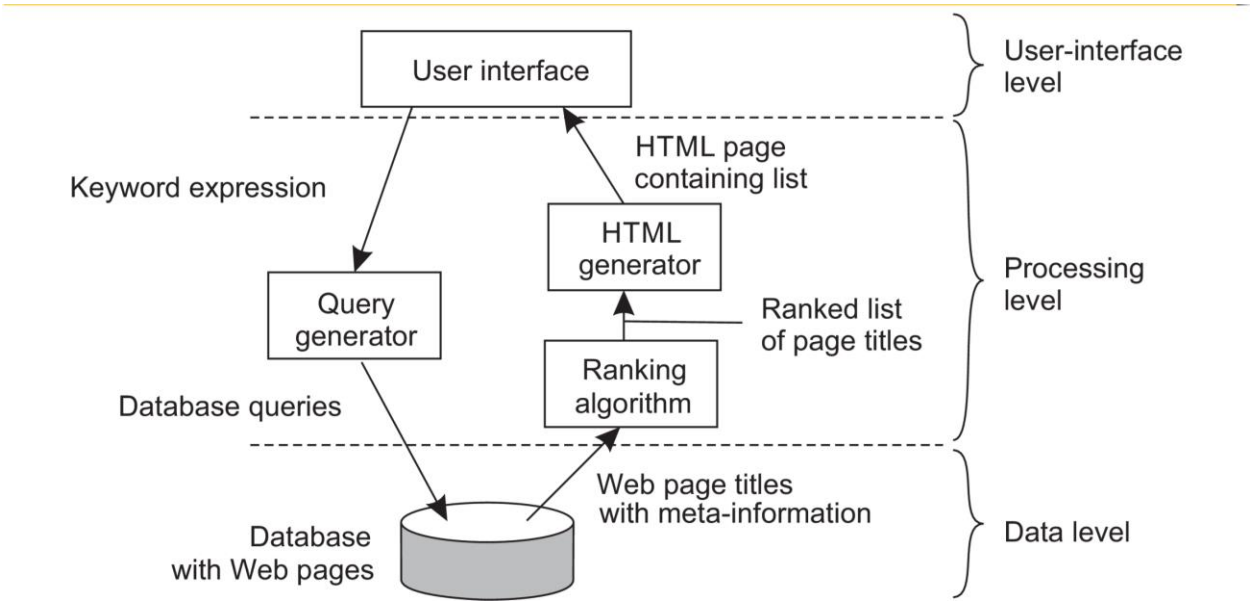
Application Layering

Considering that a large class of distributed applications is targeted toward supporting user or application access to databases, many people have advocated a distinction between three logical levels, essentially following a layered architectural style:

1. The application-interface level
2. The processing level
3. The data level

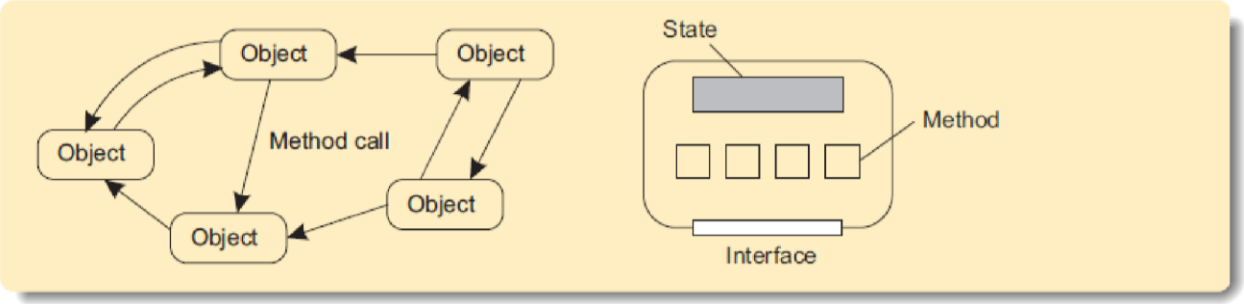
In line with this layering, we see that applications can often be constructed from roughly three different pieces: a part that handles interaction with a user or some external application, a part that operates on a database or file system, and a middle part that generally contains the core functionality of the application. This middle part is logically placed at the processing level. In contrast to user interfaces and databases, there are not many aspects common to the processing level.

For example, consider an Internet search engine. Ignoring all the animated banners, images, and other fancy window dressing, the user interface of a search engine can be very simple: a user types in a string of keywords and is subsequently presented with a list of titles of Web pages. The back end is formed by a huge database of Web pages that have been pre-fetched and indexed. The core of the search engine is a program that transforms the user's string of keywords into one or more database queries. It subsequently ranks the results into a list, and transforms that list into a series of HTML pages. This information retrieval part is typically placed at the processing level.



Object-based and service-oriented architectures

A far more loose organization is followed in object-based architectures. Each object corresponds to what we have defined as a component, and these components are connected through a procedure call mechanism. In the case of distributed systems, a procedure call can also take place over a network, that is, the calling object need not be executed on the same machine as the called object.



Object-based architectures are attractive because they provide a natural way of encapsulating data (called an object's state) and the operations that can be performed on that data (which are referred to as an object's methods) into a single entity. The interface offered by an object conceals implementation details, essentially meaning that we, in principle, can consider an object completely independent of its environment. As with components, this also means that if the interface is clearly defined and left otherwise untouched, an object should be replaceable with one having exactly the same interface. This separation between interfaces and the objects implementing these interfaces allows us to place an interface at one machine, while the object itself resides on another machine.

Resource-based architectures

As an increasing number of services became available over the Web and the development of distributed systems through service composition became more important, researchers started to rethink the architecture of mostly Web-based distributed systems. One of the problems with service composition is that connecting various components can easily turn into an integration nightmare.

