Distributed System Architectures

MSc. Huynh Nam

Very important lesson for IS and SE major!

Introduction

- Distributed systems are often complex **pieces of software** of which the components are by definition dispersed across multiple machines.
- 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. However, trade-offs need to be made to achieve transparency, which has led to various techniques to adjust the middleware to the needs of the applications that make use of it.
- 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**.

Content

- Architectural styles
- Middleware organization
- System architecture
- Example architectures
- Summary

*Architectural styles

- The notion of an **architectural style** is important. Such a 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 (**as interface in **OOP concept)** that is ***replaceable*** within its environment. That a component can be replaced, and, in particular, while a system continues to operate, is important. This is due to the fact that in many cases, it is not an option to shut down a system for maintenance. At best, only parts of it may be put temporarily out of order. Replacing a component can be done only if its interfaces remain untouched.
- A somewhat more difficult concept to grasp is that of a **connector**, which is generally described as a mechanism that mediates communication, coordination, or cooperation among components.
- *For example, a connector can be formed by the facilities for (remote) procedure calls, message passing, or streaming data. In other words, a connector allows for the flow of control and data between 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**
  - **Event-based architectures**
- Notably following an approach by which a system is **subdivided** into several (logical) layers is such a universal principle that it is generally combined with most other architectural styles.

*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 **down-call** 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.

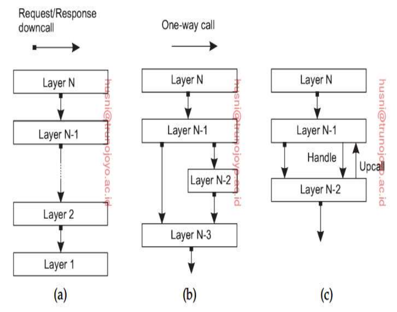


Figure 2.1: (a) Pure layered organization. (b) Mixed layered organization. (c) Layered organization with upcalls (adopted from [Krakowiak, 2009]).

*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. In principle, the interface should completely hide the actual implementation of a service. Another important concept in the case of communication is that of a **(communication) protocol**, which describes the rules that parties will follow in order to exchange information. It is important to understand the difference between a service offered by a layer, the interface by which that service is made available, and the protocol that a layer implements to establish communication. This distinction is shown in *Figure 2.2*.
- To make this distinction clear, consider a reliable, **connection-oriented service**, which is provided by many communication systems. In this case, a communicating party first needs to set up a connection to another party before the two can send and receive messages. Being **reliable** means that strong guarantees will be given that sent messages will indeed be delivered to the other side, even when there is a high risk that messages may be lost (as, for example, may be the case when using a wireless medium). In addition, such services generally also ensure that messages are delivered in the same order as that they were sent.
- This kind of service is realized on the Internet by means of the **Transmission Control Protocol (TCP)**. The protocol specifies which messages are to be exchanged for setting up or tearing down a connection, what needs to be done to preserve the ordering of transferred data, and what both parties need to do to detect and correct data that was lost during transmission. The service is made available in the form of a relatively simple programming interface, containing calls to set up a connection, send and receive messages, and to tear down the connection again. In fact, there are different interfaces available, often dependent on operating system or programming language used. Likewise, there are many different implementations of the protocol and its interfaces.

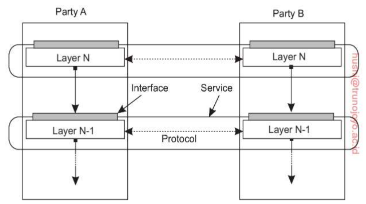
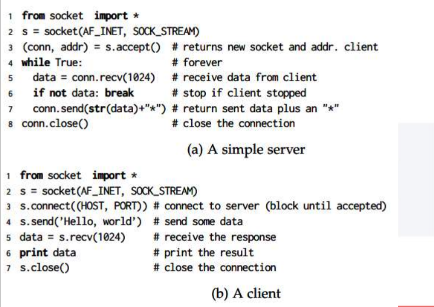


Figure 2.2: A layered communication-protocol stack, showing the difference between a service, its interface, and the protocol it deploys.

