DISTRIBUTED SYSTEMS

Module-1

ARCHITECTURE

2.0 ARCHITECTURE

* 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.

2.1 ARCHITECTURAL STYLES

* Logical organization of a DS into software components is referred to as its **software architecture.**
* **Architecture 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.
* **Component:** A modular unit with well-defined required and provided interfaces that is replaceable within its environment.
* **Connector:** A mechanism that mediates communication, coordination, or cooperation among components. It allows the flow of control and data between components.
* **Important styles of architecture** for DS are discussed.

LAYERED ARCHITECTURES

➔ In layered architecture **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.

➔ There are three common cases as shown in figure- 2.1

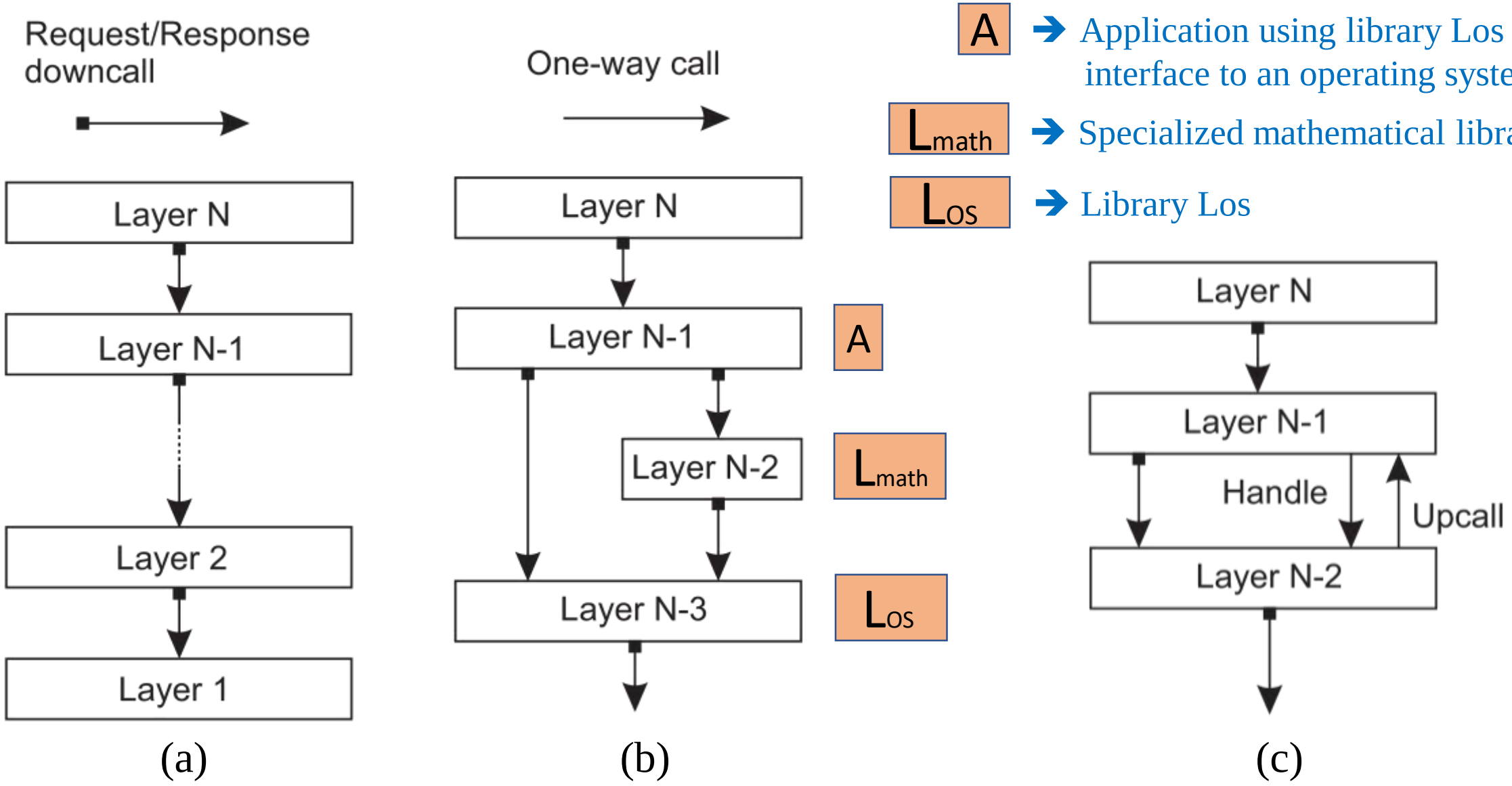


Figure 2.1: (a) Pure layered organization. (b) Mixed layered organization. (c) Layered organization with upcalls

Layered communication protocols

➔ Well-known architecture is ***Communication protocol stack.***

➔ **Communication services:** Each layer implements one or several communication services allowing data to be sent from a destination to one or several targets.

➔ **Interface**: Each layer offers an interface specifying the functions that can be called. **Interface hides the actual implementation of a service.**

➔ **Protocol:** Describes the rules that parties will follow in order to exchange information.

➔ **Transmission Control Protocol (TCP) –** Connection oriented

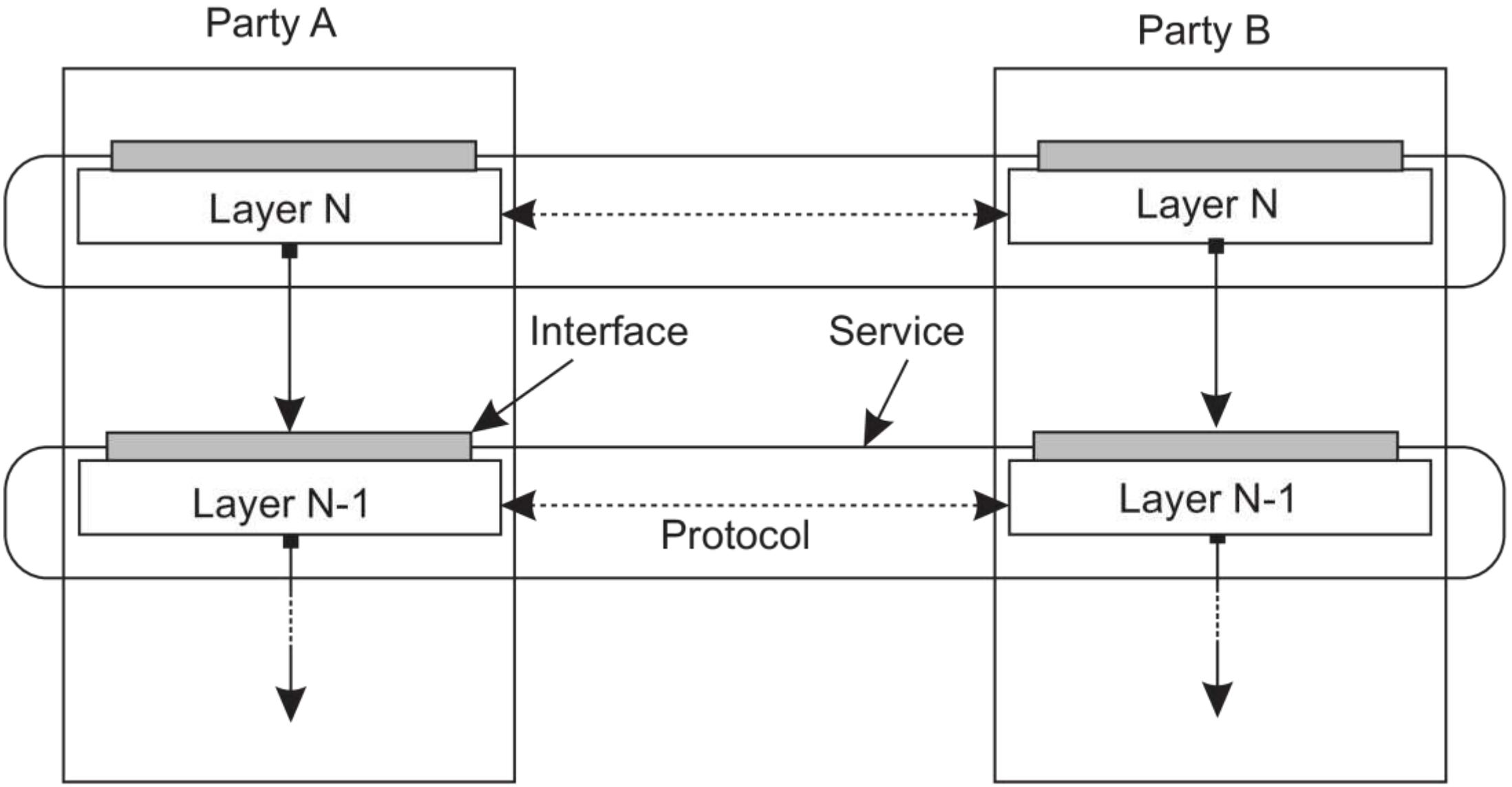


Figure 2.2: A layered communication-protocol stack, showing the difference between a service, its interface, and the protocol it deploys.

Two-party communication

Server

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

Client

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

Application Layering

➔ Logical layering of applications is viewed at three different layers.

