# Distributed Systems

## Subheadline


- Stefan Henkler **E-Mail:** stefan.henkler@hshl.de

# Overview

I. Introduction - parallel architectures
II. **Distributed Architectures**
III. GPU architecture and programming

# Overview

- ✧ Distributed systems
- ✧ Client–server computing
- ✧ Architectural patterns for distributed systems
- ✧ Software as a service

# Distributed systems

- Virtually all large computer-based systems are distributed systems.
  - "… a collection of independent computers that appears to the user as a single coherent system."
- Information processing is distributed over several computers rather than confined to a single machine.
- Embedeed systems are networks of systems
- Distributed software engineering is therefore very important for engineering technical systems

# Distributed system characteristics

- ✧ Resource sharing
  - Sharing of hardware and software resources.
- ✧ Openness
  - Use of equipment and software from different vendors.
- ✧ Concurrency
  - Concurrent processing to enhance performance.
- ✧ Scalability
  - Increased throughput by adding new resources.
- ✧ Fault tolerance
  - The ability to continue in operation after a fault has occurred.

# Distributed systems

# Distributed systems issues

- Distributed systems are more complex than systems that run on a single processor.
- Complexity arises because different parts of the system are independently managed as is the network.
- There is no single authority in charge of the system so top-down control is impossible.

# Design issues

- *Transparency* To what extent should the distributed system appear to the user as a single system?
- *Openness* Should a system be designed using standard protocols that support interoperability?
- *Scalability* How can the system be constructed so that it is scaleable?
- *Security* How can usable security policies be defined and implemented?
- *Quality of service* How should the quality of service be specified.
- *Failure management* How can system failures be detected, contained and repaired?

# Transparency

- Ideally, users should not be aware that a system is distributed and services should be independent of distribution characteristics.
- In practice, this is impossible because parts of the system are independently managed and because of network delays.
  - Often better to make users aware of distribution so that they can cope with problems
- To achieve transparency, resources should be abstracted and addressed logically rather than physically. Middleware maps logical to physical resources.

# Openness

- Open distributed systems are systems that are built according to generally accepted standards.
- Components from any supplier can be integrated into the system and can inter-operate with the other system components.
- Openness implies that system components can be independently developed in any programming language and, if these conform to standards, they will work with other components.
- Web service standards for service-oriented architectures were developed to be open standards.

# Scalability

- The scalability of a system reflects its ability to deliver a high-quality service as demands on the system increase
  - *Size* It should be possible to add more resources to a system to cope with increasing numbers of users.
  - *Distribution* It should be possible to geographically disperse the components of a system without degrading its performance.
  - *Manageability* It should be possible to manage a system as it increases in size, even if parts of the system are located in independent organizations.
- There is a distinction between scaling-up and scaling-out. Scaling up is more powerful system; scaling out is more system instances.

# Security

- When a system is distributed, the number of ways that the system may be attacked is significantly increased, compared to centralized systems.
- If a part of the system is successfully attacked then the attacker may be able to use this as a ‘back door’ into other parts of the system.
- Difficulties in a distributed system arise because different organizations may own parts of the system. These organizations may have mutually incompatible security policies and security mechanisms.

# Types of attack

- The types of attack that a distributed system must defend itself against are:
  - Interception, where communications between parts of the system are intercepted by an attacker so that there is a loss of confidentiality.
  - Interruption, where system services are attacked and cannot be delivered as expected.
    - Denial of service attacks involve bombarding a node with illegitimate service requests so that it cannot deal with valid requests.
  - Modification, where data or services in the system are changed by an attacker.
  - Fabrication, where an attacker generates information that should not exist and then uses this to gain some privileges.

# Quality of service

- The quality of service (QoS) offered by a distributed system reflects the system's ability to deliver its services dependably and with a response time and throughput that is acceptable to its users.
- Quality of service is particularly critical when the system is dealing with time-critical data such as sound or video streams or in safety critical environments
  - In these circumstances, if the quality of service falls below a threshold value then the sound or video may become so degraded that it is impossible to understand.

# Failure management

- In a distributed system, it is inevitable that failures will occur, so the system has to be designed to be resilient to these failures.

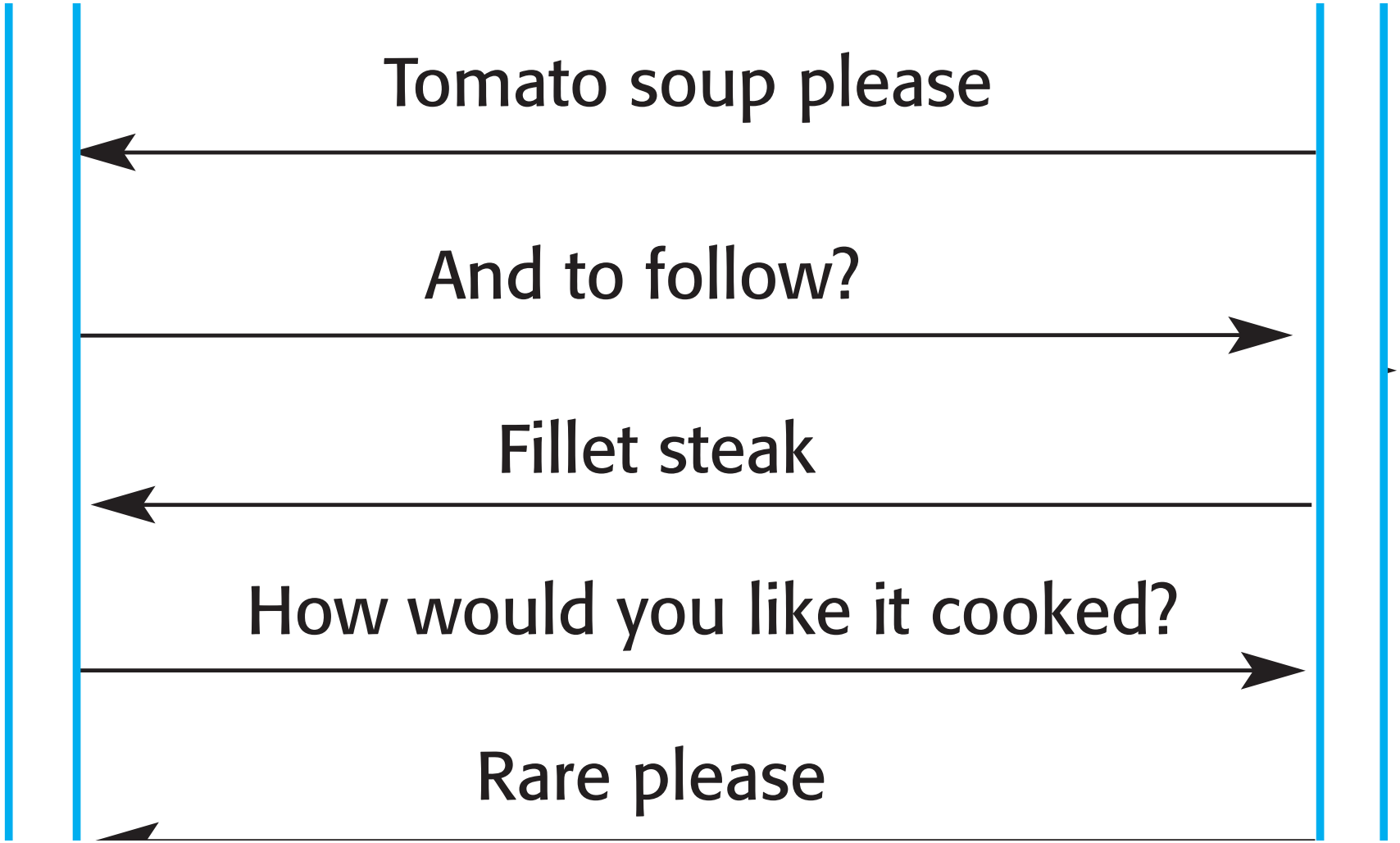
  *"You know that you have a distributed system when the crash of a system that you've never heard of stops you getting any work done."*

- Distributed systems should include mechanisms for discovering if a component of the system has failed, should continue to deliver as many services as possible in spite of that failure and, as far as possible, automatically recover from the failure.

# Models of interaction

- Two types of interaction between components in a distributed system
  - Procedural interaction, where one computer calls on a known service offered by another computer and waits for a response.
  - Message-based interaction, involves the sending computer sending information about what is required to another computer. There is no necessity to wait for a response.