Example: Two communicating parties

- To make this distinction between service, interface, and protocol more concrete, consider the following two communicating parties, also known as a **client** and a **server**, respectively, expressed in Python.



```
from socket  import *
s = socket(AF_INET, SOCK_STREAM)
(conn, addr) = s.accept()  # returns new socket and addr. client
while True:                # forever
  data = conn.recv(1024)   # receive data from client
  if not data: break       # stop if client stopped
  conn.send(str(data)+"*") # return sent data plus an "*"
conn.close()               # close the connection
```

(a) A simple server

```
from socket  import *
s = socket(AF_INET, SOCK_STREAM)
s.connect((HOST, PORT)) # connect to server (block until accepted)
s.send('Hello, world')  # send some data
data = s.recv(1024)     # receive the response
print data              # print the result
s.close()               # close the connection
```

(b) A client

Figure 2.3: Two communicating parties.

- In this example, a server is created that makes use of a **connection-oriented service** as offered by the socket library available in Python. This service allows two communicating parties to reliably send and receive data over a connection. The main functions available in its interface are:
  - *socket*(): to create an object representing the connection
  - *accept*(): a blocking call to wait for incoming connection requests; if successful, the call returns a new socket for a separate connection
  - *connect*(): to set up a connection to a specified party
  - *close*(): to tear down a connection
  - *send*(), *recv*(): to send and receive data over a connection, respectively
- The combination of constants *AF_INET* and *SOCK_STREAM* is used to specify that the TCP protocol should be used in the communication between the two parties. These two constants can be seen as part of the interface, whereas making use of TCP is part of the offered service. How TCP is implemented, or for that matter any part of the communication service is hidden completely from the applications.
- Finally, also note that these two programs implicitly adhere to an application-level protocol: apparently, if the client sends some data, the server will return it. Indeed, it operates as an echo server where the server adds an asterisk to the data sent by the client.

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:
  - **The application-interface level**: interaction with a user or some external application
  - **The processing level**: the processing level consists of a relatively large collection of programs, each having rather simple processing capabilities.
  - **The data level**: that maintain the actual data on which the applications operate. An important property of this level is that data are often ***persistent***, that is, even if no application is running, data will be stored somewhere for next use. In its simplest form, the data level consists of a file system, but it is also common to use a full-fledged database. Besides merely storing data, the data level is generally also responsible for keeping data consistent across different applications. When databases are being used, ***maintaining consistency*** means that metadata such as table descriptions, entry constraints and application-specific metadata are also stored at this level.
- The applications layer 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.

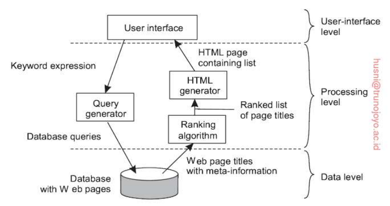


Figure 2.4: The simplified organization of an Internet search engine into three different layers.

*Object-based and service-oriented architectures

- In essence, each object corresponds to what we have defined as a component, and these components are connected through a ***procedure call*** mechanism (if two objects reside on the same system → **method call**). In the case of distributed systems, a procedure call can also take place over a network (over a network → **remote procedure call - RPC**), 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.

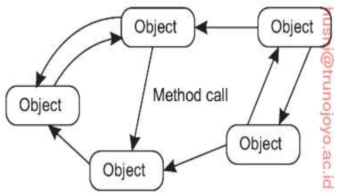


Figure 2.5: An object-based architectural style.

*Object-based and service-oriented architectures

- *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.* This organization, which is shown in Figure 2.6 is commonly referred to as a distributed object.

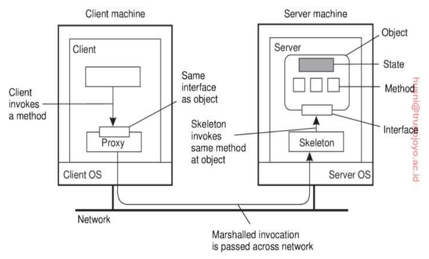


Figure 2.6: Common organization of a remote object with client-side proxy.