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) A part that handles interaction with a user or some external application

b) A part that operates on a database or file system

c) A middle part that generally contains the core functionality of the application

Example: Internet search engine

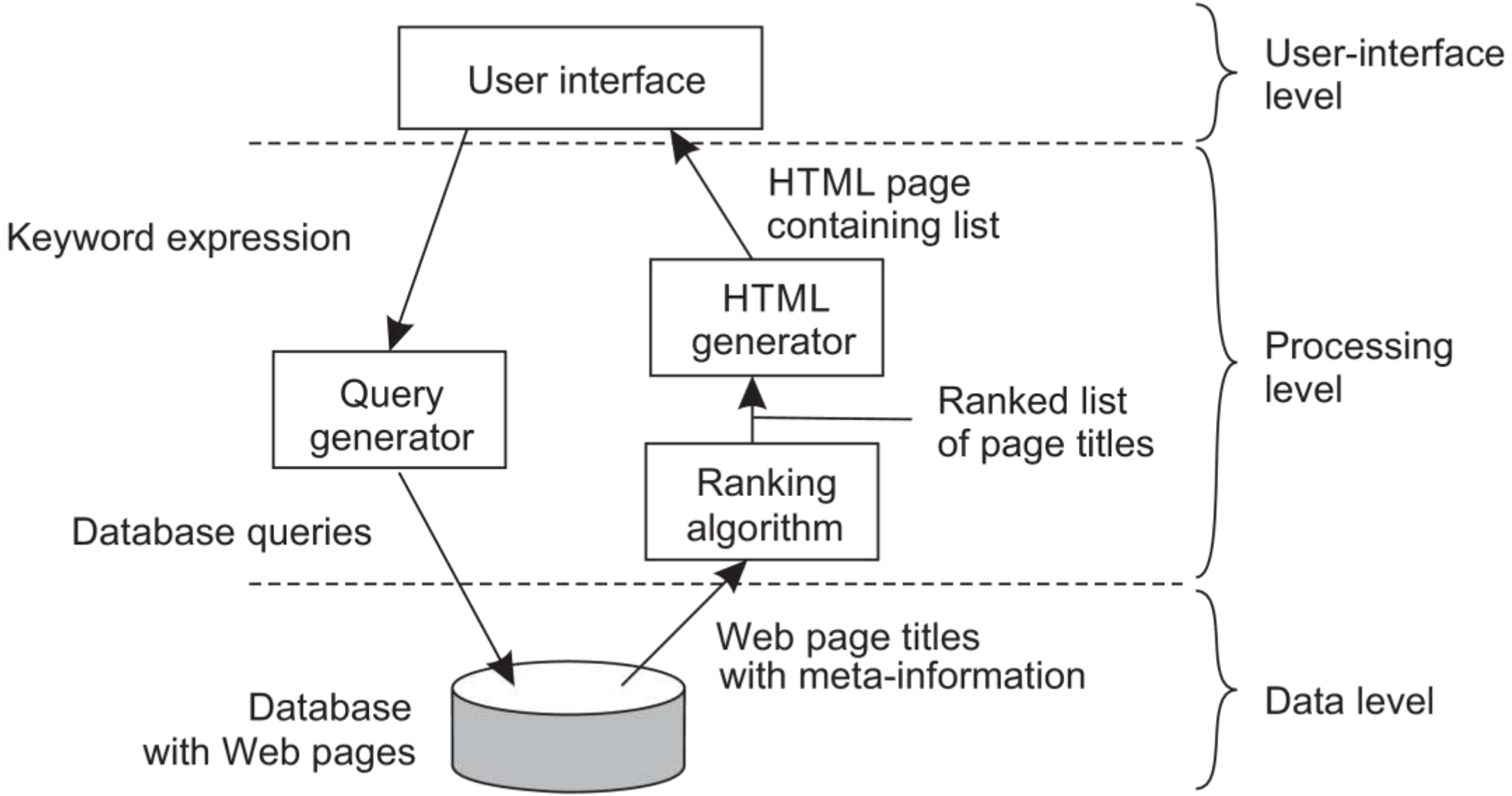


Figure 2.4: The simplified organization of an Internet search engine into three different layers.

Example-2: Decision support system for stock brokerage

The system can be divided into following three layers

1) A front end implementing the user interface or offering a programming interface to external applications
2) A back end for accessing a database with the financial data
3) The analysis programs between these two.

OBJECT BASED AND SERVICE ORIENTED ARCHITECTURES

➔ A far more loose organization is followed in object-based architectures

➔ Each object corresponds to a component and these components are connected through a procedure call mechanism.

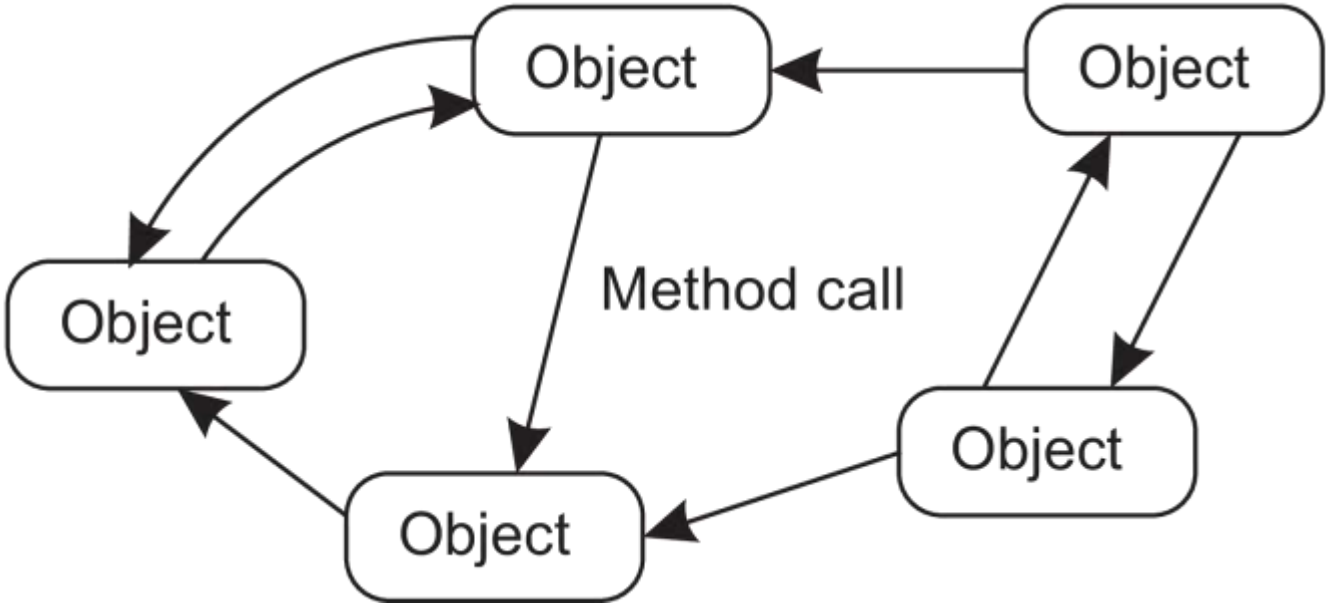


Figure 2.5: An object-based architectural style

- In a DS, 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 as they provide *encapsulation*.
- The interface offered by an object hides its implementation details.
- 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 is commonly referred to as a distributed object
- The ***state*** of distributed objects is not distributed and it resides at a single machine. The interfaces of objects are made available on other machines.

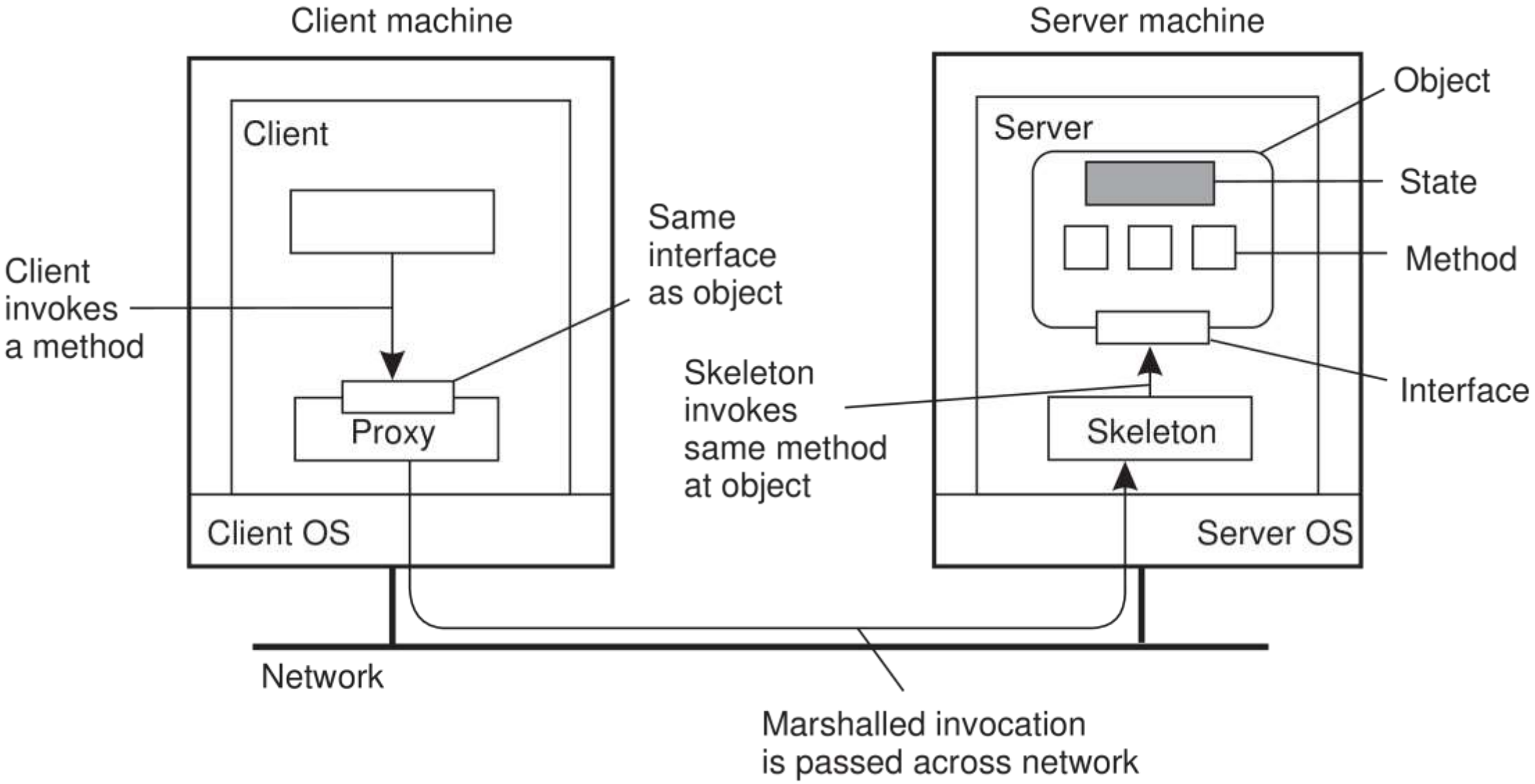


Figure 2.6: Common organization of a remote object with client-side proxy.