# Procedural interaction between a diner and a waiter



# Remote procedure calls

- Procedural communication in a distributed system is implemented using remote procedure calls (RPC).
- In a remote procedure call, one component calls another component as if it was a local procedure or method. The middleware in the system intercepts this call and passes it to a remote component.
- This carries out the required computation and, via the middleware, returns the result to the calling component.
- A problem with RPCs is that the caller and the callee need to be available at the time of the communication, and they must know how to refer to each other.

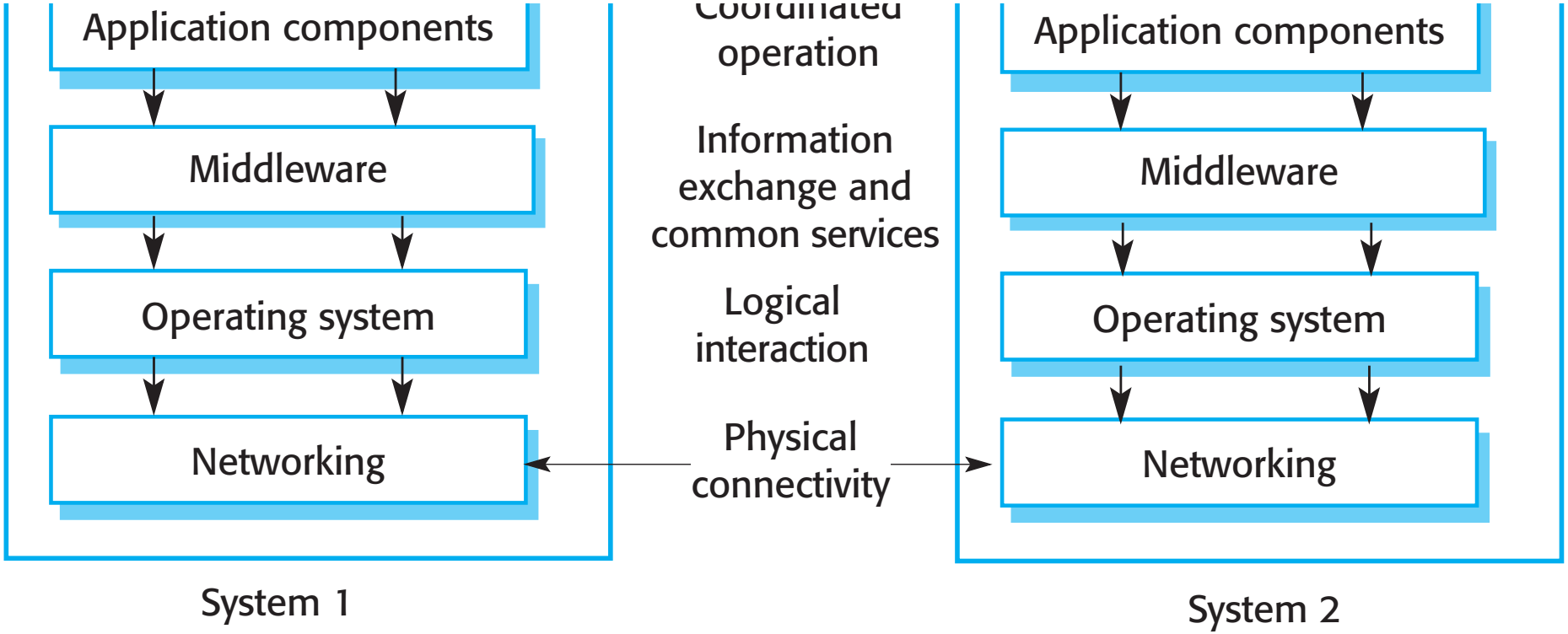
# Message passing

- Message-based interaction normally involves one component creating a message that details the services required from another component.
- Through the system middleware, this is sent to the receiving component.
- The receiver parses the message, carries out the computations and creates a message for the sending component with the required results.
- In a message-based approach, it is not necessary for the sender and receiver of the message to be aware of each other. They simple communicate with the middleware.

# Middleware

- The components in a distributed system may be implemented in different programming languages and may execute on completely different types of processor. Models of data, information representation and protocols for communication may all be different.
- Middleware is software that can manage these diverse parts, and ensure that they can communicate and exchange data.

# Middleware in a distributed system



# Middleware support

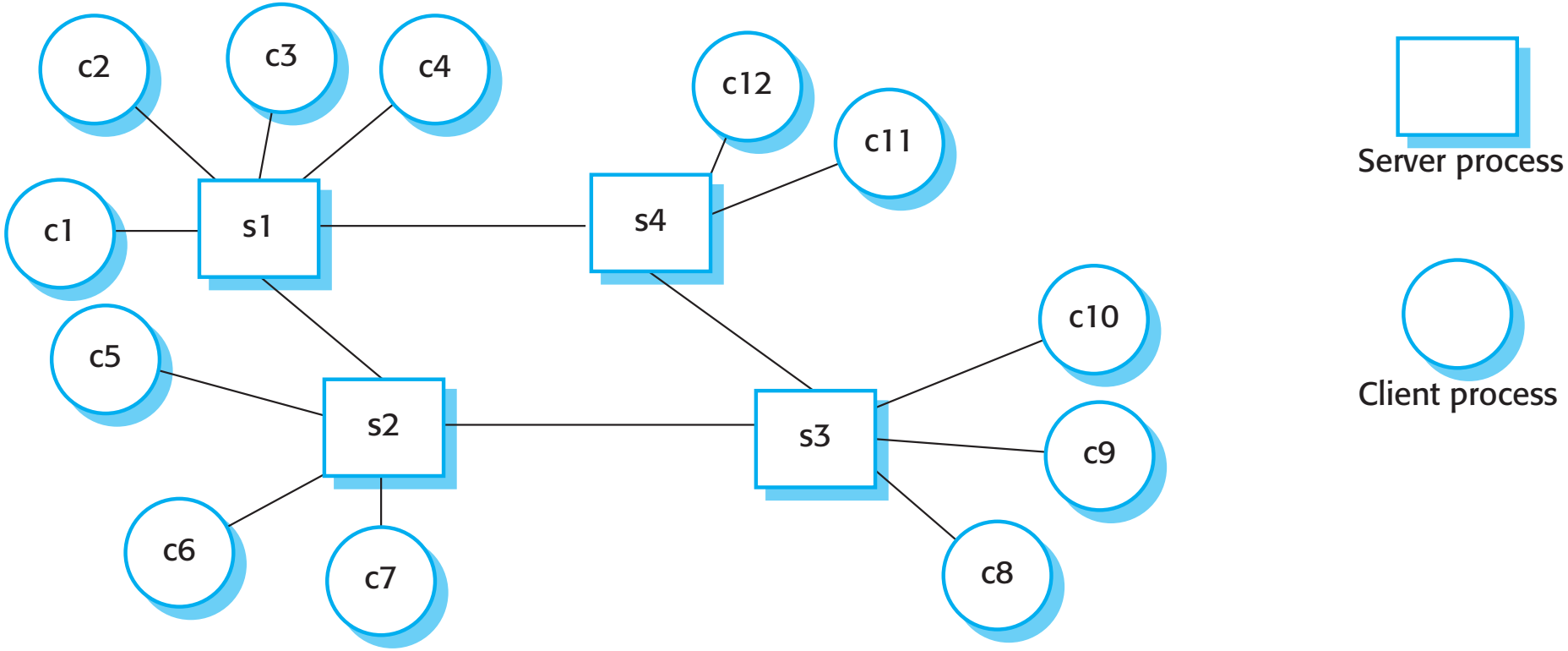
- Interaction support, where the middleware coordinates interactions between different components in the system
  - The middleware provides location transparency in that it isn't necessary for components to know the physical locations of other components.
- The provision of common services, where the middleware provides reusable implementations of services that may be required by several components in the distributed system.
  - By using these common services, components can easily inter-operate and provide user services in a consistent way.