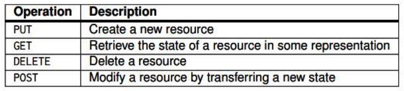
- When a client binds to a distributed object, an implementation of the object's interface, called a **proxy**, is then loaded into the client's address space. A proxy is analogous to a **client stub** in RPC systems. The only thing it does is ***marshal*** method invocations into messages and ***un-marshal*** reply messages to return the result of the method invocation to the client. The actual object resides at a server machine, where it offers the same interface as it does on the client machine. Incoming invocation requests are first passed to a **server stub**, which un-marshals them to make method invocations at the object's interface at the server.
- The server stub is also responsible for marshaling replies and forwarding reply messages to the client-side proxy. The server-side stub is often referred to as a ***skeleton*** as it provides the bare means for letting the server middleware access the user-defined objects. In practice, it often contains incomplete code in the form of a language-specific class that needs to be further specialized by the developer

Argumentation

- A characteristic, but somewhat counterintuitive feature of most distributed objects is that their state is not distributed: it resides at a single machine. Only the interfaces implemented by the object are made available on other machines. Such objects are also referred to as remote objects. In a general distributed object, the state itself may be physically distributed across multiple machines, but this distribution is also hidden from clients behind the object's interfaces.
- One could argue that object-based architectures form the foundation of encapsulating services into independent units. Encapsulation is the keyword here: the service as a whole is realized as a self-contained entity, although it can possibly make use of other services. By clearly separating various services such that they can operate independently, we are paving the road toward **service-oriented architectures**, generally abbreviated as **SOAs**.
- In a service-oriented architecture, a distributed application or system is essentially constructed as a composition of many different services. Not all of these services may belong to the same administrative organization. We already came across this phenomenon when discussing cloud computing: it may very well be that an organization running its business application makes use of storage services offered by a cloud provider. These storage services are logically completely encapsulated into a single unit, of which an interface is made available to customers. Of course, storage is a rather basic service, but more sophisticated situations easily come to mind.
- In this way, we see that the problem of developing a distributed system is partly one of service composition and making sure that those services operate in harmony. Indeed, this problem is completely analogous to the enterprise application integration issues. Crucial is, and remains, that each service offers a well-defined (programming) interface. In practice, this also means that each service offers its own interface, in turn, possibly making the composition of services far from trivial.

*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 architectures:
  - Resources are identified through a single naming scheme
  - All services offer the same interface, consisting of at most four operations, as shown in *Figure 2.7*
  - Messages sent to or from a service are fully self-described
  - After executing an operation at a service, that component forgets everything about the caller (is also referred to as a stateless execution)



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

Figure 2.7: The four operations available in RESTful architectures.

- To illustrate how RESTful can work in practice, consider a cloud storage service, such as Amazon's Simple Storage Service (Amazon S3). Amazon S3, described in supports only two resources: objects, which are essentially the equivalent of files, and buckets, the equivalent of directories. There is no concept of placing buckets into buckets. An object named Object-Name contained in bucket Bucket-Name is referred to by means of the following Uniform Resource Identifier (URI): http://BucketName.s3.amazonaws.com/ObjectName
- To create a bucket, or an object for that matter, an application would essentially send a PUT request with the URI of the bucket/object. In principle, the protocol that is used with the service is HTTP. In other words, it is just another HTTP request, which will subsequently be correctly interpreted by S3. If the bucket or object already exists, an HTTP error message is returned.
- In a similar fashion, to know which objects are contained in a bucket, an application would send a GET request with the URI of that bucket. S3 will return a list of object names, again as an ordinary HTTP response.

*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.
- The 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, as shown in Figure 2.9.

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

Figure 2.9: Examples of different forms of coordination.

Referentially coupled

	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.

*Referentially decoupled

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

- 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.
- A well-known coordination model is the combination of referentially and temporally decoupled processes, leading to what is known as a shared data space. The key idea is that processes communicate entirely through tuples, which are structured data records consisting of a number of fields, very similar to a row in a database table. Processes can put any type of tuple into the shared data space. In order to retrieve a tuple, a process provides a search pattern that is matched against the tuples. Any tuple that matches is returned.
- Shared data spaces are thus seen to implement an associative search mechanism for tuples. When a process wants to extract a tuple from the data space, it specifies (some of) the values of the fields it is interested in. Any tuple that matches that specification is then removed from the data space and passed to the process.
- Shared data spaces are often combined with event-based coordination: a process subscribes to certain tuples by providing a search pattern; when a process inserts a tuple into the data space, matching subscribers are notified. In both cases, we are dealing with a publish-subscribe architecture, and indeed, the key characteristic feature is that processes have no explicit reference to each other. The difference between a pure event-based architectural style, and that of a shared data space is shown in Figure 2.10. We have also shown an abstraction of the mechanism by which publishers and subscribers are matched, known as an event bus.