Service Oriented Architectures (SOAs)

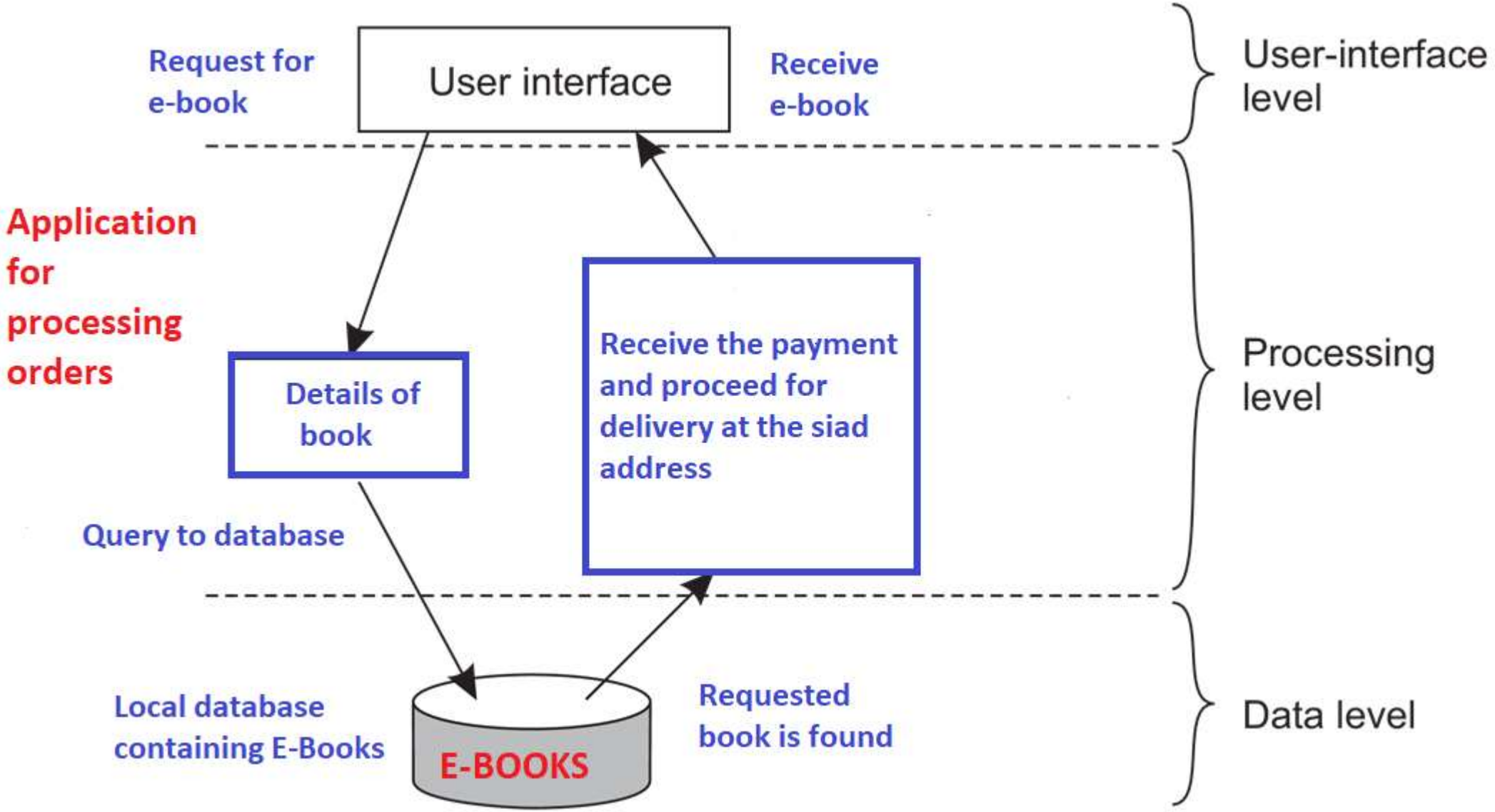
➔ Object-based architectures form the foundation of encapsulating services into independent units.

➔ In SOA, 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.

Example: An organization running its business application makes use of storage services offered by a cloud provider.

Web shop selling goods such as e-books



RESOURCE BASED ARCHITECTURES

- ➔ Distributed system can be viewed 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

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. The last property is also referred to as a stateless execution.

➔ To illustrate how RESTful works in practice, we consider a cloud storage service, such as Amazon's Storage Service (Amazon S3).

➔ Amazon S3 supports only two resources:

1) Objects: Which are essentially the equivalent of files.

2) Buckets: The equivalent of directories. Buckets into buckets is not allowed.

➔ An object named **ObjectName** contained in bucket **BucketName** 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. The protocol that is used with the service is HTTP.
- ➔ 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.
- ➔ Similarly, 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 ARCHITECTURE

➔ An architecture in which dependencies between processes is less.

➔ There is a strong separation between ***processing*** and ***coordination***.

➔ Here system is viewed as a collection of autonomously operating processes.

➔ In this model, **coordination** encompasses the **communication** and **cooperation** between processes.

➔ The activities performed by processes are grouped into a whole.

➔ The distinction between the coordination models is done 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.

➔ **Direct coordination:** When processes are **temporally** and **referentially** coupled, coordination takes place in a direct way.

➔ **Referential coupling:** Generally appears in the form of explicit referencing in communication.

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.

Example of direct communication: Talking over cell phones.

➔ **Mailbox coordination:** Processes are temporally decoupled, but referentially coupled.

➔ **Event-based coordination:** Referentially decoupled and temporally coupled systems form event based coordination. Processes will not know each other and processes can only publish notifications describing the occurrence of events.

➔ **Shared data space coordination:** Combination of referentially and temporally decoupled processes. Processes communicate by writing tuples to shared data space.

➔ **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.

➔ An important aspect of publish-subscribe systems is that communication takes place by **describing the events that a subscriber is interested in**.

The difference between a pure event-based architectural style, and that of a shared data space is shown in Figure 2.10.

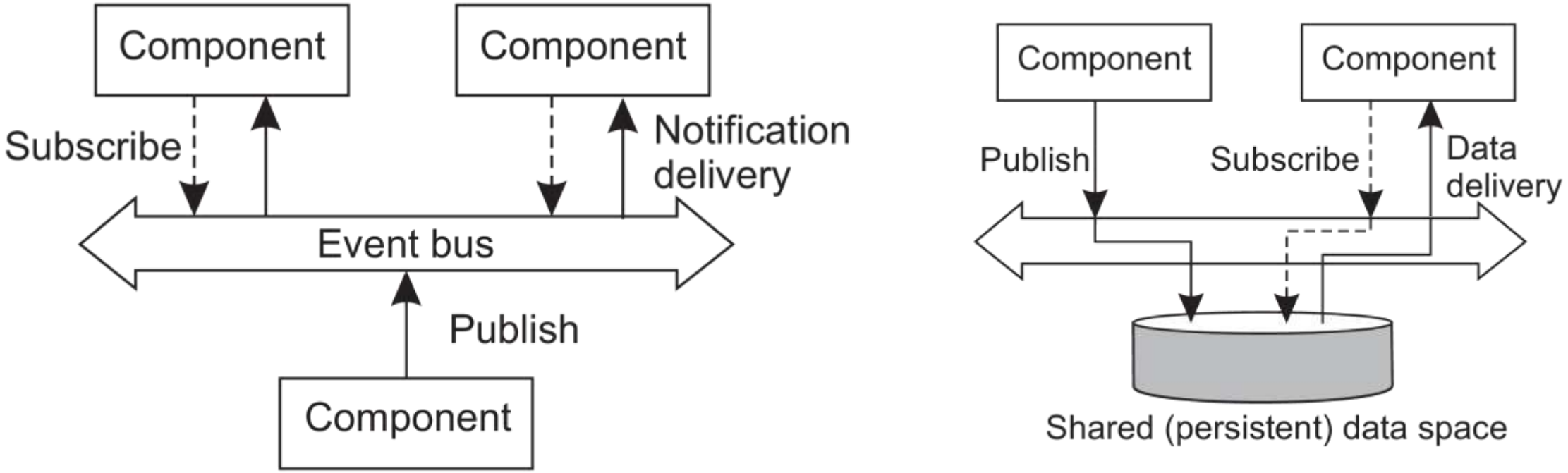


Figure 2.10: The (a) event-based and (b) shared data-space architectural style.

Example: Linda tuple space (Linda- A programming model)

Data space in Linda is known as a **tuple space**, which essentially supports three operations:

- in(t): remove a tuple that matches the template t
- rd(t): obtain a copy of a tuple that matches the template t
- out(t): add the tuple t to the tuple space

  - Calling out(t) twice in a row, leads to storing two copies of tuple t ⇒ a tuple space is modeled as a multiset.
  - Both in and rd are blocking operations: the caller will be blocked until a matching tuple is found, or has become available.

- Each message in a microblog application is modeled as a tuple : <string, string, string>
  First string names the poster, Second string represents the topic and third is the actual content.

- Assuming that we have created a shared data space called MicroBlog.
  Example below shows how Alice and Bob can post messages to that space,
  and how Chuck can pick a (randomly selected) message.

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 blog._out(("bob","distsys","I am studying chap 2"))
4 blog._out(("bob","distsys","The linda example's pretty simple"))
5 blog._out(("bob","gtcn","Cool book!"))
```

(a) Bob's code for creating a microblog and posting two messages.

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 blog._out(("alice","gtcn","This graph theory stuff is not easy"))
4 blog._out(("alice","distsys","I like systems more than graphs"))
```

(b) Alice's code for creating a microblog and posting two messages.