# Client-server computing

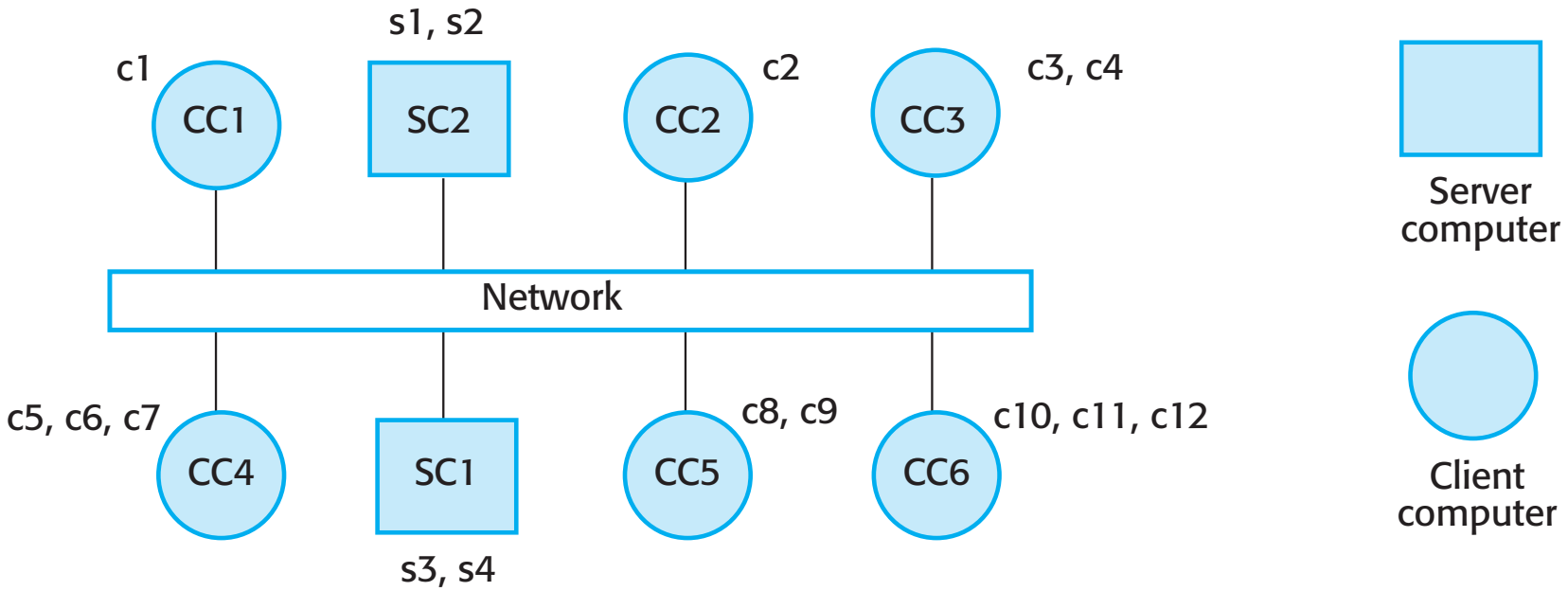
# Client-server computing

- Distributed systems that are accessed over the Internet are normally organized as client-server systems.
- In a client-server system, the user interacts with a program running on their local computer (e.g. a web browser or mobile application). This interacts with another program running on a remote computer (e.g. a web server).
- The remote computer provides services, such as access to web pages, which are available to external clients.

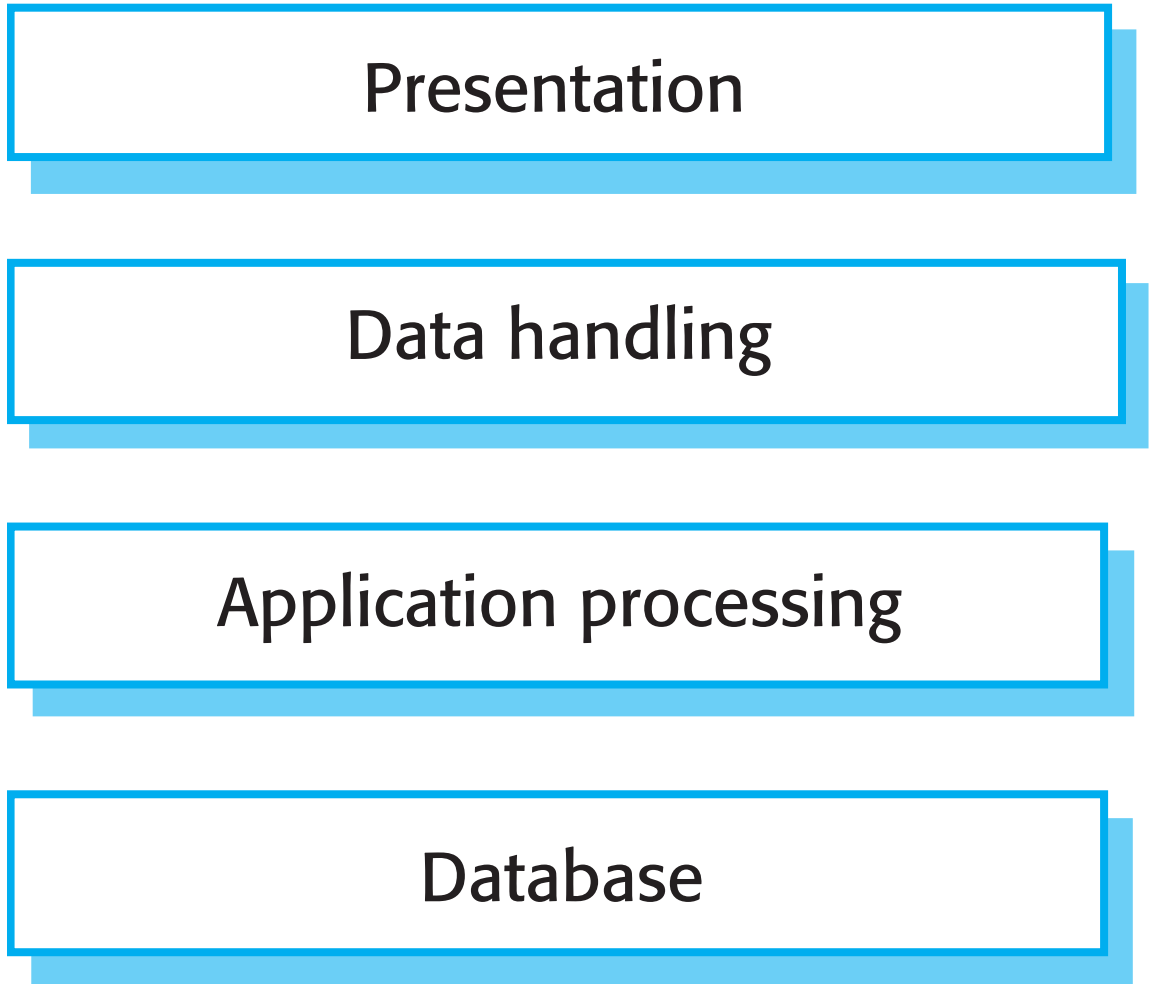
# Client–server interaction



# Mapping of clients and servers to networked computers



# Layered architectural model for client–server applications



# Layers in a client/server system

- *Presentation*
  - concerned with presenting information to the user and managing all user interaction.
- *Data handling*
  - manages the data that is passed to and from the client. Implement checks on the data, generate web pages, etc.
- Application processing layer
  - concerned with implementing the logic of the application and so providing the required functionality to end users.
- Database
  - Stores data and provides transaction management services, etc.