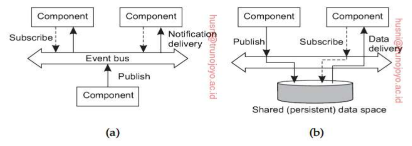


Figure 2.10: The (a) event-based and (b) shared data-space architectural style.

Communication in publish-subscribe systems

- An important aspect of publish-subscribe systems is that communication takes place by describing the events that a subscriber is interested in. As a consequence, naming plays a crucial role. We return to naming later, but for now the important issue is that in many cases, data items are not explicitly identified by senders and receivers.
- Let us first assume that events are described by a **series of attributes**. A notification describing an event is said to be published when it is made available for other processes to read. To that end, a subscription needs to be passed to the middleware, containing a description of the event that the subscriber is interested in. Such a description typically consists of some **(attribute, value)** pairs, which is common for so-called topic-based publish-subscribe systems.
- As an alternative, in content-based publish-subscribe systems, a subscription may also consist of **(attribute, range)** pairs. In this case, the specified attribute is expected to take on values within a specified range. Descriptions can sometimes be given using all kinds of predicates formulated over the attributes, very similar in nature to SQL-like queries in the case of relational databases. Obviously, the more complex a description is, the more difficult it will be to test whether an event matches a description.
- We are now confronted with a situation in which subscriptions need to be matched against notifications, as shown in Figure 2.12. In many cases, an event actually corresponds to data becoming available. In that case, when matching succeeds, there are two possible scenarios. In the first case, the middleware may decide to forward the published notification, along with the associated data, to its current set of subscribers, that is, processes with a matching subscription. As an alternative, the middleware can also forward only a notification at which point subscribers can execute a read operation to retrieve the associated data item.

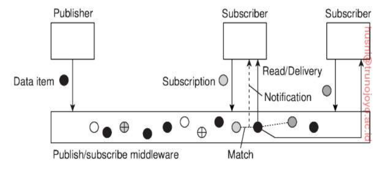


Figure 2.12: The principle of exchanging data items between publishers and subscribers.

Communication in publish-subscribe systems

- In those cases, in which data associated with an event are immediately forwarded to subscribers, the middleware will generally not offer storage of data. Storage is either explicitly handled by a separate service or is the responsibility of subscribers. In other words, we have a referentially decoupled, but temporally coupled system.
- This situation is different when notifications are sent so that subscribers need to explicitly read the associated data. Necessarily, the middleware will have to store data items. In these situations, there are additional operations for data management. It is also possible to attach a lease to a data item such that when the lease expires that the data item is automatically deleted.
- Events can easily complicate the processing of subscriptions. To illustrate, consider a subscription such as "notify when room ZI.1060 is unoccupied and the door is unlocked." Typically, a distributed system supporting such subscriptions can be implemented by placing independent sensors for monitoring room occupancy (e.g., motion sensors) and those for registering the status of a door lock. Following the approach sketched so far, we would need to compose such primitive events into a publishable data item to which processes can then subscribe. Event composition turns out to be a difficult task, notably when the primitive events are generated from sources dispersed across the distributed system.
- Clearly, in publish-subscribe systems such as these, the crucial issue is the efficient and scalable implementation of matching subscriptions to notifications. From the outside, the publish-subscribe architecture provides lots of potential for building very large-scale distributed systems due to the strong decoupling of processes. On the other hand, devising scalable implementations without losing this independence is not a trivial exercise, notably in the case of content-based publish-subscribe systems.

Middleware organization

- The actual organization of middleware, that is, independent of the overall organization of a distributed system or application.
- In particularly, there are two important types of design patterns that are often applied to the organization of middleware:
  - Wrappers
  - Interceptors.
- Each targets different problems yet addresses the same goal for middleware: achieving openness. However, it can be argued that the ultimate openness is achieved when we can compose middleware at runtime.