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 t1 = blog._rd(("bob","distsys",str))
4 t2 = blog._rd(("alice","gtcn",str))
5 t3 = blog._rd(("bob","gtcn",str))
```

(c) Chuck reading a message from Bob's and Alice's microblog.

Figure 2.11: A simple example of using a shared data space.

2.2 MIDDLEWARE ORGANIZATION

➔ Actual organization of middleware is independent of the overall organization of a distributed system or application.

➔ Two important types of design patterns are often applied to the organization of middleware: **Wrappers and Interceptors**.

➔ **Wrappers:** When building a DS out of existing components 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. It solves the problem of incompatible interfaces.

➔ Wrappers are much more than simple interface transformers.

Example: An **object adapter** is a component that allows applications to invoke remote objects.

➔ Wrappers help in extending systems with existing components.

Example: If an 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.

With this approach, for N applications we would, in theory, need to develop $N \times (N-1) = O(N^2)$ wrappers.

➔ **Broker:** A centralized component that handles all the accesses between different applications and helps to reduce number of wrappers.

➔ **Message broker:** A type of broker in which applications simply send requests to the broker containing information on what they need.

➔ Broker contacts the relevant applications and gives back result. (Figure 2.13)

➔ As broker offers a single interface to each application, we need at most **2N = O(N) wrappers**, instead of $O(N^2)$

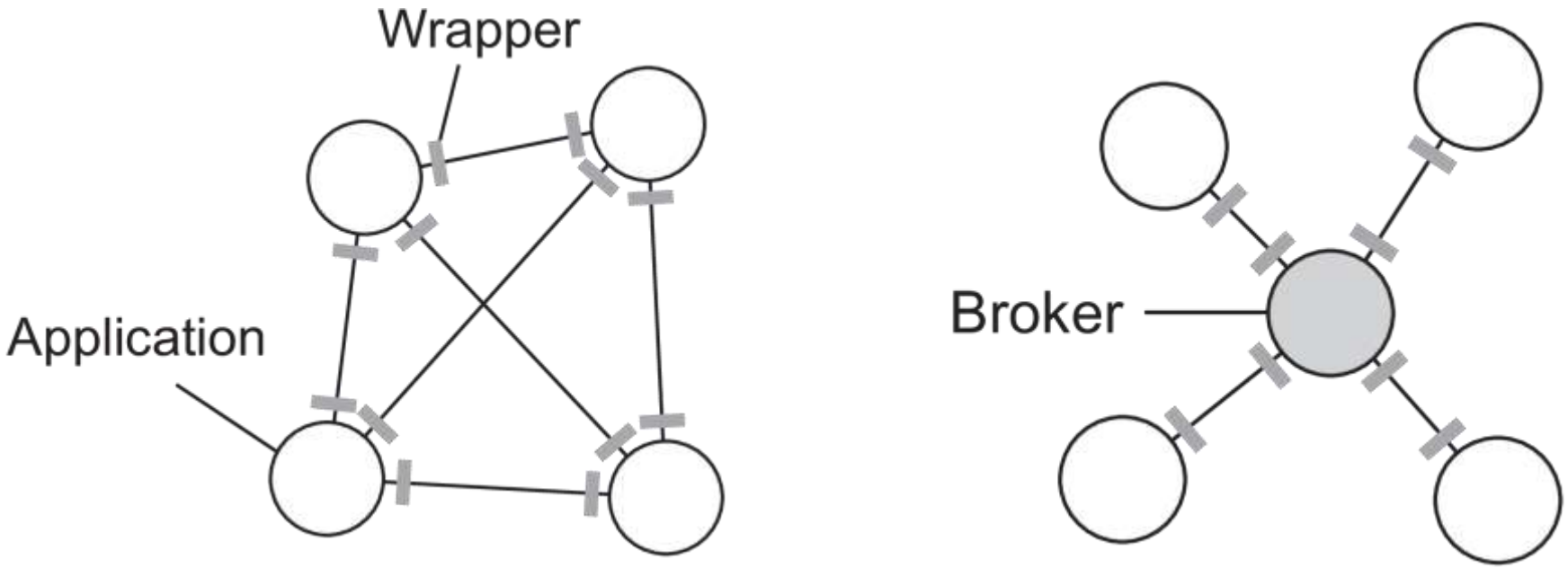


Figure 2.13: (a) Requiring each application to have a wrapper for each other application. (b) Reducing the number of wrappers by making use of a broker.

➔ **Interceptors:** A software construct that will break the usual flow of control and allow other (application specific) code to be executed. Used for adapting middleware to the specific needs of an application.

Basic idea: An object A can call a method that belongs to an object B, while the latter resides on a different machine than A. This remote object invocation is carried out in 3 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.

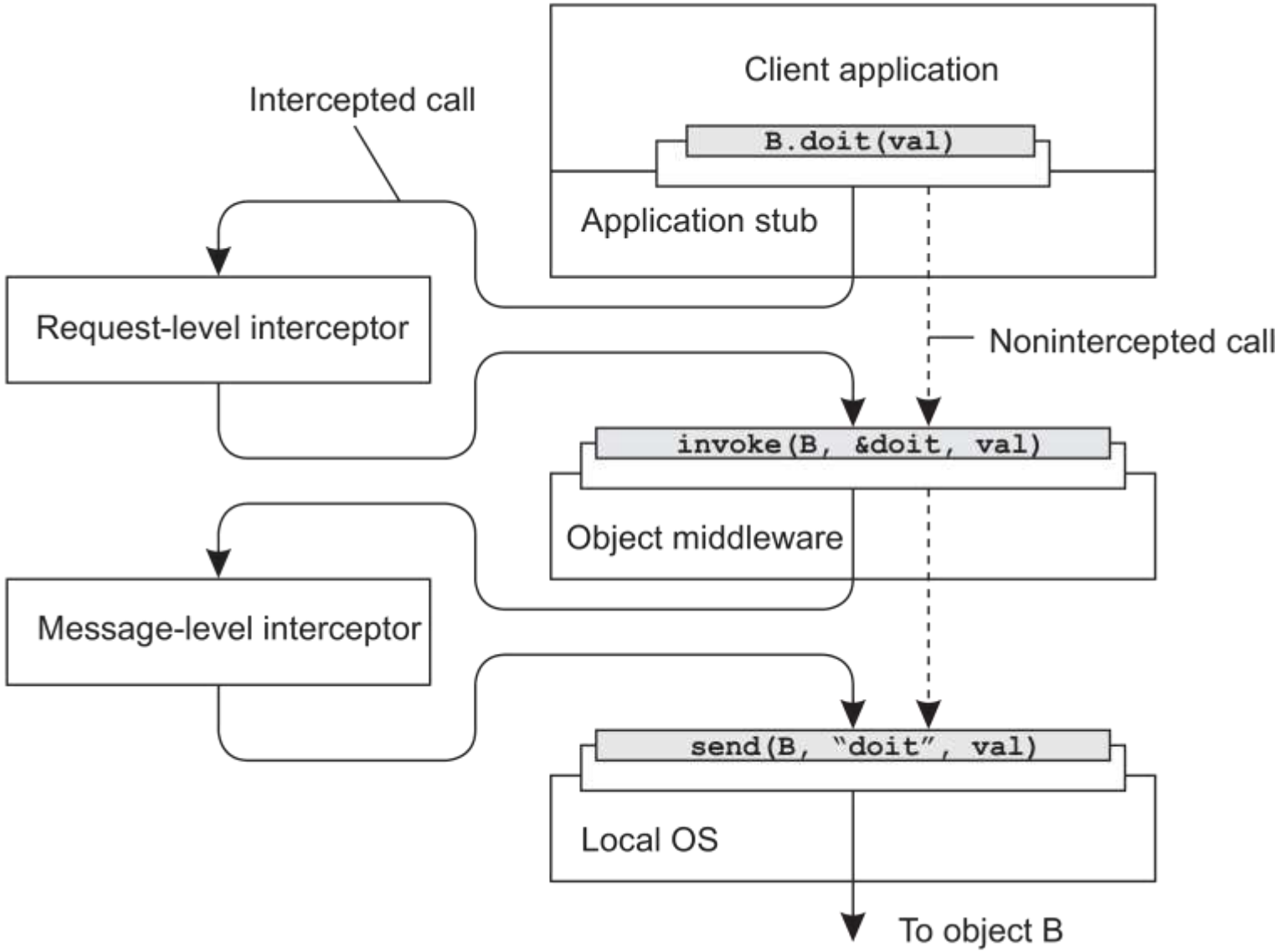


Figure 2.14: Using interceptors to handle remote-object invocations.

➔ After the first step, the call B.doit(val) is transformed into a generic call such as invoke(B, &doit, val) with a reference to B's method and the parameters that go along with the call.

➔ Now object B is replicated. In that case, each replica should actually be invoked. This is a clear point where interception can help. The request-level interceptor will call invoke(B, &doit, val) for each of the replicas.

➔ A call to a remote object will be sent over the network by using the messaging interface as offered by the local OS.

➔ At that level, a message-level interceptor may assist in transferring the invocation to the target object.