# Architectural patterns for distributed systems

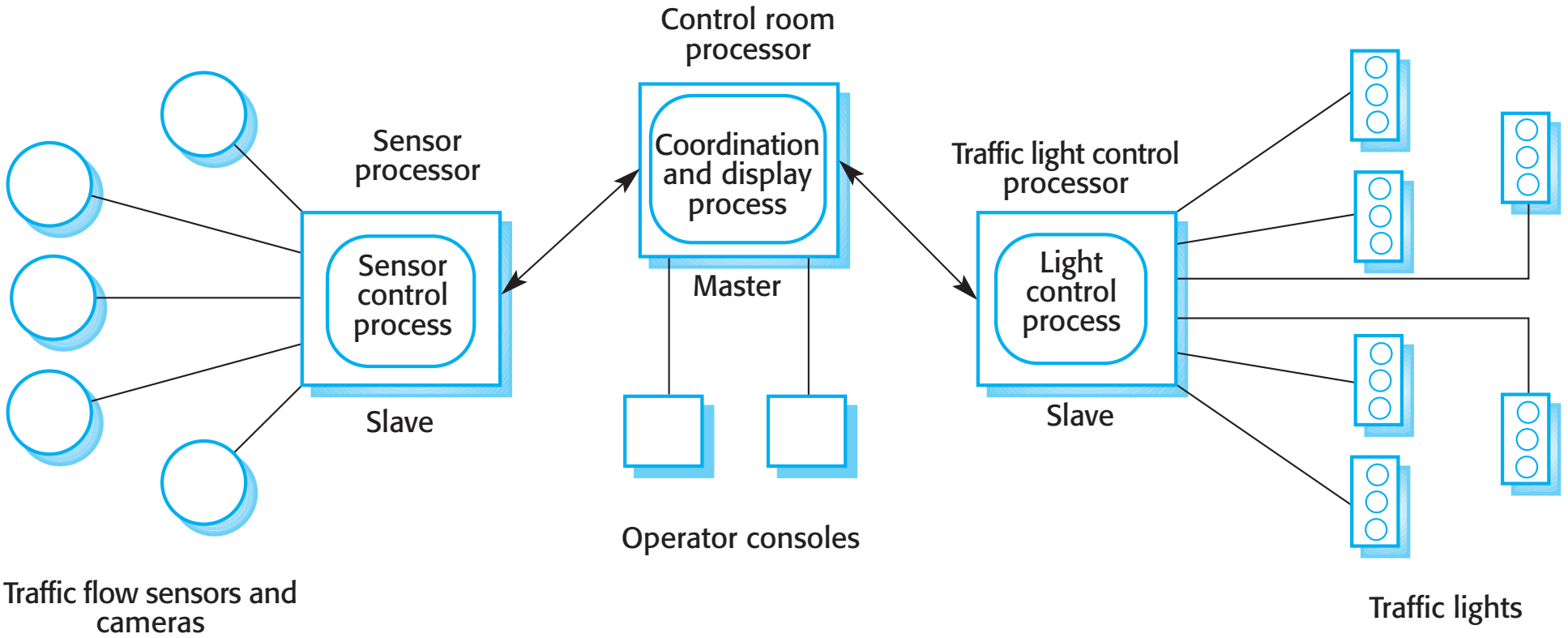
# Architectural patterns

- Widely used ways of organizing the architecture of a distributed system:
  - *Master-slave architecture*, which is used in real-time systems in which guaranteed interaction response times are required.
  - *Two-tier client-server architecture*, which is used for simple client-server systems, and where the system is centralized for security reasons.
  - *Multi-tier client-server architecture*, which is used when there is a high volume of transactions to be processed by the server.
  - *Distributed component architecture*, which is used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client-server systems.
  - *Peer-to-peer architecture*, which is used when clients exchange locally stored information and the role of the server is to introduce clients to each other

# Master-slave architectures

- Master-slave architectures are commonly used in real-time systems where there may be separate processors associated with data acquisition from the system's environment, data processing and computation and actuator management.
- The 'master' process is usually responsible for computation, coordination and communications and it controls the 'slave' processes.
- 'Slave' processes are dedicated to specific actions, such as the acquisition of data from an array of sensors.

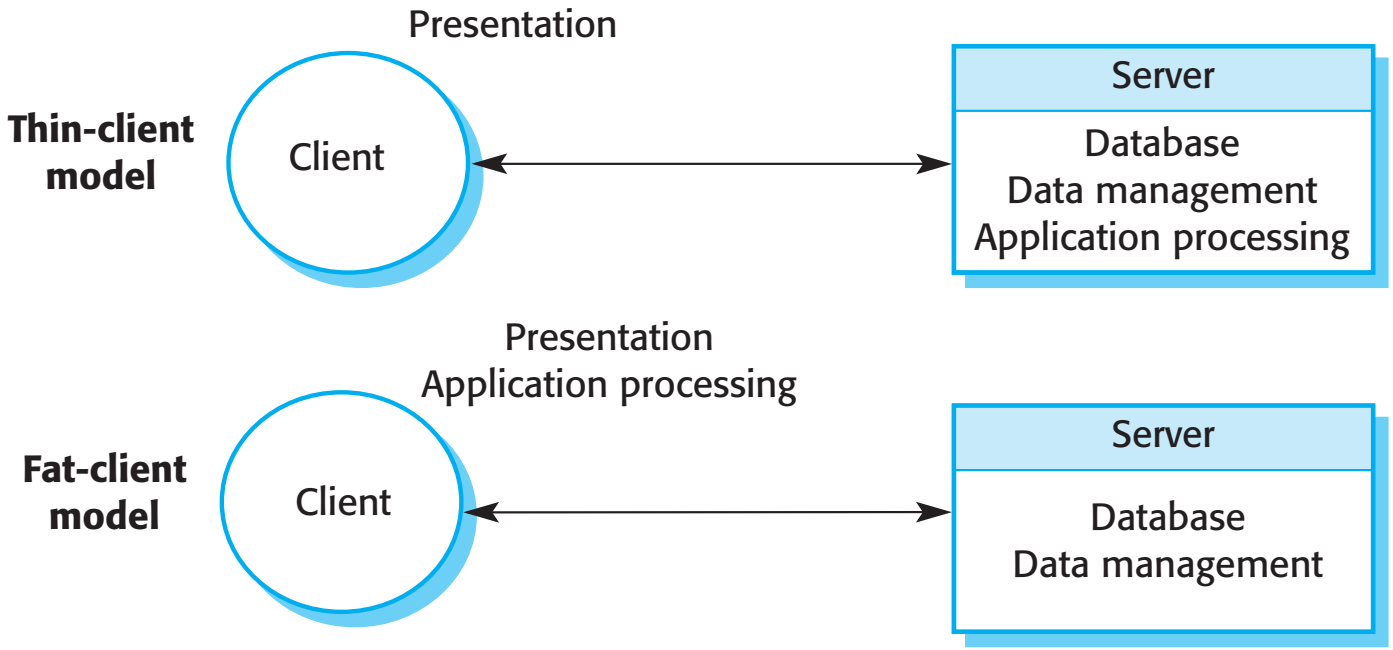
# A traffic management system with a master-slave architecture



# Two-tier client server architectures

- In a two-tier client-server architecture, the system is implemented as a single logical server plus an indefinite number of clients that use that server.
  - Thin-client model, where the presentation layer is implemented on the client and all other layers (data management, application processing and database) are implemented on a server.
  - Fat-client model, where some or all of the application processing is carried out on the client. Data management and database functions are implemented on the server.

# Thin- and fat-client architectural models



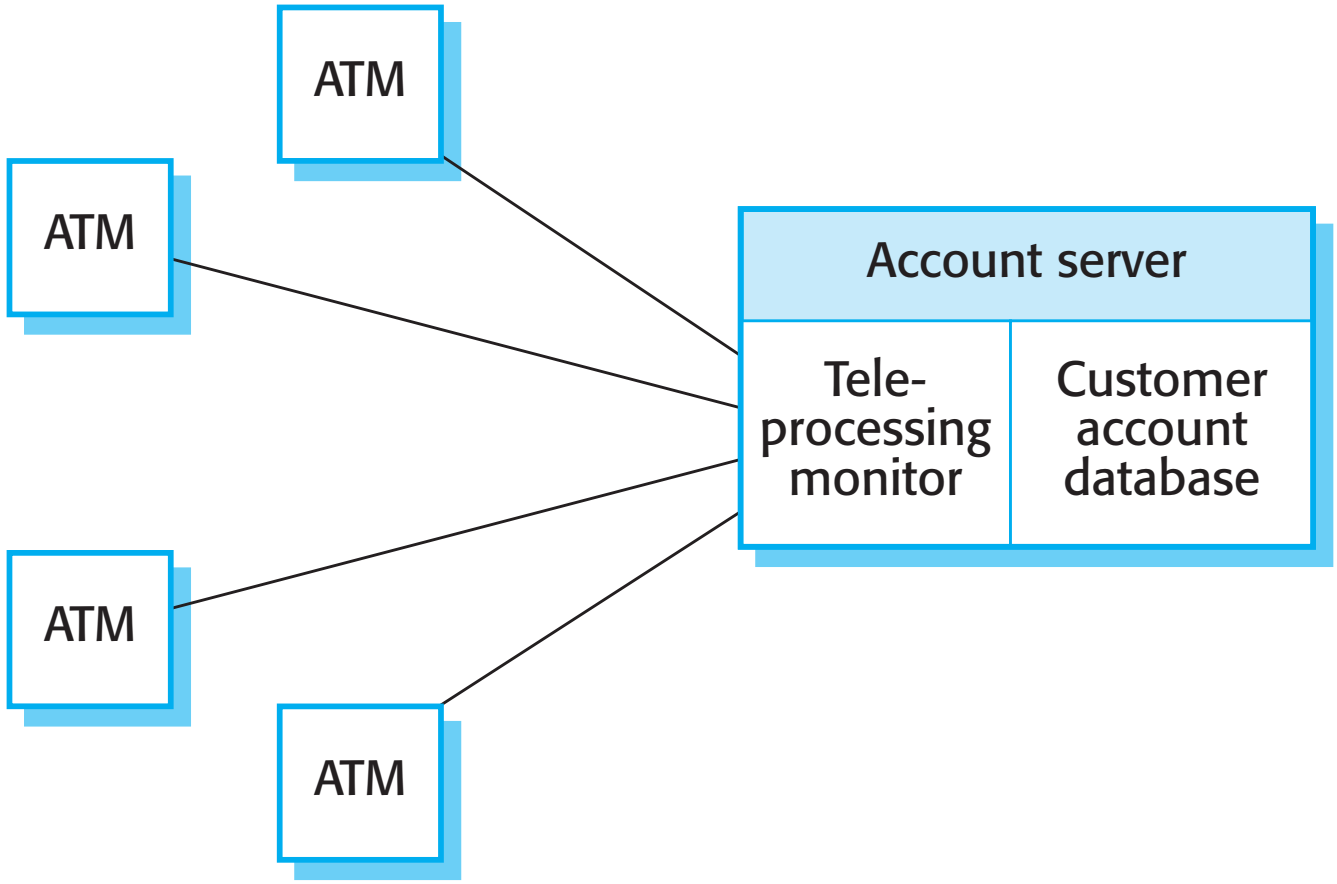
# Thin client model

- Used when legacy systems are migrated to client server architectures.
  - The legacy system acts as a server in its own right with a graphical interface implemented on a client.
- A major disadvantage is that it places a heavy processing load on both the server and the network.