As an alternative, one can also view a distributed system as a huge collection of resources that are individually managed by components. Resources may be added or removed by (remote) applications, and likewise can be retrieved or modified. This approach has now been widely adopted for the Web and is known as Representational State Transfer (REST). There are four key characteristics of what are known as RESTful:

1. Resources are identified through a single naming scheme
2. All services offer the same interface, consisting of at most four operations:

Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource in some representation
DELETE	Delete a resource
POST	Modify a resource by transferring a new state

3. Messages sent to or from a service are fully self-described.
4. After executing an operation at a service, that component forgets everything about the caller is also referred to as a stateless execution.

Publish-subscribe architectures

As systems continue to grow and processes can more easily join or leave, it becomes important to have an architecture in which dependencies between processes become as loose as possible. A large class of distributed systems have adopted an architecture in which there is a strong separation between processing and coordination. The idea is to view a system as a collection of autonomously operating processes.

In this model, coordination encompasses the communication and cooperation between processes. It forms the glue that binds the activities performed by processes into a whole.

Cabri et al. provide a taxonomy of coordination models that can be applied equally to many types of distributed systems. Slightly adapting their terminology, we make a distinction between models along two different dimensions, temporal and referential.

	Temporally coupled	Temporally decoupled
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

When processes are temporally and referentially coupled, coordination takes place in a direct way, referred to as **direct coordination**. The referential coupling generally appears in the form of explicit referencing in communication. For example, a process can communicate only if it knows the name or identifier of the other processes it wants to exchange information with. Temporal coupling means that processes that are communicating will both have to be up and running. In real life, talking over cell phones (and assuming that a cell phone has only one owner), is an example of direct communication.

A different type of coordination occurs when processes are temporally decoupled, but referentially coupled, which we refer to as mailbox coordination. In this case, there is no need for two communicating processes to be executing at the same time in order to let communication take place. Instead, communication takes place by putting messages in a (possibly shared) mailbox. Because it is necessary to explicitly address the mailbox that will hold the messages that are to be exchanged, there is a referential coupling.

The combination of referentially decoupled and temporally coupled systems form the group of models for event-based coordination. In referentially decoupled systems, processes do not know each other explicitly. The only thing a process can do is publish a notification describing the occurrence of an event (e.g., that it wants to coordinate activities, or that it just produced some interesting results). Assuming that notifications come in all sorts and kinds, processes may subscribe to a specific kind of notification. In an ideal event-based coordination model, a published notification will be delivered exactly to those processes that have subscribed to it. However, it is generally required that the subscriber is up-and-running at the time the notification was published.

2.2 Middleware organization

A middleware is a component responsible for communicating and coordinating the heterogeneous components like hardware, software, protocol, communication media used in the distributed system. It is independent of the overall organization of a distributed system or application. In particular, there are two important types of design patterns that are often applied to the organization of middleware:

1. wrappers and
2. interceptors

Each targets different problems, yet addresses the same goal for middleware: achieving openness.

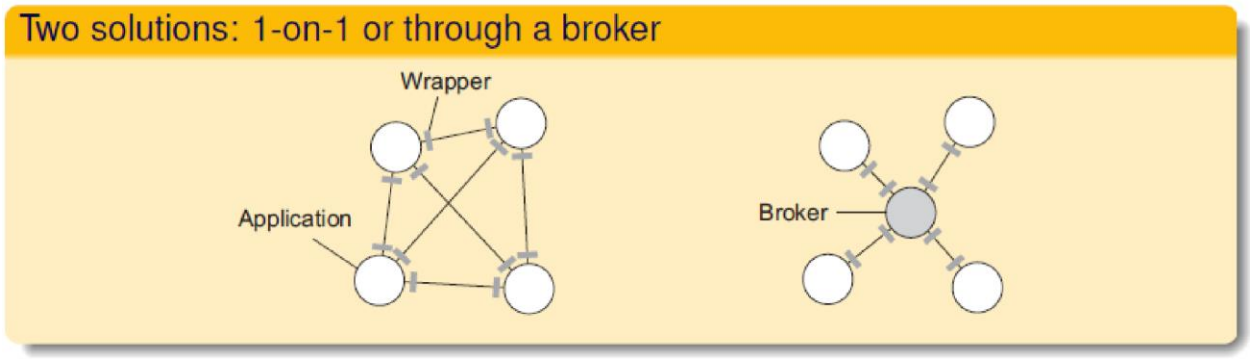
Wrappers

A wrapper or adapter is a special component that offers an interface acceptable to a client application, of which the functions are transformed into those available at the component. In essence, it solves the problem of incompatible interfaces.

Although originally narrowly defined in the context of object-oriented programming, in the context of distributed systems wrappers are much more than simple interface transformers. For example, an object adapter is a component that allows applications to invoke remote objects, although those objects may have been implemented as a combination of library functions operating on the tables of a relational database.

Wrappers have always played an important role in extending systems with existing components. Extensibility, which is crucial for achieving openness, used to be addressed by adding wrappers as needed. In other words, if application A managed data that was needed by application B, one approach would be to develop a wrapper specific for B so that it could have access to A's data. Clearly, this approach does not scale well: with N applications we would, in theory, need to develop $N \times (N - 1) = O(N^2)$ wrappers.

Again, facilitating a reduction of the number of wrappers is typically done through middleware. One way of doing this is implementing a so called **broker**, which is logically a centralized component that handles all the accesses between different applications. In the case of a message broker, applications simply send requests to the broker containing information on what they need. The broker, having knowledge of all relevant applications, contacts the appropriate applications, possibly combines and transforms the responses and returns the result to the initial application. In principle, because a broker offers a single interface to each application, we now need at most $2N = O(N)$ wrappers instead of $O(N^2)$.