2.3 SYSTEM ARCHITECTURE

➔ Deciding on software components, their interaction, and their placement leads to an instance of a software architecture, also known as a system architecture.

CENTRALIZED ORGANIZATIONS

Simple layered organization and multi-layered organizations are discussed.

Simple client-server architecture

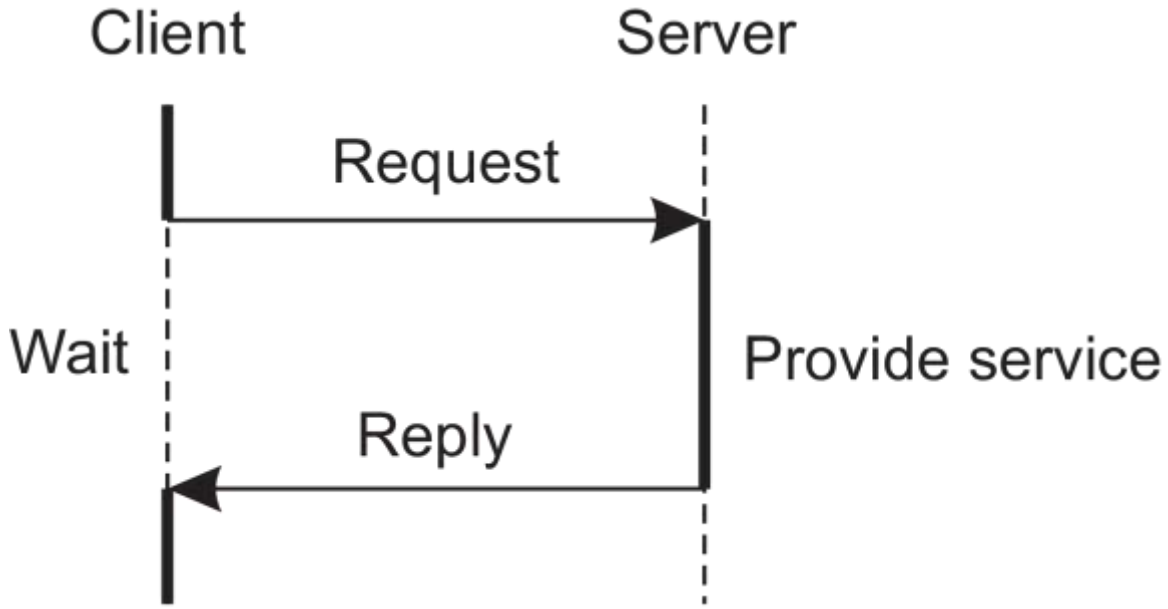
➔Processes in a DS are divided into two groups. (Client and Server)

➔ **Server:** A process implementing a specific service. **Example**: A database service.

➔ **Client:** A process that requests a service from a server by sending it a request and subsequently waiting for the server's reply.

➔ **Connectionless** protocol can be used for communication. (Not reliable)

➔ Many client-server systems use a reliable **connection oriented** protocol. (TCP/IP)



Issue with client server model: Clear distinction cannot be made between client and server.

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.

Multitiered Architectures

Based on 3 logical levels, a client server application can be physically distributed across several machines in different ways.

The simplest organization is to have 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 programs implementing the processing and data level

- Everything is handled by server and client has only graphical interface.
- The approach for organizing client and sever is to distribute the 3 layers (User interface layer, Processing layer and data layer) across different machines as shown in Figure 2.16.

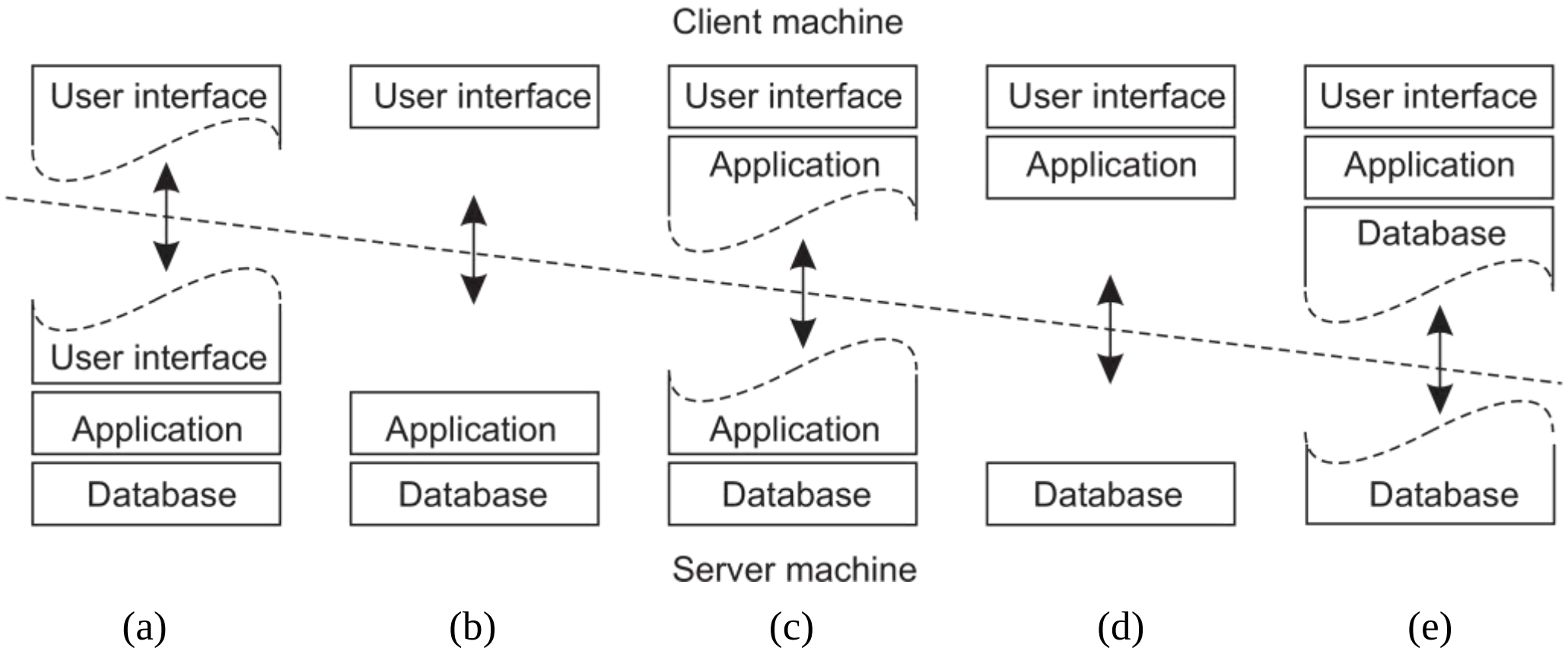


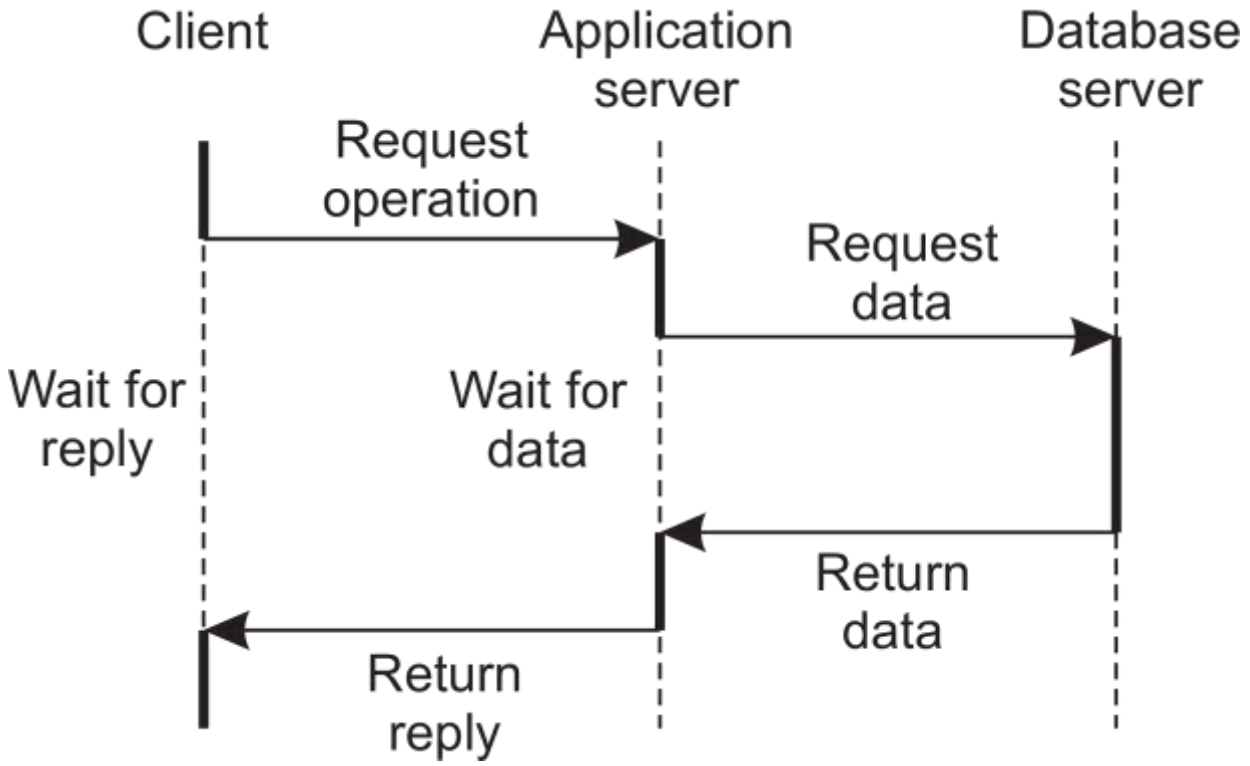
Figure 2.16: Client-server organizations in a two-tiered architecture.

Two-tiered architecture (Physically): We make the distinction between only two kinds of machines client and server.

Figure	Distribution of layers	
	Client machine	**Server machine**
2.16 (a)	Terminal-dependent part of the user interface	Applications remote control over the presentation of their data
2.16 (b)	Entire Graphical front end	Rest of the application and data
2.16 (c)	Entire Graphical front end and part of the application	Remaining application and database
	Example-1: Forms to be filled entirely before processing. **Example-2:** Word processor checking spellings at the client side.	

Figure	Distribution of layers	
	Client machine	**Server machine**
2.16 (d)	Most of the application is running on the client machine.	All operations on files or database entries
	Example: Banking applications run on an end-user's machine where the user prepares transactions. Once finished, the application contacts the database on the bank's server and uploads the transactions for further processing.	
2.16 (e)	Client's local disk contains part of the data.	Remaining part of the database
	Example: When browsing the Web, a client can gradually build a huge cache on local disk of most recent inspected Web pages.	