# Fat client model

- More processing is delegated to the client as the application processing is locally executed.
- Most suitable for new C/S systems where the capabilities of the client system are known in advance.
- More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.

# A fat-client architecture for an ATM system



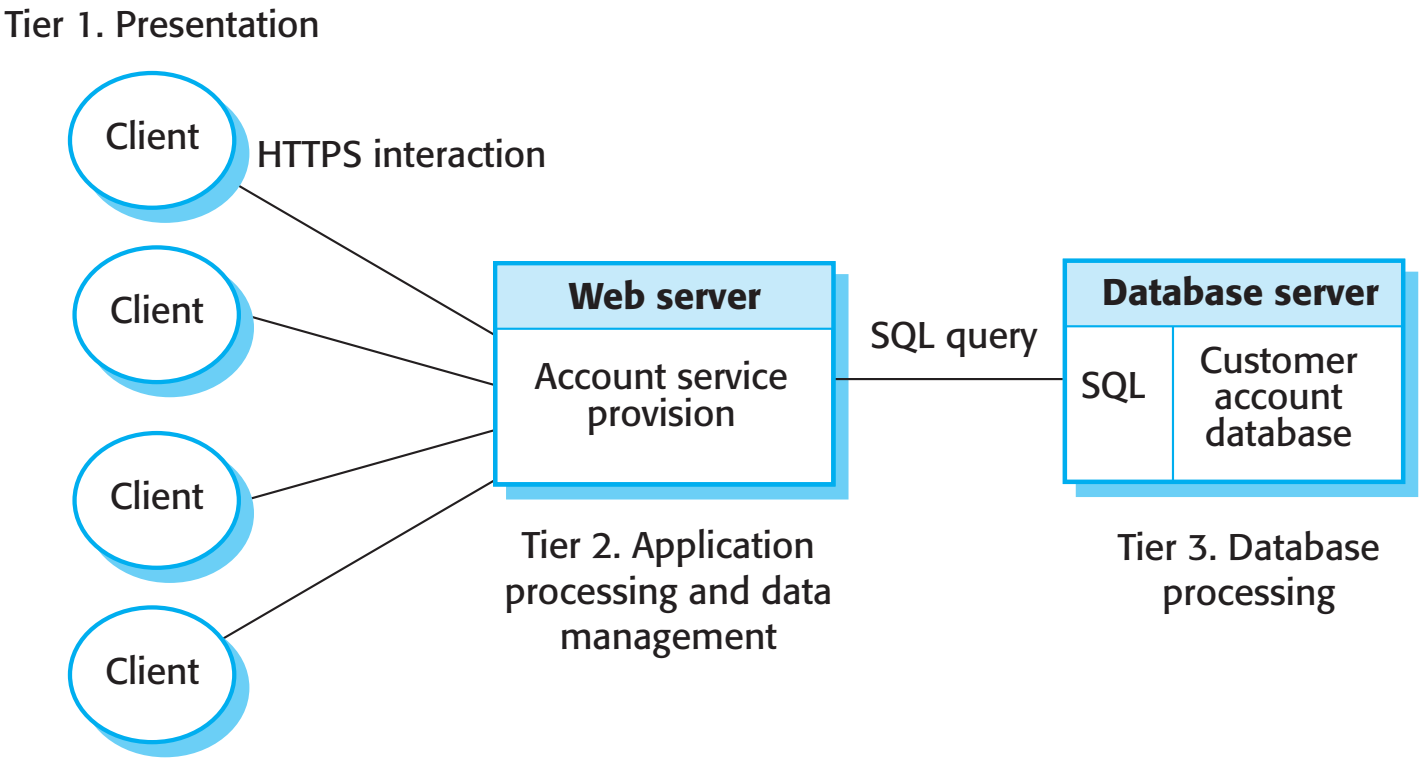
# Thin and fat clients

- Distinction between thin and fat client architectures has become blurred
- Javascript allows local processing in a browser so ‘fat-client’ functionality available without software installation
- Mobile apps carry out some local processing to minimize demands on network
- Auto-update of apps reduces management problems
- There are now very few thin-client applications with all processing carried out on remote server.

# Multi-tier client-server architectures

- In a ‘multi-tier client–server’ architecture, the different layers of the system, namely presentation, data management, application processing, and database, are separate processes that may execute on different processors.
- This avoids problems with scalability and performance if a thin-client two-tier model is chosen, or problems of system management if a fat-client model is used.

# Three-tier architecture for an Internet banking system



# Use of client–server architectural patterns

| Architecture | Applications |
| --- | --- |
| Two-tier client–server architecture with thin clients | Legacy system applications that are used when separating application processing and data management is impractical. Clients may access these as services, as discussed in Section 18.4.<br>Computationally intensive applications such as compilers with little or no data management.<br>Data-intensive applications (browsing and querying) with nonintensive application processing. Browsing the Web is the most common example of a situation where this architecture is used. |

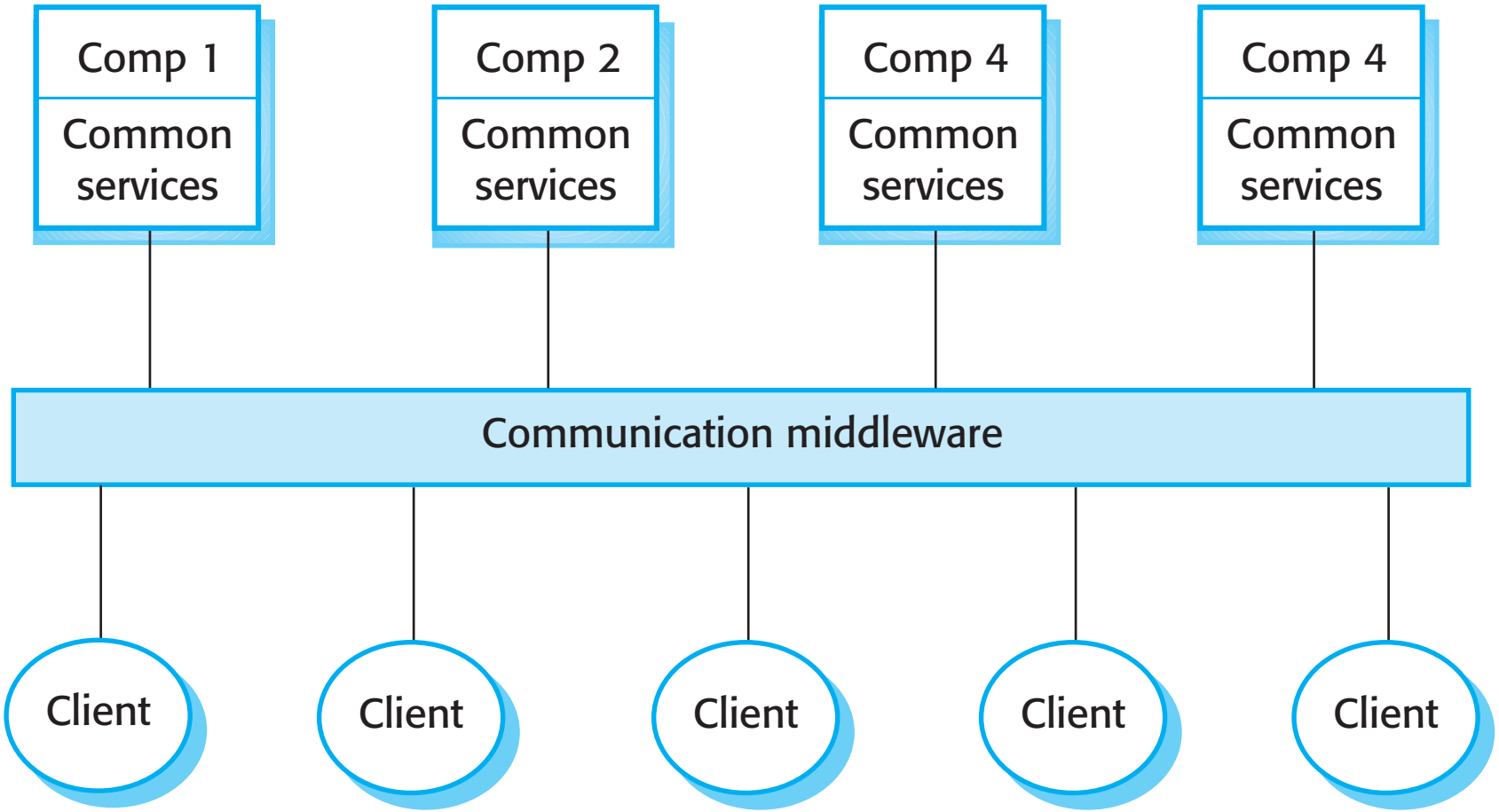
# Use of client–server architectural patterns