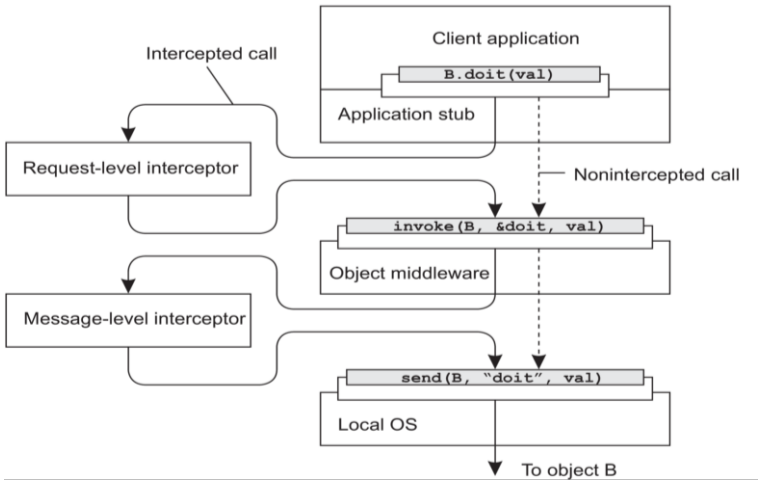
Interceptors

Conceptually, an interceptor is a software construct that will break the usual flow of control and allow other (application specific) code to be executed. Interceptors are a primary means for adapting middleware to the specific needs of an application. As such, they play an important role in making middleware open. To make interceptors generic may require a substantial implementation effort and it is unclear whether in such cases generality should be preferred over restricted applicability and simplicity. Also, in many cases having only limited interception facilities will improve management of the software and the distributed system as a whole.

To make matters concrete, consider interception as supported in many object-based distributed systems. The basic idea is simple: an object A can call a method that belongs to an object B, while the latter resides on a different machine than A. Such a remote-object invocation is carried out in three steps:

1. Object A is offered a local interface that is exactly the same as the interface offered by object B. A calls the method available in that interface.

2. The call by A is transformed into a generic object invocation, made possible through a general object-invocation interface offered by the middleware at the machine where A resides.
3. Finally, the generic object invocation is transformed into a message that is sent through the transport-level network interface as offered by A's local operating system.



Modifiable middleware

What wrappers and interceptors offer are means to extend and adapt the middleware. The need for adaptation comes from the fact that the environment in which distributed applications are executed changes continuously. Changes include those resulting from mobility, a strong variance in the quality-of service of networks, failing hardware, and battery drainage, amongst others. Rather than making applications responsible for reacting to changes, this task is placed in the middleware. Moreover, as the size of a distributed system increases, changing its parts can rarely be done by temporarily shutting it down. What is needed is being able to make changes on-the-fly.

These strong influences from the environment have brought many designers of middleware to consider the construction of adaptive software. Middleware may not only need to be adaptive, but that we should be able to purposefully modify it without bringing it down. In this context, interceptors can be thought of offering a means to adapt the standard flow of control. Replacing software components at runtime is an example of modifying a system.

Component-based design focuses on supporting modifiability through composition. A system may either be configured statically at design time, or dynamically at runtime. The latter requires support for late binding, a technique that has been successfully applied in programming language environments, but also for operating systems where modules can be loaded and unloaded at will. Research is now well underway to automatically select the best implementation of a component during runtime but again, the process remains complex for distributed systems, especially when considering that replacement of one component requires to know exactly what the effect of that replacement on other components will be. In many cases, components are less independent as one may think.

The bottom line is that in order to accommodate dynamic changes to the software that makes up middleware, we need at least basic support to load and unload components at runtime. In addition, for each component explicit specifications of the interfaces it offers, as well the interfaces it requires, are needed. If state is maintained between calls to a component, then further special measures are needed. By-and-large, it should be clear that organizing middleware to be modifiable requires very special attention.

2.3 System Architecture

System Architecture, also known as software architecture includes the software components, their interactions and their placements.

Types of System Architecture

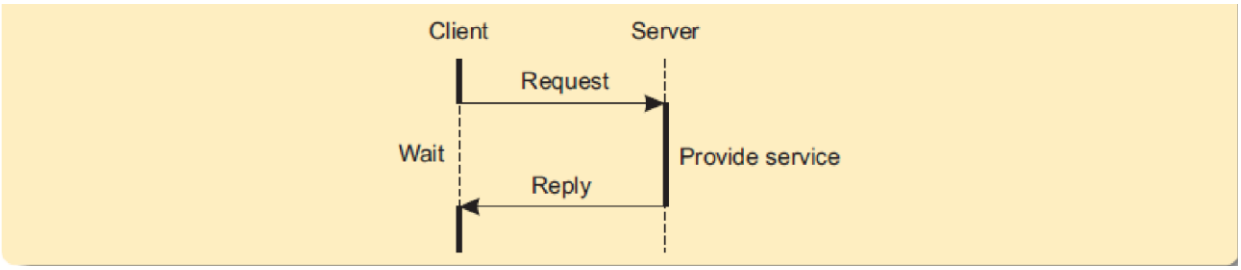
1. Centralized Organizations
    - Simple client-server architecture
    - Multitiered architecture
2. Decentralized Organizations: Peer-to-Peer Systems
    - Structured peer-to-peer systems
    - Unstructured peer-to-peer systems
    - Hierarchically organized peer-to-peer networks
3. Hybrid Architectures
    - Edge-server systems
    - Collaborative distributive systems

Centralized Organizations

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 understanding and managing the complexity of distributed systems. It is implemented by using: simple client-server architecture and multitiered architecture.

Simple client-server architecture

In the basic client-server model, processes in a distributed system are divided into two 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.



Communication between a client and a server can be implemented by means of a simple connectionless protocol when the underlying network is fairly reliable as in many local-area networks. In these cases, when a client requests a service, it simply packages a message for the server, identifying the service it wants, along with the necessary input data. The message is then sent to the server. The latter, in turn, will always wait for an incoming request, subsequently process it, and package the results in a reply message that is then sent to the client.

Using a connectionless protocol has the obvious advantage of being efficient. As long as messages do not get lost or corrupted, the request/reply protocol just sketched works fine.