- **Three-tiered architecture (Physically):** A server may some times act as a client.



- **Example:** In organization of websites,

User ➔ Web server ➔ An application server ➔ Database server

Decentralized organizations: peer-to-peer systems

- **Vertical distribution:** The organization of client-server application as a multitiered architecture.
- In vertical distribution functions are logically and physically split across multiple machines.
- Vertical distribution is only one way of organizing client-server applications.
- **Horizontal distribution:** Distribution of the clients and the servers is referred as horizontal 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.
- An architecture supporting HD is **peer-to-peer systems.**

- Processes that constitute a peer-to-peer system are all equal and 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**).

- **Structured peer-to-peer systems:**

- Nodes 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.

- These systems use a semantic free index, in which each data item that is to be maintained by the system, is uniquely associated with a key, and that key is used as an index.

Data item ➔ Associated with key ➔Key is used as an index.

Hash function is used: **key(data item) = hash(data item's value).**

- P2p system is now responsible for storing (key, value) pairs.
- 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.
- The system provides an efficient implementation of a function lookup that maps a key to an existing node:

existing node = lookup(key).

Example : A hypercube which is nothing but an n-dimensional cube.

4D hypercube is viewed as 2 ordinary cubes, each with 8 vertices and 12 edges.

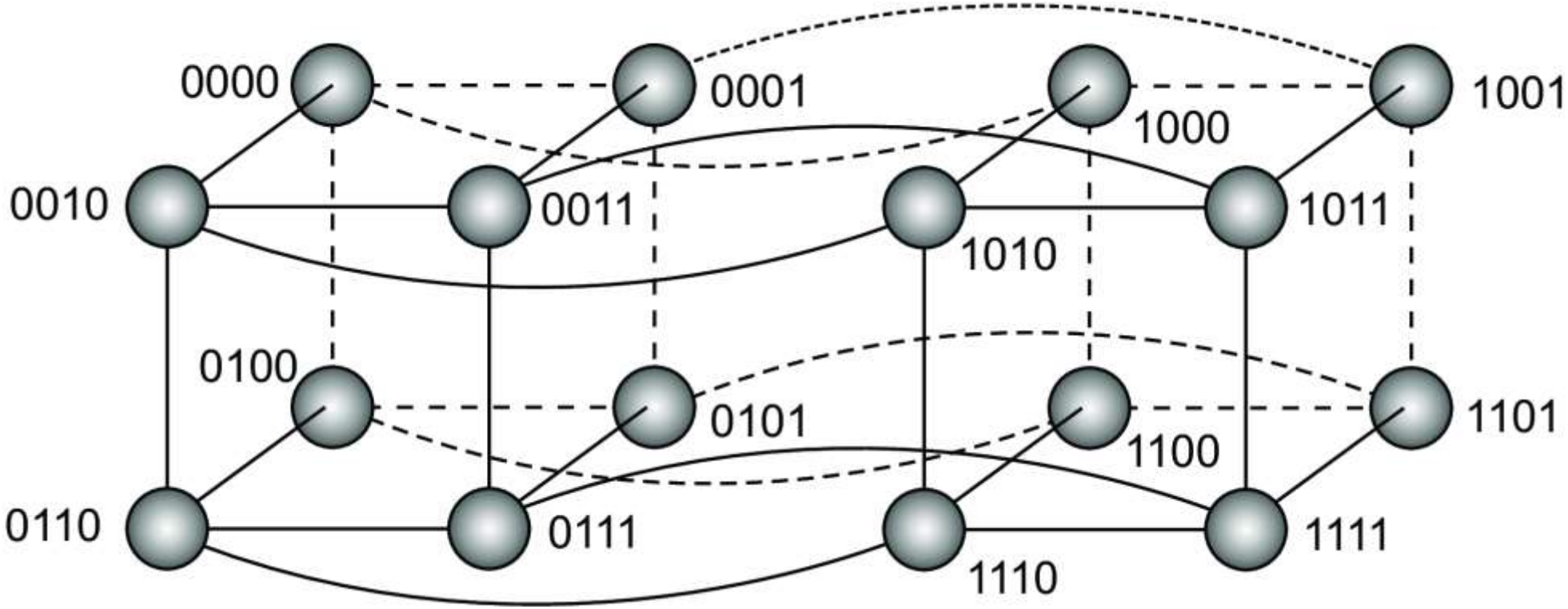


Figure 2.18: A simple peer-to-peer system organized as a four-dimensional hypercube.

- For this (admittedly naive) system, each data item is associated with one of the 16 nodes. This can be achieved by hashing the value of a data item to a

$$\text{key } k \in \{0, 1, 2, \ldots, 2^4 - 1\}$$

- Now suppose that the node with identifier 0111 is requested to look up the data having key 14, corresponding to the binary value 1110. We assume that the node with identifier 1110 is responsible for storing all data items that have key 14.
- What node 0111 can simply do, is forward the request to a neighbor that is closer to node 1110. In this case, this is either node 0110 or node 1111. If it picks node 0110, that node will then forward the request directly to node 1110 from where the data can be retrieved.

Unstructured peer-to-peer systems

- **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 **<u, v>** between two nodes **u** and **v** exists only with a certain probability **P[<u, v>].**
- When a node joins it often contacts a well-known node to obtain a starting list of other peers in the system.
- Looking up data cannot follow a predetermined route when lists of neighbors are constructed in an ad hoc fashion and we need to search for data.