| Architecture | Applications |
| --- | --- |
| Two-tier client-server architecture with fat clients | Applications where application processing is provided by off-the-shelf software (e.g., Microsoft Excel) on the client.<br>Applications where computationally intensive processing of data (e.g., data visualization) is required.<br>Mobile applications where internet connectivity cannot be guaranteed. Some local processing using cached information from the database is therefore possible. |
| Multi-tier client–server architecture | Large-scale applications with hundreds or thousands of clients.<br>Applications where both the data and the application are volatile.<br>Applications where data from multiple sources are integrated. |

# Distributed component architectures

- There is no distinction in a distributed component architecture between clients and servers.
- Each distributable entity is a component that provides services to other components and receives services from other components.
- Component communication is through a middleware system.

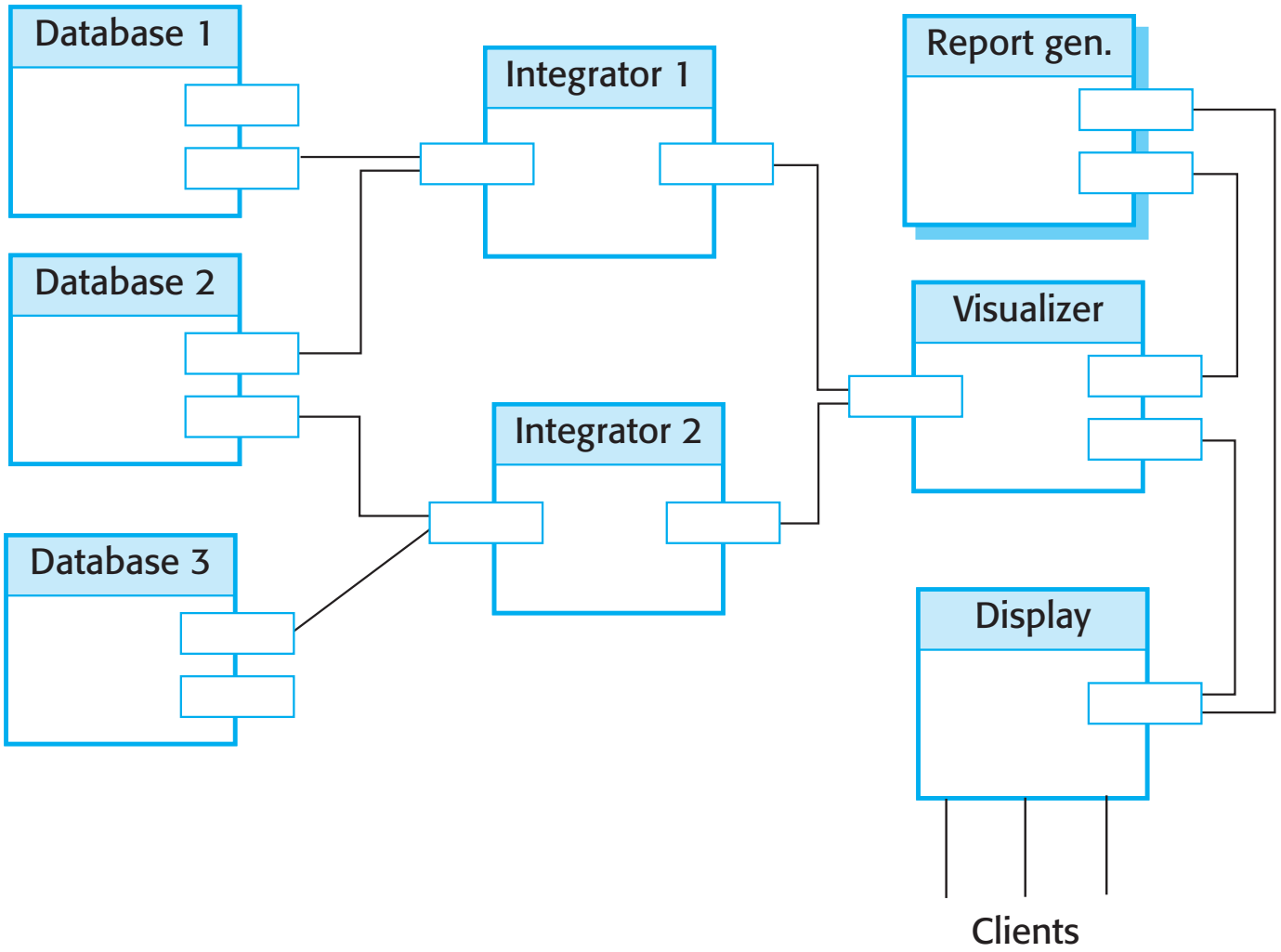
# A distributed component architecture



# Benefits of distributed component architecture

- It allows the system designer to delay decisions on where and how services should be provided.
- It is a very open system architecture that allows new resources to be added as required.
- The system is flexible and scalable.
- It is possible to reconfigure the system dynamically with objects migrating across the network as required.

# A distributed component architecture for a data mining system



# Disadvantages of distributed component architecture

- Distributed component architectures suffer from two major disadvantages:
  - They are more complex to design than client–server systems. Distributed component architectures are difficult for people to visualize and understand.
  - Standardized middleware for distributed component systems has never been accepted by the community. Different vendors, such as Microsoft and Sun, have developed different, incompatible middleware.
- As a result of these problems, service-oriented architectures are replacing distributed component architectures in many situations.

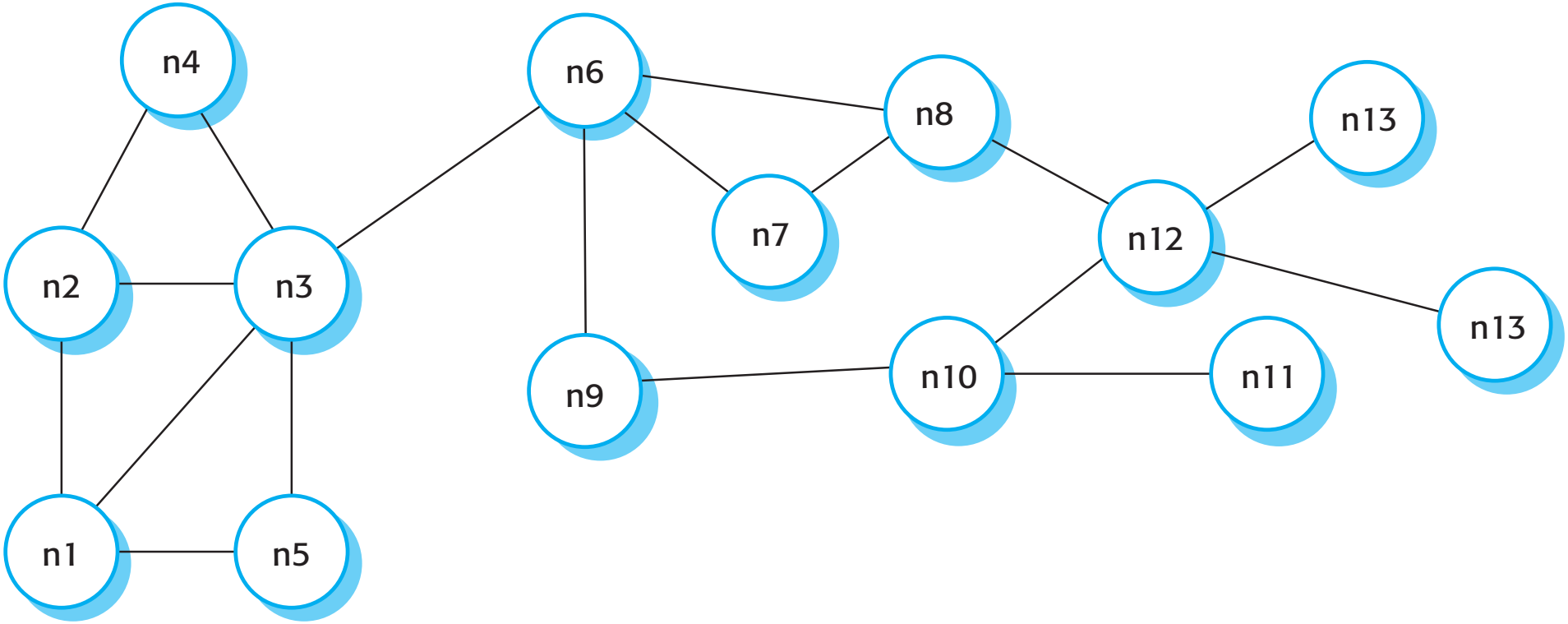
# Peer-to-peer architectures

- Peer to peer (p2p) systems are decentralised systems where computations may be carried out by any node in the network.
- The overall system is designed to take advantage of the computational power and storage of a large number of networked computers.
- Most p2p systems have been personal systems but there is increasing business use of this technology.