As an alternative, many client-server systems use a reliable connection oriented protocol. Although this solution is not entirely appropriate in a local-area network due to relatively low performance, it works perfectly fine in wide-area systems in which communication is inherently unreliable.

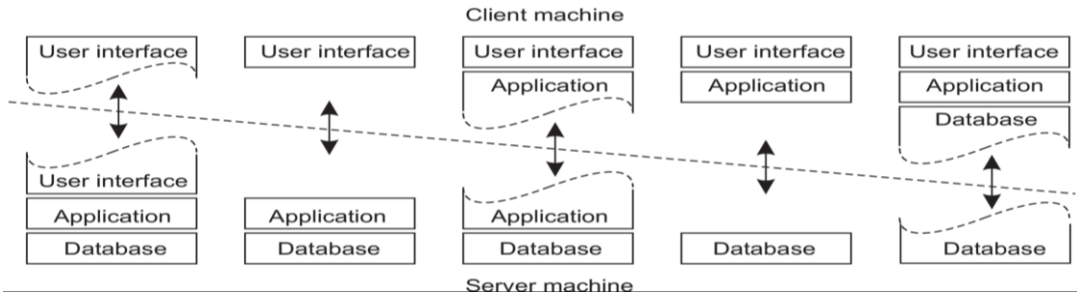
Multitiered Architectures

The distinction into three logical levels (user interface, processing and data level), suggests a number of possibilities for physically distributing a client-server application across several machines. The simplest organization is to have only two types of machines:

1. A client machine containing only the programs implementing (part of) the user-interface level
2. A server machine containing the rest, that is, the programs implementing the processing and data level

In this organization everything is handled by the server while the client is essentially no more than a dumb terminal, possibly with only a convenient graphical interface. There are, however, many other possibilities.

One approach for organizing clients and servers is then to distribute these layers across different machines.



As a first step, we make a distinction between only two kinds of machines: client machines and server machines, leading to what is also referred to as a (physically) two-tiered architecture.

One possible organization is to have only the terminal-dependent part of the user interface on the client machine and give the applications remote control over the presentation of their data.

An alternative is to place the entire user-interface software on the client side. In such cases, we essentially divide the application into a graphical front end, which communicates with the rest of the application (residing at the server) through an application-specific protocol. In this model, the front end (the client software) does no processing other than necessary for presenting the application's interface.

Continuing along this line of reasoning, we may also move part of the application to the front end. An example where this makes sense is where the application makes use of a form that needs to be filled in entirely before it can be processed. The front end can then check the correctness and consistency of the form, and where necessary interact with the user.

In many client-server environments, the organizations shown in last two figures are particularly popular. These organizations are used where the client machine is a PC or workstation, connected through a network to a distributed file system or database. Essentially, most of the application is running on the client machine, but all operations on files or database entries go to the server. For example, many banking applications run on an end-user's machine where the user prepares transactions and such. Once finished, the application contacts the database on the bank's server and uploads the transactions for further processing. The last figure represents the situation where the client's local disk contains part of the data.

For example, when browsing the Web, a client can gradually build a huge cache on local disk of most recent inspected Web pages.

Decentralized organizations: peer-to-peer systems

Multitiered client-server architectures are a direct consequence of dividing distributed applications into a user interface, processing components, and data-management components. 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.

From a systems-management perspective, having a vertical distribution can help: functions are logically and physically split across multiple machines, where each machine is tailored to a specific group of functions. However, vertical distribution is only one way of organizing client-server applications. In modern architectures, it is often the distribution of the clients and the servers that counts, which we refer to as horizontal distribution. In this type of distribution, a client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set, thus balancing the load. This class of modern system architectures that support horizontal distribution, known as peer-to-peer systems.

From a high-level perspective, the processes that constitute a peer-to-peer system are all equal. This means that the functions that need to be carried out are represented by every process that constitutes the distributed system. As a consequence, much of the interaction between processes is symmetric: each process will act as a client and a server at the same time.

Peer-to-peer systems can be:

- Structured peer-to-peer systems
- Unstructured peer-to-peer systems
- Hierarchically organized peer-to-peer networks

Structured peer-to-peer systems

As its name suggests, in a structured peer-to-peer system the nodes (i.e., processes) are organized in an overlay that follows to a specific, deterministic topology: a ring, a binary tree, a grid, etc. This topology is used to efficiently look up data.

Characteristic for structured peer-to-peer systems, is that they are generally based on using a so-called semantic-free index. What this means is that each data item that is to be maintained by the system, is uniquely associated with a key, and that this key is subsequently used as an index. To this end, it is common to use a hash function, so that we get:

key(data item) = hash(data item's value).

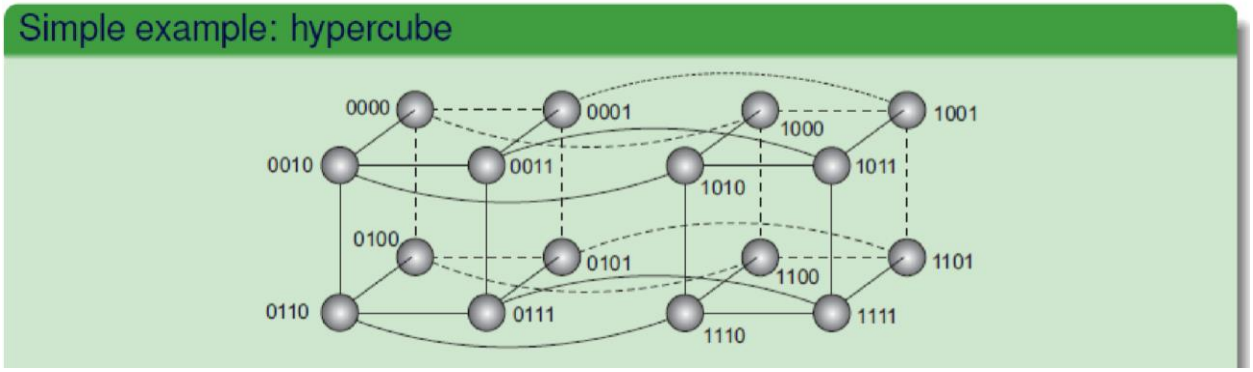
The peer-to-peer system as a whole is now responsible for storing (key, value) pairs. To this end, each node is assigned an identifier from the same set of all possible hash values, and each node is made responsible for storing data associated with a specific subset of keys. In essence, the system is thus seen to implement a distributed hash table, generally abbreviated to a DHT.