- Two usual ways to search for data: **Flooding and Random Walk.**

- **Flooding:** In the case of flooding, an issuing node ***'u'*** simply passes a request for a data item to all its neighbors.
- **Time-to-Live (TTL)** is used to reduce the expense.

- **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.

Hierarchically organized peer-to-peer networks

- These are useful in the situations where peer-to-peer nature of the systems is not suitable.
- 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.
- **Broker:** Collects data on resource usage and availability for a number of nodes that are in each other's proximity.
- This will help to quickly select a node with sufficient resources.

- **Super peers:** Nodes maintaining an index or acting as a broker. Super peers are often organized in a peer-to-peer network. **(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.
- Whenever a weak peer joins the network, it attaches to one of the super peers and remains attached until it leaves the network.
- There is need that super peers should be long lived and highly available.
- Selection of a node as a super peer is a problem.

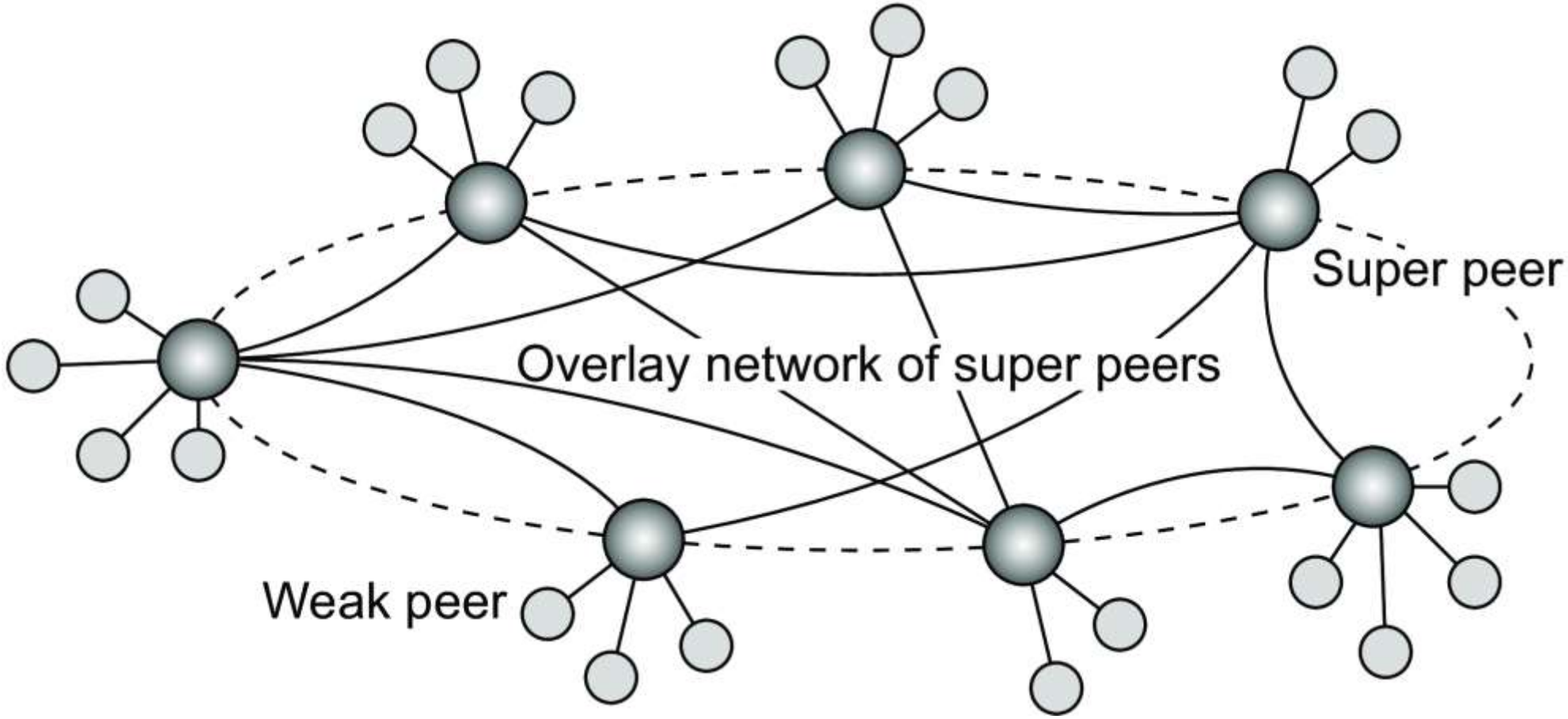


Figure 2.20: A hierarchical organization of nodes into a super-peer network.

Example: The skype network

Hybrid Architectures

Here client -server architectures are combined with decentralized architectures.

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.
- The general organization is shown in Figure 2.21.

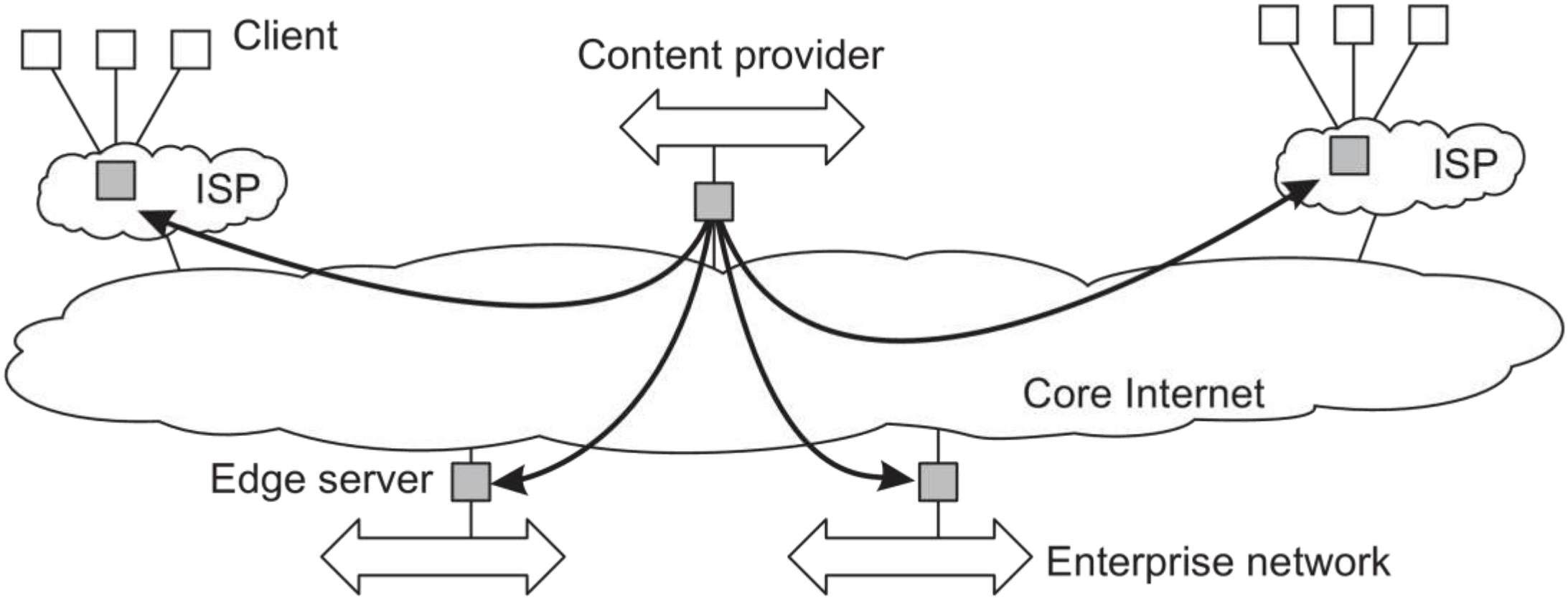


Figure 2.21: Viewing the Internet as consisting of a collection of edge servers.

- 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.
- For a specific organization, one edge server acts as an origin server from which all content originates.
- Origin server can use other edge servers for replicating Web pages.
- **Cloud computing:** Edge servers are used to assist in computations and storage, essentially leading to distributed cloud systems.
- **Fog computing:** 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.

Example: BitTorrent file-sharing system

- BitTorrent is a peer-to-peer file downloading system and its principal working is shown in Figure 2.22

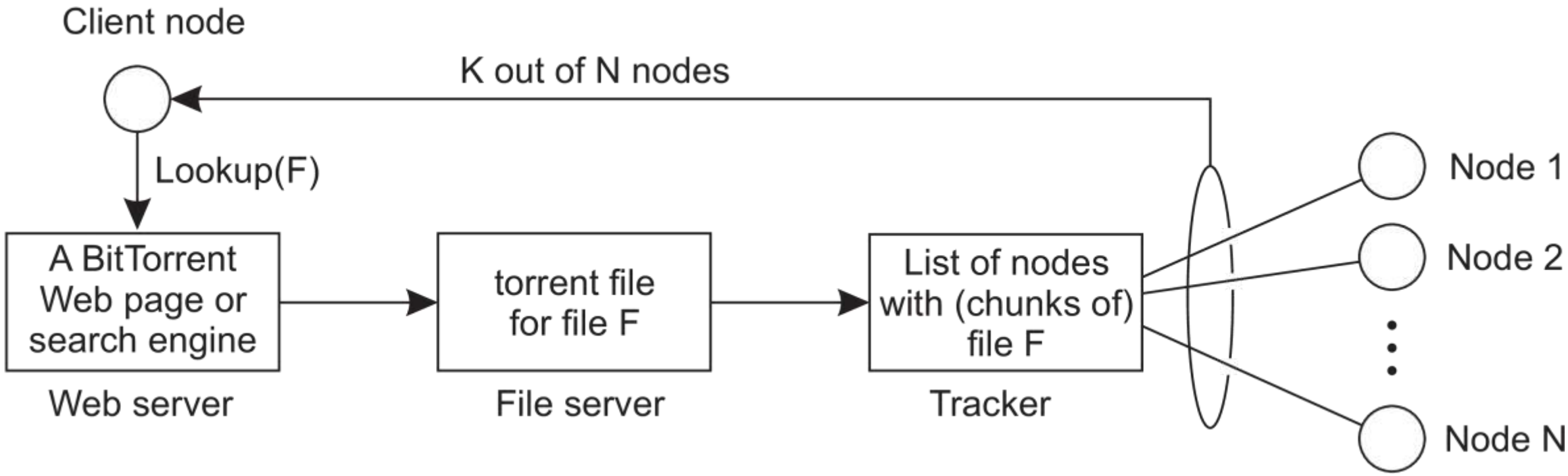


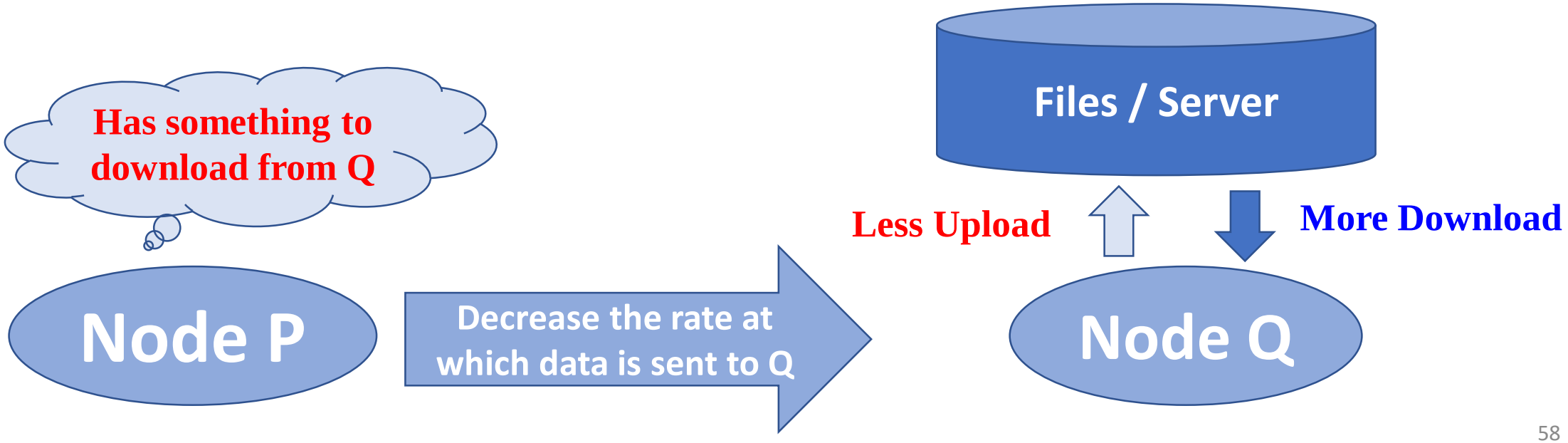
Figure 2.22: The principal working of BitTorrent

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.

- **Free riding:** A phenomenon in which a significant fraction of participants merely download files but contribute nothing.
- To prevent this situation, in BitTorrent a file can be downloaded only when the downloading client is providing content to someone else.
- **Downloading a file:** A user needs to access a **global directory -**Well know Web site. 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.

- 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.



2.4 EXAMPLE ARCHITECTURES

The Network File System (NFS)

➔ Sun microsystem's NFS is widely deployed architecture for Unix systems.