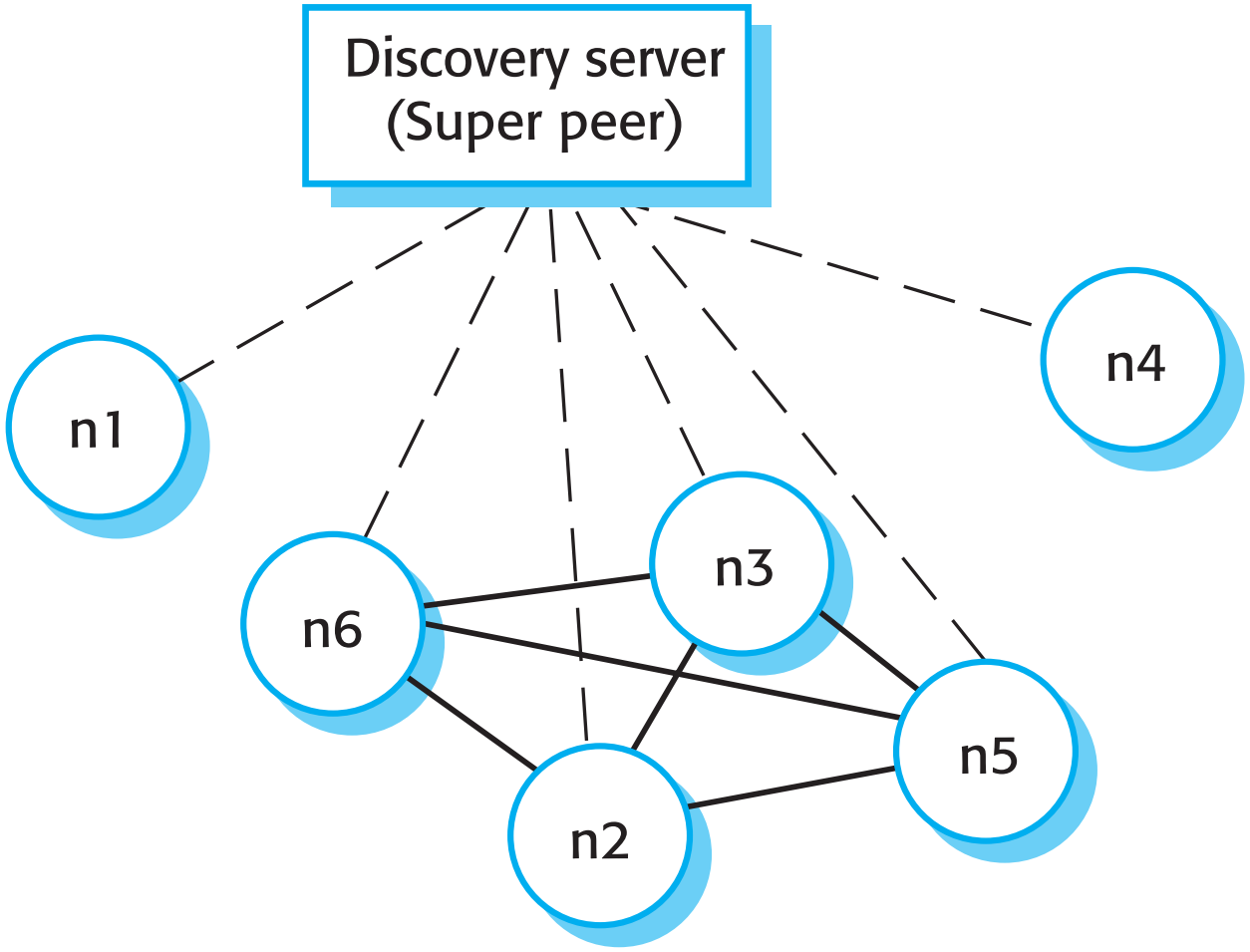
# Peer-to-peer systems

- File sharing systems based on the BitTorrent protocol
- Messaging systems such as Jabber
- Payments systems – Bitcoin
- Databases – Freenet is a decentralized database
- Phone systems – Viber
- Computation systems - SETI@home

# A decentralized p2p architecture



# A semicentralized p2p architecture



# Main Reference

- M. van Steen and A. S. Tanenbaum, Distributed systems, Third edition (Version 3.01 (2017)). London: Pearson Education, 2017. (in the following additional slides)
- G. F. Coulouris, Ed., *Distributed systems: concepts and design*, 5th ed. Boston: Addison-Wesley, 2012.

# Definition of a Distributed System (1)

- A distributed system is:

- A collection of independent computers that appears to its users as a single coherent system.

# Definition of a Distributed System (2)

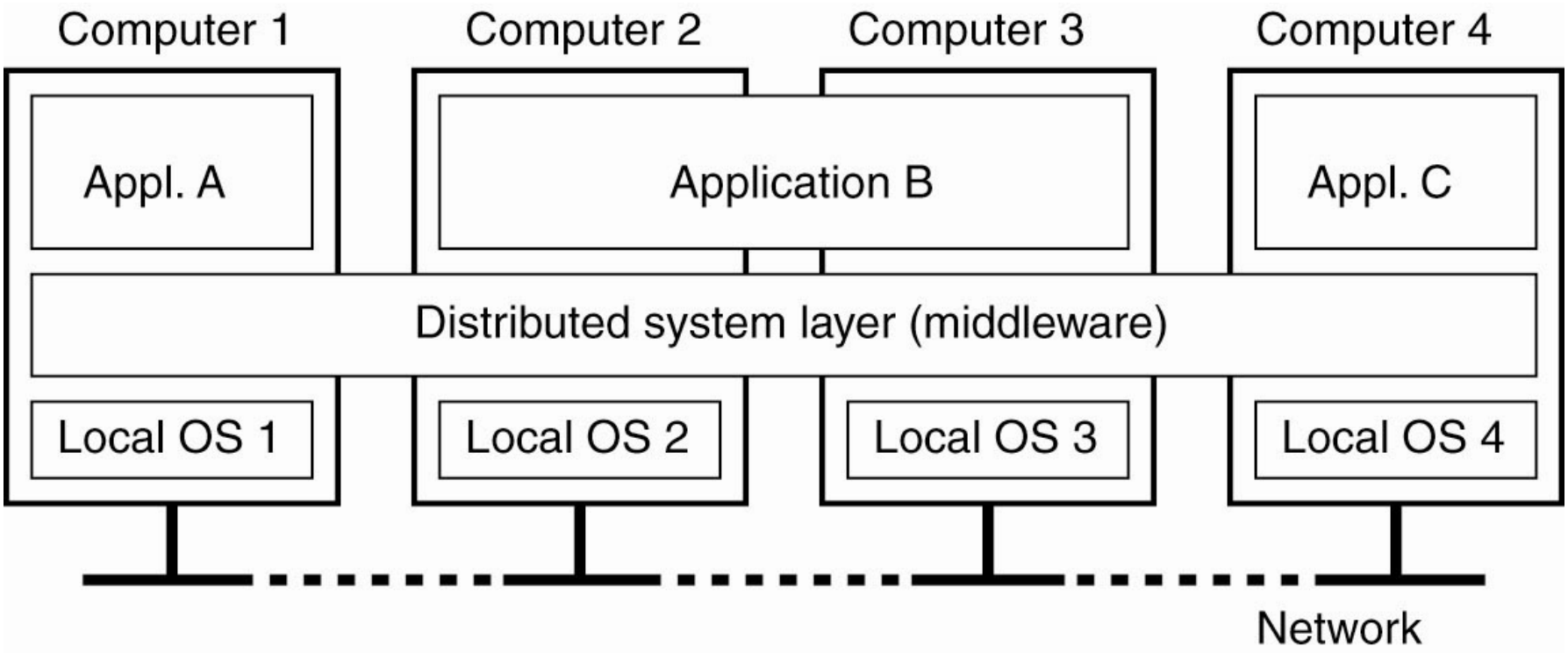



Figure 1-1. A distributed system organized as middleware. The middleware layer extends over multiple machines, and offers each application the same interface.