Following this approach now reduces the essence of structured peer-to-peer systems to being able to look up a data item by means of its key. That is, the system provides an efficient implementation of a function lookup that maps a key to an existing node:

existing node = lookup(key).

This is where the topology of a structured peer-to-peer system plays a crucial role. Any node can be asked to look up a given key, which then boils down to efficiently routing that lookup request to the node responsible for storing the data associated with the given key.

To clarify these matters, let us consider a simple peer-to-peer system with a fixed number of nodes, organized into a hypercube. A hypercube is an n-dimensional cube. The hypercube shown in Figure is four-dimensional. It can be thought of as two ordinary cubes, each with 8 vertices and 12 edges. To expand the hypercube to five dimensions, we would add another set of two interconnected cubes to the figure, connect the corresponding edges in the two halves, and so on.



Unstructured peer-to-peer systems

Structured peer-to-peer systems attempt to maintain a specific, deterministic overlay network. In contrast, in an unstructured peer-to-peer system each node maintains an ad hoc list of neighbors. The resulting overlay resembles what is known as a random graph: a graph in which an edge hu, vi between two nodes u and v exists only with a certain probability P[hu, vi]. Ideally, this probability is the same for all pairs of nodes, but in practice a wide range of distributions is observed.

In an unstructured peer-to-peer system, when a node joins it often contacts a well-known node to obtain a starting list of other peers in the system. This list can then be used to find more peers, and perhaps ignore others, and so on. In practice, a node generally changes its local list almost continuously. For example, a node may discover that a neighbor is no longer responsive and that it needs to be replaced. There may be other reasons, which we will describe shortly.

Unlike structured peer-to-peer systems, looking up data cannot follow a predetermined route when lists of neighbors are constructed in an ad hoc fashion. Instead, in an unstructured peer-to-peer systems we really need to resort to searching for data. Let us look at two extremes and consider the case in which we are requested to search for specific data (e.g., identified by keywords).

Flooding: In the case of flooding, an issuing node u simply passes a request for a data item to all its neighbors. A request will be ignored when its receiving node, say v, had seen it before. Otherwise, v searches locally for the requested data item. If v has the required data, it can either respond directly to the issuing node u, or send it back to the original forwarder, who will then return it to its original forwarder, and so on. If v does not have the requested data, it forwards the request to all of its own neighbors.

Obviously, flooding can be very expensive, for which reason a request often has an associated time-to-live or TTL value, giving the maximum number of hops a request is allowed to be forwarded. Choosing the right TTL value is crucial: too small means that a request will stay close to the issuer and may thus not reach a node having the data. Too large incurs high communication costs.

As an alternative to setting TTL values, a node can also start a search with an initial TTL value of 1, meaning that it will first query only its neighbors. If no, or not enough results are returned, the TTL is increased and a new search is initiated.

Random walks: At the other end of the search spectrum, an issuing node u can simply try to find a data item by asking a randomly chosen neighbor, say v. If v does not have the data, it forwards the request to one of its randomly chosen neighbors, and so on. The result is known as a random walk. Obviously, a random walk imposes much less network traffic, yet it may take much longer before a node is reached that has the requested data. To decrease the waiting time, an issuer can simply start n random walks simultaneously. Indeed, studies show that in this case, the time it takes before reaching a node that has the data drops approximately by a factor n. Lv et al. report that relatively small values of n, such as 16 or 64, turn out to be effective.

A random walk also needs to be stopped. To this end, we can either again use a TTL, or alternatively, when a node receives a lookup request, check with the issuer whether forwarding the request to another randomly selected neighbor is still needed.

Note that neither method relies on a specific comparison technique to decide when requested data has been found. For structured peer-to-peer systems, we assumed the use of keys for comparison; for the two approaches just described, any comparison technique would suffice.

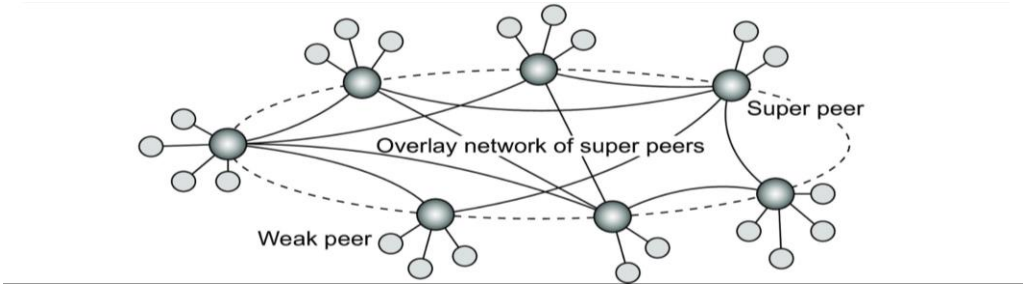
Between flooding and random walks lie policy-based search methods. For example, a node may decide to keep track of peers who responded positively, effectively turning them into preferred neighbors for succeeding queries. Likewise, we may want to restrict flooding to fewer neighbors, but in any case give preference to neighbors having many neighbors themselves.

Hierarchically organized peer-to-peer networks

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 searching for the request by means of flooding or randomly walking through the network. 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 to each other. For example, in a collaborative content delivery network (CDN), nodes may offer storage for hosting copies of Web documents allowing Web clients to access pages nearby, and thus to access them quickly. What is needed is a means to find out where documents can be stored best. In that case, making use of a broker that collects data on resource usage and availability for a number of nodes that are in each other's proximity will allow to quickly select a node with sufficient resources.