Wrappers

- When building a distributed system out of existing components, we immediately bump into a fundamental problem: the interfaces offered by the legacy component are most likely not suitable for all applications.
- 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. An often-used type is a message broker. 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 $2*N = O(N)$ wrappers instead of $O(N^2)$. This situation is sketched in Figure 2.13.

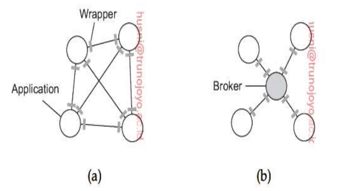


Figure 2.13: (a) Requiring each application to have a wrapper for each other application. (b) Reducing the number of wrappers by making use of a broker.

Interceptors

- Conceptually, an interceptor is nothing but 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. As we explain in detail later in the book, such a remote-object invocation is carried out in three steps:
  - 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.
  - 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.
  - 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.

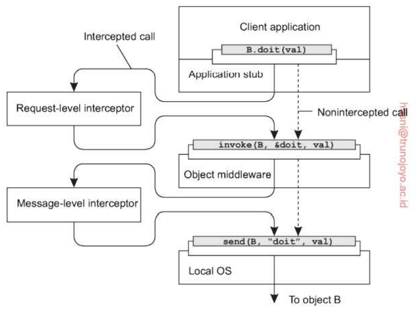


Figure 2.14: Using interceptors to handle remote-object invocations.

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. And indeed, perhaps one of the most popular approaches toward modifiable middleware is that of dynamically constructing middleware from components.
- 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. 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.

*System architecture

- Now that we have briefly discussed some commonly applied architectural styles, let us take a look at how many distributed systems are actually organized by considering where software components are placed.
- Deciding on software components, their interaction, and their placement leads to an instance of a software architecture, also known as a **system architecture**.
- We will discuss **centralized** and **decentralized** organizations, as well as various **hybrid forms**.

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.
- In the following, we first consider a **simple layered** organization, followed by looking at **multi-layered** organizations.

*Simple client-server architecture

- 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 Figure 2.15 in the form of a message sequence chart.

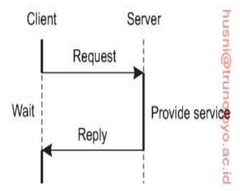


Figure 2.15: General interaction between a client and a server.

- 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. Unfortunately, making the protocol resistant to occasional transmission failures is not trivial. The only thing we can do is possibly let the client resend the request when no reply message comes in. The problem, however, is that the client cannot detect whether the original request message was lost, or that transmission of the reply failed. If the reply was lost, then resending a request may result in performing the operation twice. If the operation was something like "transfer $10,000 from my bank account," then clearly, it would have been better that we simply reported an error instead. On the other hand, if the operation was "tell me how much money I have left," it would be perfectly acceptable to resend the request. When an operation can be repeated multiple times without harm, it is said to be idempotent. Since some requests are idempotent and others are not it should be clear that there is no single solution for dealing with lost messages.
- 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. For example, virtually all Internet application protocols are based on reliable TCP/IP connections. In this case, whenever a client requests a service, it first sets up a connection to the server before sending the request. The server generally uses that same connection to send the reply message, after which the connection is torn down. The trouble may be that setting up and tearing down a connection is relatively costly, especially when the request and reply messages are small.
- The client-server model has been subject to many debates and controversies over the years. One of the main issues was how to draw a clear distinction between a client and a server. Not surprisingly, there is often no clear distinction. For example, a server for a distributed database may continuously act as a client because it is forwarding requests to different file servers responsible for implementing the database tables. In such a case, the database server itself only processes the queries.

*Multitiered Architectures

- The distinction into three logical levels as discussed so far, 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:
  - A client machine containing only the programs implementing (part of) the user-interface level.
  - 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. As explained in Section 2.1, many distributed applications are divided into the three layers (1) user interface layer, (2) processing layer, and (3) data layer. One approach for organizing clients and servers is then to distribute these layers across different machines, as shown in Figure 2.16. 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.

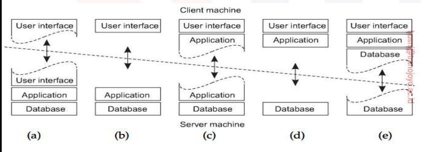


Figure 2.16: Client-server organizations in a two-tiered architecture.