Basic idea: Each file server provides a standardized view of its local file system. Irrespective of how the local file system is implemented; each NFS server supports the same model.

The NFS model (Similar to Remote File Service)

➔ Clients are offered transparent access to a file system that is managed by a remote server.

➔ Location of files is not known to clients and they are given with only interface to the FS.

➔ Client is offered only an interface containing various file operations, but the server is responsible for implementing those operations. Hence this model is also referred to as the **remote access model. (Figure 2.24(a))**

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.

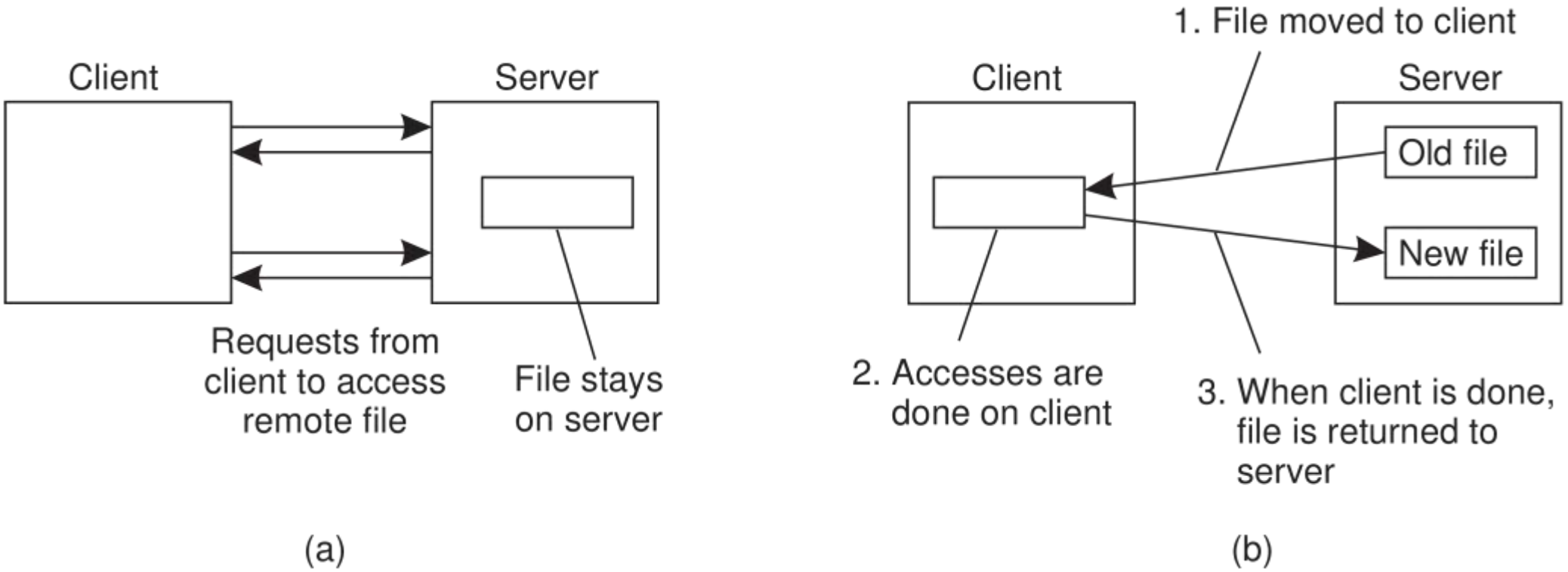


Figure 2.24: (a) The remote access model. (b) The upload/download model.

- In Unix systems NFS is generally implemented following the layered architecture as shown in Figure 2.25.

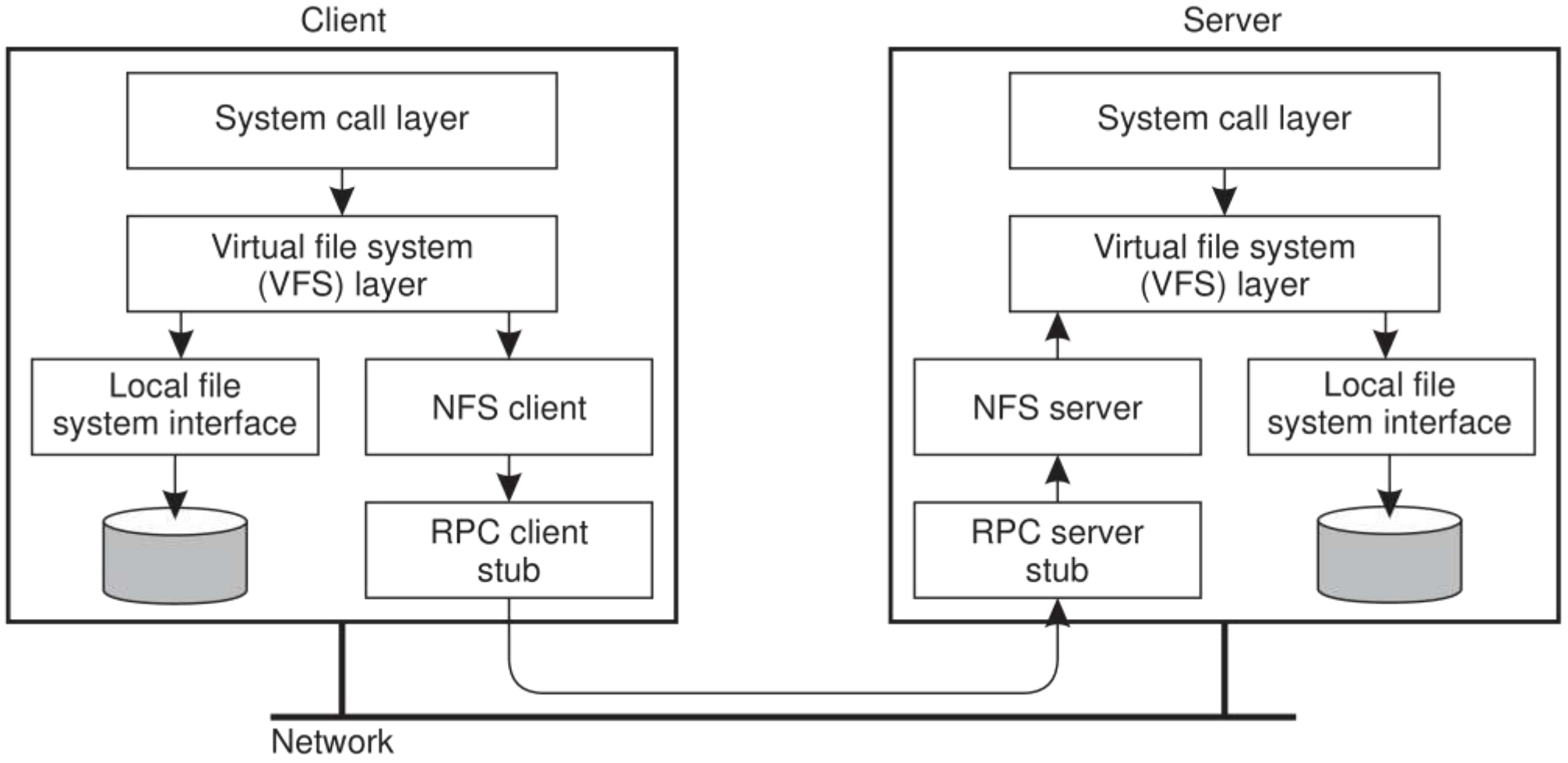


Figure 2.25: The basic NFS architecture for Unix systems.

- **Client side:** Accesses the FS using the system calls provided by its local OS. However, the local Unix FS interface is replaced by an interface to the Virtual File System (VFS).
- With NFS, operations on the VFS interface are either passed to a local file system, or passed to a separate component known as the NFS client, which takes care of handling access to files stored at a remote server.
- In NFS, all client-server communication is done through so-called remote procedure calls (RPCs).
- The NFS client implements the NFS file system operations as remote procedure calls to the server.

- **Server side:** The NFS server is responsible for handling incoming client requests.
- The RPC component at the server converts incoming requests to regular VFS file operations that are subsequently passed to the VFS layer.
- Again, the VFS is responsible for implementing a local file system in which the actual files are stored.
- An important advantage of this scheme is that NFS is largely independent of local file systems.

The Web

- The architecture of Web-based distributed systems is not fundamentally different from other distributed systems.

Simple Web-based systems

- 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).
- A client interacts with Web servers through a browser, which is responsible for properly displaying a document

- 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 2.27.

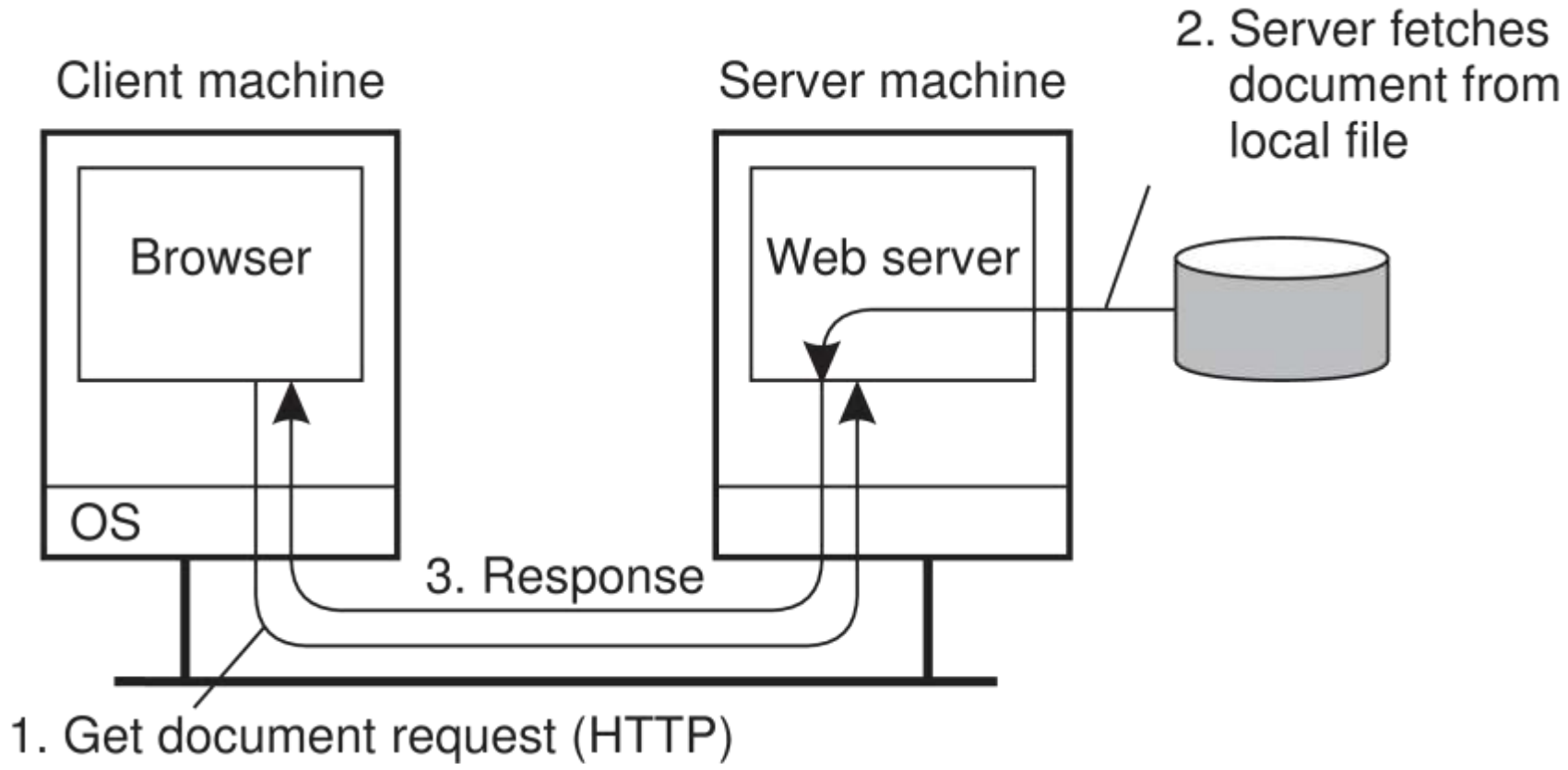


Figure 2.27: The overall organization of a traditional Web site.

Multitiered architectures

- One of the first enhancements to the basic architecture was support for simple user interaction by means of the **Common Gateway Interface (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.

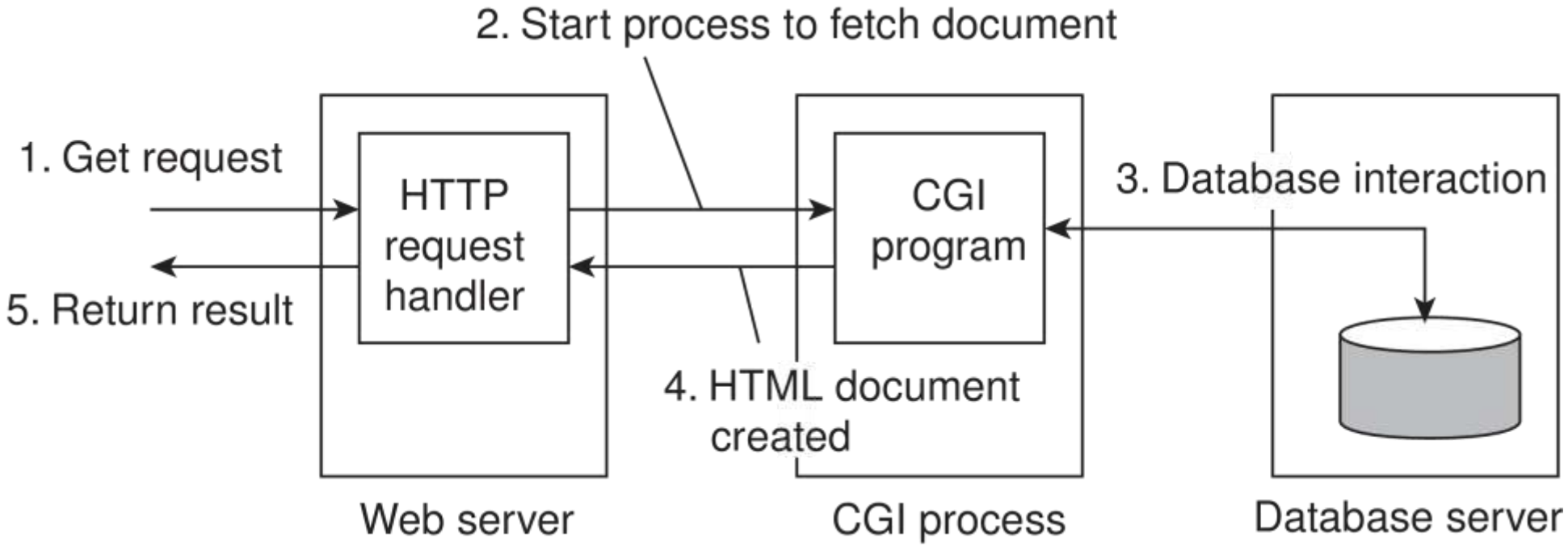


Figure 2.28: The principle of using server-side CGI programs.

- 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.