Nodes such as those maintaining an index or acting as a broker are generally referred to as super peers. As the name suggests, super peers are often also organized in a peer-to-peer network, leading to a hierarchical organization as explained in Yang and Garcia-Molina [2003]. A simple example of such an organization is shown in Figure. In this organization, every regular peer, now referred to as a weak peer, is connected as a client to a super peer. All communication from and to a weak peer proceeds through that peer's associated super peer.



In many cases, the association between a weak peer and its super peer is fixed: whenever a weak peer joins the network, it attaches to one of the super peers and remains attached until it leaves the network. Obviously, it is expected that super peers are long-lived processes with high availability. To compensate for potential unstable behavior of a super peer, backup schemes can be deployed, such as pairing every super peer with another one and requiring weak peers to attach to both.

Having a fixed association with a super peer may not always be the best solution. For example, in the case of file-sharing networks, it may be better for a weak peer to attach to a super peer that maintains an index of files that the weak peer is currently interested in. In that case, chances are bigger that when a weak peer is looking for a specific file, its super peer will know where to find it. Garbacki et al. [2010] describe a relatively simple scheme in which the association between weak peer and strong peer can change as weak peers discover better super peers to associate with. In particular, a super peer returning the result of a lookup operation is given preference over other super peers.

As we have seen, peer-to-peer networks offer a flexible means for nodes to join and leave the network. However, with super-peer networks a new problem is introduced, namely how to select the nodes that are eligible to become super peer.

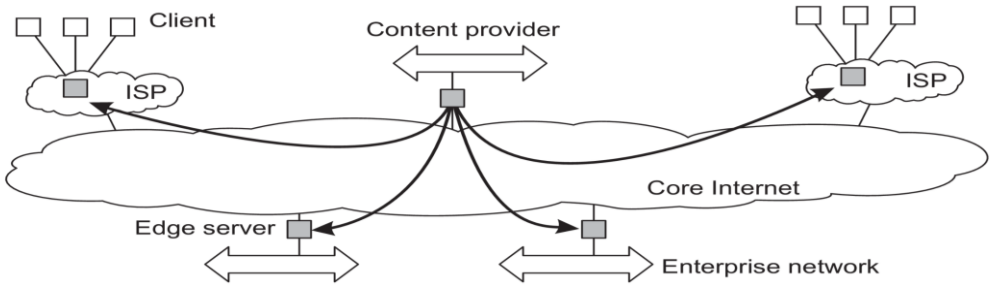
Hybrid Architectures

Many distributed systems combine architectural features, as we already came across in super-peer networks. There exists some specific classes of distributed systems in which client-server solutions are combined with decentralized architectures. Some of them are:

- Edge-server systems
- Collaborative distributive systems

Edge-server systems

An important class of distributed systems that is organized according to a hybrid architecture is formed by edge-server systems. These systems are deployed on the Internet where servers are placed “at the edge” of the network. This edge is formed by the boundary between enterprise networks and the actual Internet, for example, as provided by an Internet Service Provider (ISP). Likewise, where end users at home connect to the Internet through their ISP, the ISP can be considered as residing at the edge of the Internet. This leads to a general organization like the one shown in Figure.



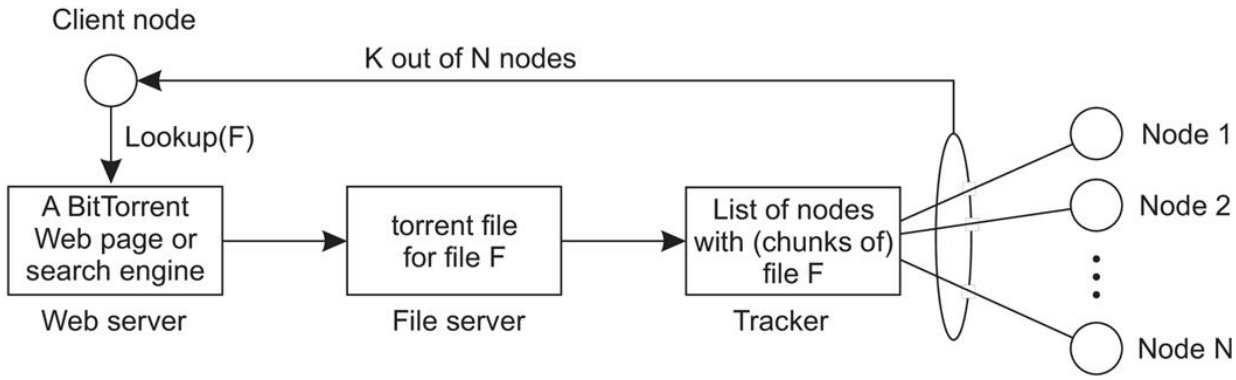
End users, or clients in general, connect to the Internet by means of an edge server. The edge server's main purpose is to serve content, possibly after applying filtering and transcoding functions. More interesting is the fact that a collection of edge servers can be used to optimize content and application distribution. The basic model is that for a specific organization, one edge server acts as an origin server from which all content originates. That server can use other edge servers for replicating Web pages and such.

This concept of edge-server systems is now often taken a step further: taking cloud computing as implemented in a data center as the core, additional servers at the edge of the network are used to assist in computations and storage, essentially leading to distributed cloud systems. In the case of fog computing, even end-user devices form part of the system and are (partly) controlled by a cloud-service provider.

Collaborative distributed systems

Hybrid structures are notably deployed in collaborative distributed systems. The main issue in many of these systems is to first get started, for which often a traditional client-server scheme is deployed. Once a node has joined the system, it can use a fully decentralized scheme for collaboration.