- One possible organization is to have only the terminal-dependent part of the user interface on the client machine, as shown in Figure 2.16(a), 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, as shown in Figure 2.16(b). 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, as shown in Figure 2.16(c). 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. Another example of the organization of Figure 2.16(c), is that of a word processor in which the basic editing functions execute on the client side where they operate on locally cached, or in-memory data, but where the advanced support tools such as checking the spelling and grammar execute on the server side.
- In many client-server environments, the organizations shown in Figure 2.16(d) and Figure 2.16(e) 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. Figure 2.16(e) 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.

*Multitiered Architectures

- When distinguishing only client and server machines as we did so far, we miss the point that a server may sometimes need to act as a client, as shown in Figure 2.17, leading to a (physically) three-tiered architecture.

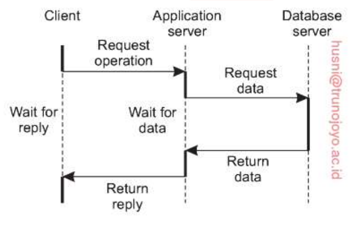


Figure 2.17: An example of a server acting as client.

- In this architecture, traditionally programs that form part of the processing layer are executed by a separate server but may additionally be partly distributed across the client and server machines. A typical example of where a three-tiered architecture is used is in transaction processing. A separate process, called the transaction processing monitor, coordinates all transactions across possibly different data servers.
- Another, but very different example were we often see a three-tiered architecture is in the organization of Web sites. In this case, a Web server acts as an entry point to a site, passing requests to an application server where the actual processing takes place. This application server, in turn, interacts with a database server. For example, an application server may be responsible for running the code to inspect the available inventory of some goods as offered by an electronic bookstore. To do so, it may need to interact with a database containing the raw inventory data.

*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. The characteristic feature of vertical distribution is that it is achieved by placing logically different components on different machines. The term is related to the concept of vertical fragmentation as used in distributed relational databases, where it means that tables are split column-wise, and subsequently distributed across multiple machines.
- Again, 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. In this section we will take a look at a 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 (which is also referred to as acting as a servant).
- Given this symmetric behavior, peer-to-peer architectures evolve around the question how to organize the processes in an overlay network in which the nodes are formed by the processes and the links represent the possible communication channels (which are often realized as TCP connections). A node may not be able to communicate directly with an arbitrary other node but is required to send messages through the available communication channels. Two types of overlay networks exist: those that are structured and those that are not.

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 adheres 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.

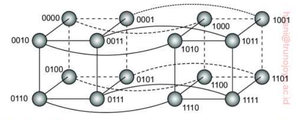


Figure 2.18: A simple peer-to-peer system organized as a four-dimensional hypercube.

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[<u, v>]. 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 and random walk*

Flooding and Random walks

- **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*** (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. A simple example of such an organization is shown in Figure 2.20. 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. 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. This problem is closely related to the leader-election problem.

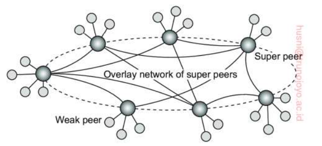


Figure 2.20: A hierarchical organization of nodes into a super-peer network.

*Hybrid Architectures

- So far, we have focused on client-server architectures and a number of peer-to-peer architectures. Many distributed systems combine architectural features, as we already came across in super-peer networks.
- In this section we take a look at some specific classes of distributed systems in which client-server solutions are combined with decentralized architectures.

*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

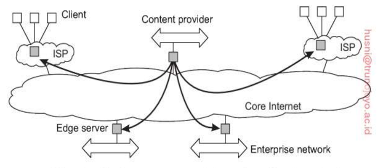


Figure 2.21: Viewing the Internet as consisting of a collection of edge servers.

- 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.
- 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.