To make matters concrete, let us consider the widely popular BitTorrent file-sharing system [Cohen, 2003]. BitTorrent is a peer-to-peer file downloading system. Its principal working is shown in Figure. The basic idea is that when an end user is looking for a file, he downloads chunks of the file from other users until the downloaded chunks can be assembled together yielding the complete file. An important design goal was to ensure collaboration. In most file-sharing systems, a significant fraction of participants merely download files but otherwise contribute close to nothing, a phenomenon referred to as free riding. To prevent this situation, in BitTorrent a file can be downloaded only when the downloading client is providing content to someone else.



To download a file, a user needs to access a global directory, which is generally just one of a few well-known Web sites. Such a directory contains references to what are called torrent files. A torrent file contains the information that is needed to download a specific file. In particular, it contains a link to what is known as a tracker, which is a server that is keeping an accurate account of active nodes that have

(chunks of) the requested file. An active node is one that is currently downloading the file as well. Obviously, there will be many different trackers, although there will generally be only a single tracker per file (or collection of files).

Once the nodes have been identified from where chunks can be downloaded, the downloading node effectively becomes active. At that point, it will be forced to help others, for example by providing chunks of the file it is downloading that others do not yet have. This enforcement comes from a very simple rule: if node P notices that node Q is downloading more than it is uploading, P can decide to decrease the rate at which it sends data to Q. This scheme works well provided P has something to download from Q. For this reason, nodes are often supplied with references to many other nodes putting them in a better position to trade data.

Clearly, BitTorrent combines centralized with decentralized solutions. As it turns out, the bottleneck of the system is, not surprisingly, formed by the trackers. In an alternative implementation of BitTorrent, a node also joins a separate structured peer-to-peer system (i.e., a DHT) to assist in tracking file downloads. In effect, a central tracker's load is now distributed across the participating nodes, with each node acting as a tracker for a relatively small set of torrent files. The original function of the tracker coordinating the collaborative downloading of a file is retained. However, we note that in many BitTorrent systems used today, the tracking functionality has actually been minimized to a one-time provisioning of peers currently involved in downloading the file. From that moment on, the newly participating peer will communicate only with those peers and no longer with the initial tracker. The initial tracker for the requested file is looked up in the DHT through a so-called magnet link.

2.4 Example architectures

1. The Network File System
2. The Web

The Network File System

Many distributed files systems are organized along the lines of client-server architectures, with Sun Microsystem's Network File System (NFS) being one of the most widely-deployed ones for Unix systems. NFSv3 is the widely-used third version of NFS and NFSv4, the most recent, fourth version.

The basic idea behind NFS is that each file server provides a standardized view of its local file system. In other words, it should not matter how that local file system is implemented; each NFS server supports the same model. This approach has been adopted for other distributed files systems as well. NFS comes with a communication protocol that allows clients to access the files stored on a server, thus allowing a heterogeneous collection of processes, possibly running on different operating systems and machines, to share a common file system.

The model underlying NFS and similar systems is that of a remote file service. In this model, clients are offered transparent access to a file system that is managed by a remote server. However, clients are normally unaware of the actual location of files. Instead, they are offered an interface to a file system that is similar to the interface offered by a conventional local file system. In particular, the client is offered only an interface containing various file operations, but the server is responsible for implementing those operations. This model is therefore also referred to as the remote access model.

In contrast, in the upload/download model a client accesses a file locally after having downloaded it from the server, when the client is finished with the file, it is uploaded back to the server again so that it can be

used by another client. The Internet's FTP service can be used this way when a client downloads a complete file, modifies it, and then puts it back.

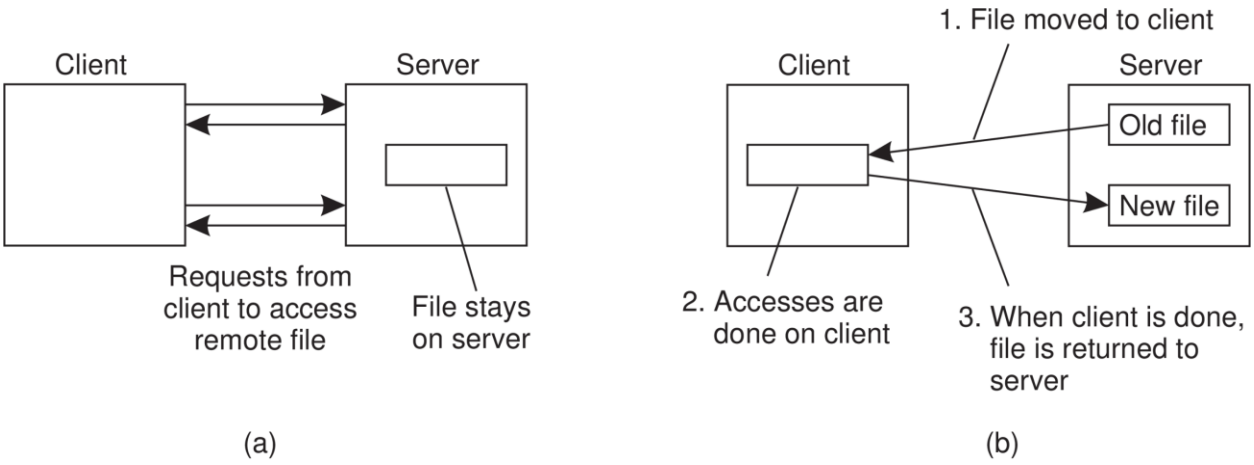


Figure: (a) The Remote Access Model (b) The upload/download Model

NFS has been implemented for a large number of different operating systems, although the Unix versions are predominant.

The Web

The architecture of Web-based distributed systems is not fundamentally different from other distributed systems. However, it is interesting to see how the initial idea of supporting distributed documents has evolved since its inception in 1990s. Documents turned from being purely static and passive to dynamically generated content. Furthermore, in recent years, many organizations have begun supporting services instead of just documents.

Simple Web-based systems

Many Web-based systems are still organized as relatively simple client-server architectures. The core of a Web site is formed by a process that has access to a local file system storing documents. The simplest way to refer to a document is by means of a reference called a Uniform Resource Locator (URL). It specifies where a document is located by embedding the DNS name of its associated server along with a file name by which the server can look up the document in its local file system. Furthermore, a URL specifies the application-level protocol for transferring the document across the network.

A client interacts with Web servers through a browser, which is responsible for properly displaying a document. Also, a browser accepts input from a user mostly by letting the user select a reference to another document, which it then subsequently fetches and displays. The communication between a browser and Web server is standardized: they both adhere to the HyperText Transfer Protocol (HTTP). This leads to the overall organization shown in Figure below.

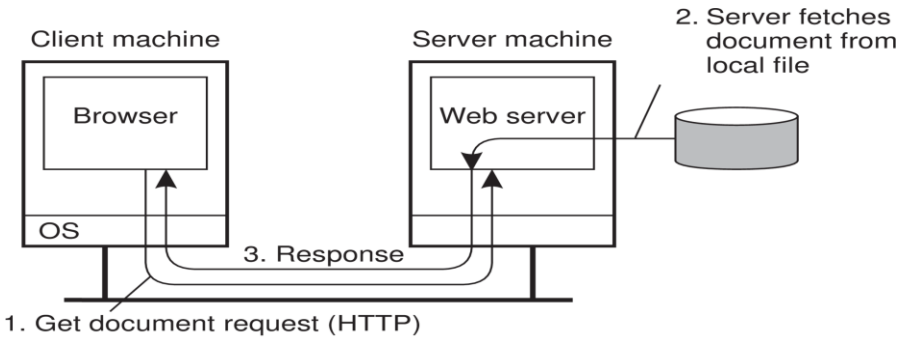


Figure: The overall organization of a traditional web site

It uses the document in the simplest form is a standard text file. In that case, the server and browser have barely anything to do: the server copies the file from the local file system and transfers it to the browser. The latter, in turn, merely displays the content of the file ad verbatim without further ado.

More interesting are Web documents that have been marked up, which is usually done in the HyperText Markup Language, or simply HTML. In that case, the document includes various instructions expressing how its content should be displayed, similar to what one can expect from any decent word-processing system (although those instructions are normally hidden from the end user).

Multitiered architectures

The Web started out as the relatively simple two-tiered client-server system shown in Figure above. By now, this simple architecture has been extended to support much more sophisticated means of documents. Most things that we get to see in our browser has been generated on the spot as the result of sending a request to a Web server. Content is stored in a database at the server's side, along with client-side scripts and such, to be composed on-the-fly into a document which is then subsequently sent to the client's browser. Documents have thus become completely dynamic.

One of the first enhancements to the basic architecture was support for simple user interaction by means of the Common Gateway Interface or simply CGI. CGI defines a standard way by which a Web server can execute a program taking user data as input. Usually, user data come from an HTML form; it specifies the program that is to be executed at the server side, along with parameter values that are filled in by the user. Once the form has been completed, the program's name and collected parameter values are sent to the server, as shown in Figure below.

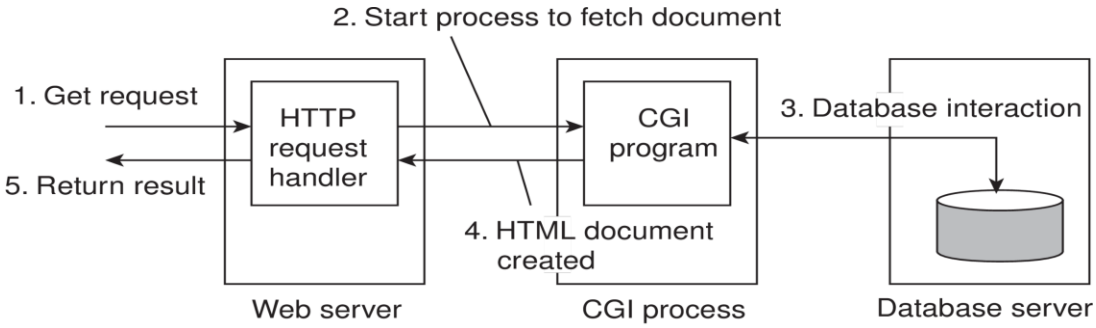


Figure: The principle of using server-side CGI programs

When the server sees the request, it starts the program named in the request and passes it the parameter values. At that point, the program simply does its work and generally returns the results in the form of a document that is sent back to the user's browser to be displayed.

CGI programs can be as sophisticated as a developer wants. For example, as shown in Figure above many programs operate on a database local to the Web server. After processing the data, the program generates an HTML document and returns that document to the server. The server will then pass the document to the client. An interesting observation is that to the server, it appears as if it is asking the CGI program to fetch a document. In other words, the server does nothing but delegate the fetching of a document to an external program.

The main task of a server used to be handling client requests by simply fetching documents. With CGI programs, fetching a document could be delegated in such a way that the server would remain unaware of whether a document had been generated on the fly, or actually read from the local file system. Note that we have just described a two-tiered organization of server-side software.

However, servers nowadays do much more than just fetching documents. One of the most important enhancements is that servers can also process a document before passing it to the client. In particular, a document may contain a server-side script, which is executed by the server when the document has been fetched locally. The result of executing a script is sent along with the rest of the document to the client. The script itself is not sent. In other words, using a server-side script changes a document by essentially replacing the script with the results of its execution.

Practice Questions

1. What are the different architecture styles of distributed system? Explain each of them in detail.
2. Define middleware and explain its importance in distributed system with suitable diagram.
3. Describe types of system architecture with suitable example.
4. Define modifiable middleware. Write down its advantages with suitable example.
5. Explain centralized system with example, advantages and disadvantages.
6. Explain de-centralized system with example, advantages and disadvantages.
7. Explain peer-to-peer system with example, advantages and disadvantages.
8. Define layered architecture. Compare it with event based architecture.
9. Define object based architecture. Compare it with data centered architecture
10. Define hybrid architecture with suitable example.
11. Write short notes and NFS and the Web.