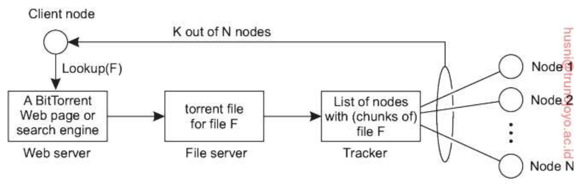


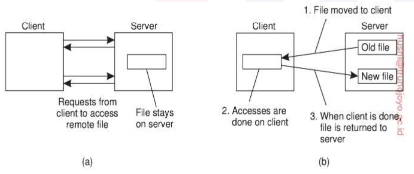
Figure 2.22: The principal working of BitTorrent [adapted with permission from Pouwelse et al. [2005]].

Example architectures

- The Network File System
- The Web

The Network File System

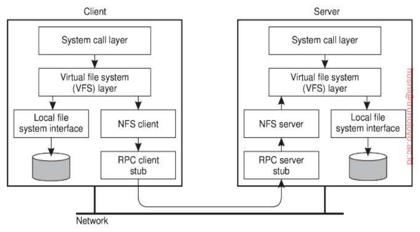
- Many distributed files systems are organized along the lines of client-server architectures, with Sun Microsystem's Network File System (NFS) being one of the most widely-deployed ones for Unix systems.
- The model underlying NFS and similar systems is that of a remote file service. In this model, 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. It is shown in Figure 2.24(a).
- In contrast, in the upload/download model a client accesses a file locally after having downloaded it from the server, as shown in Figure 2.24(b) 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.



Client
Server
Requests from client to access remote file
File stays on server
(a)
1. File moved to client
Client
Server
Old file
New file
2. Accesses are done on client
3. When client is done, file is returned to server
(b)

Figure 2.24: (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. For virtually all modern Unix systems, NFS is generally implemented following the layered architecture shown in Figure 2.25.



Client
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
Server
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

Figure 2.25: The basic NFS architecture for Unix systems.

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
  - Multitiered architectures

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 **Hyper-Text Transfer Protocol** (***HTTP***). This leads to the overall organization shown in Figure 2.27.

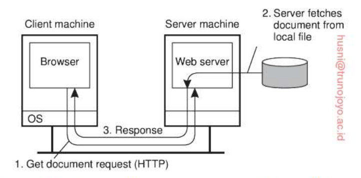


Figure 2.27: The overall organization of a traditional Web site.

Multitiered architectures

- The Web started out as the relatively simple two-tiered client-server system shown in Figure 2.27. By now, this simple architecture has been extended to support much more sophisticated means of documents. In fact, one could justifiably argue that the term "document" is no longer appropriate. For one, 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 2.28.

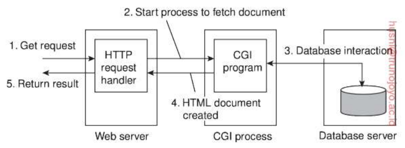


Figure 2.28: The principle of using server-side CGI programs.

*Summary

- Distributed systems can be organized in many different ways. We can make a distinction between software architecture and system architecture. The latter considers where the components that constitute a distributed system are placed across the various machines. The former is more concerned about the logical organization of the software: how do components interact, in what ways can they be structured, how can they be made independent, and so on.
- A keyword when talking about architectures is architectural style. A style reflects the basic principle that is followed in organizing the interaction between the software components comprising a distributed system. Important styles include layering, object-based styles, resource-based styles, and styles in which handling events are prominent.
- There are many different organizations of distributed systems. An important class is where machines are divided into clients and servers. A client sends a request to a server, who will then produce a result that is returned to the client. The client-server architecture reflects the traditional way of modularizing software in which a module calls the functions available in another module. By placing different components on different machines, we obtain a natural physical distribution of functions across a collection of machines.
- Client-server architectures are often highly centralized. In decentralized architectures we often see an equal role played by the processes that constitute a distributed system, also known as peer-to-peer systems. In peer-to-peer systems, the processes are organized into an overlay network, which is a logical network in which every process has a local list of other peers that it can communicate with. The overlay network can be structured, in which case deterministic schemes can be deployed for routing messages between processes. In unstructured networks, the list of peers is more or less random, implying that search algorithms need to be deployed for locating data or other processes.
- In hybrid architectures, elements from centralized and decentralized organizations are combined. A centralized component is often used to handle initial requests, for example to redirect a client to a replica server, which, in turn, may be part of a peer-to-peer network as is the case in BitTorrent-based systems.