# Transparency in a Distributed System

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

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

# Scalability Problems

| Concept | Example |
|---|---|
| Centralized services | A single server for all users |
| Centralized data | A single on-line telephone book |
| Centralized algorithms | Doing routing based on complete information |

Figure 1-3. Examples of scalability limitations.

## Scalability Problems

✧ Characteristics of decentralized algorithms:

- No machine has complete information about the system state.
- Machines make decisions based only on local information.
- Failure of one machine does not ruin the algorithm.
- There is no implicit assumption that a global clock exists.

# Scaling Techniques (1)

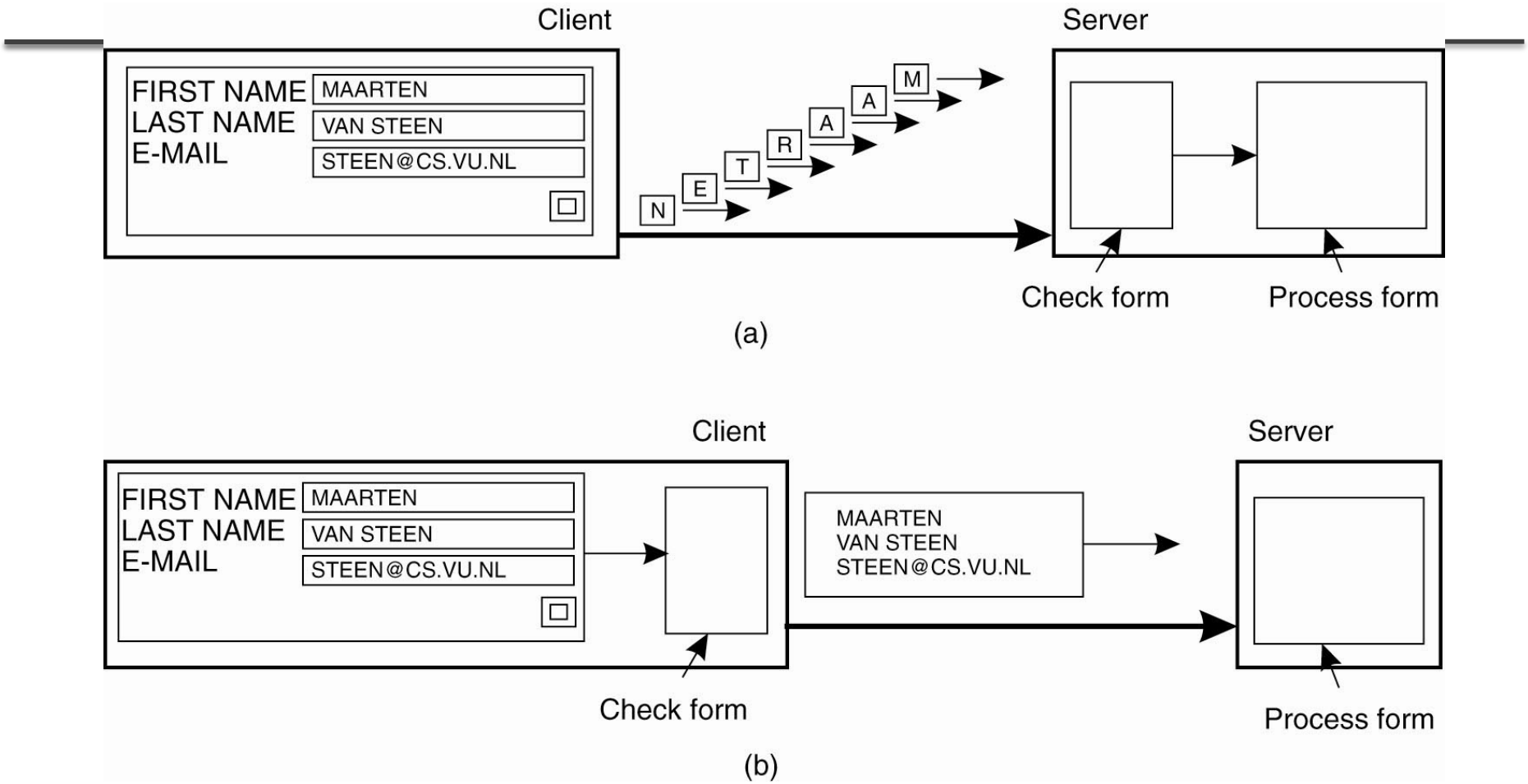



Figure 1-4. The difference between letting (a) a server or (b) a client check forms as they are being filled.

# Scaling Techniques (2)

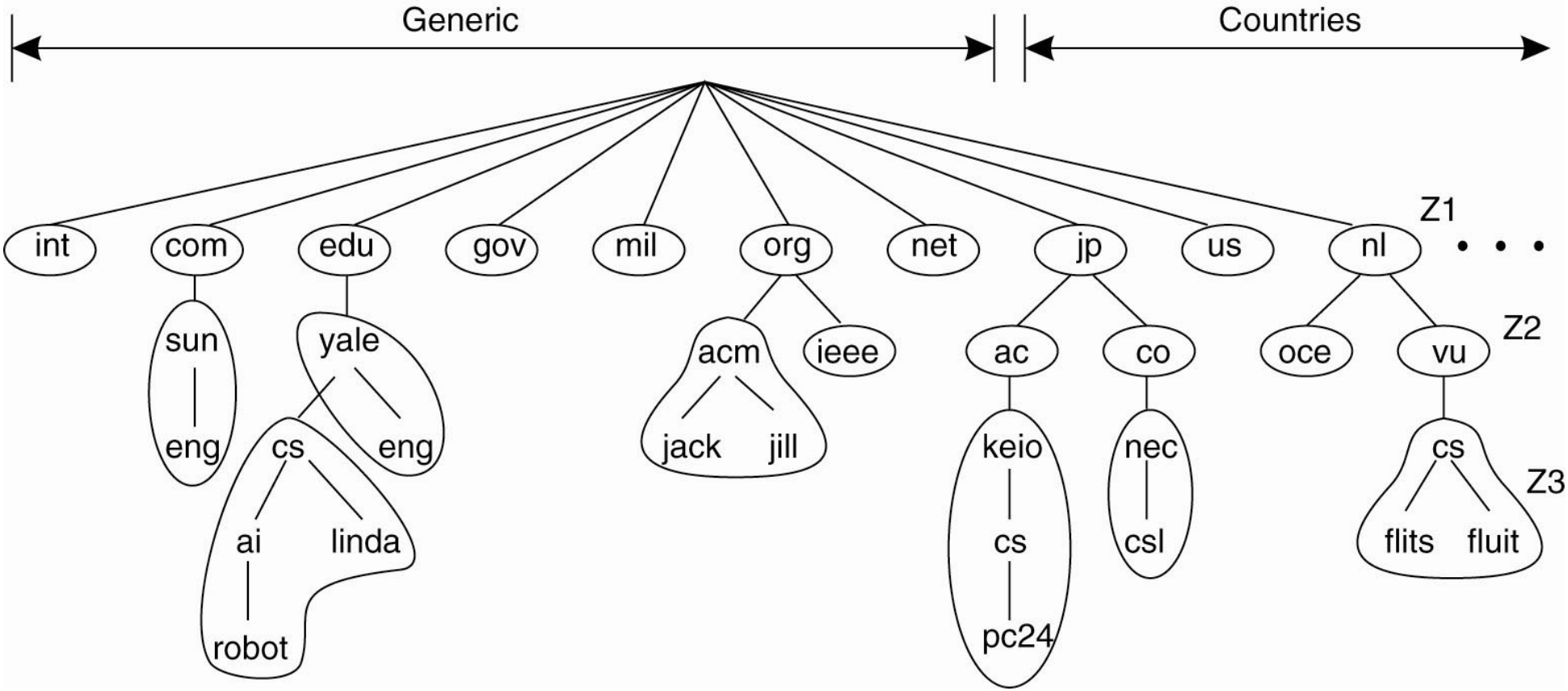


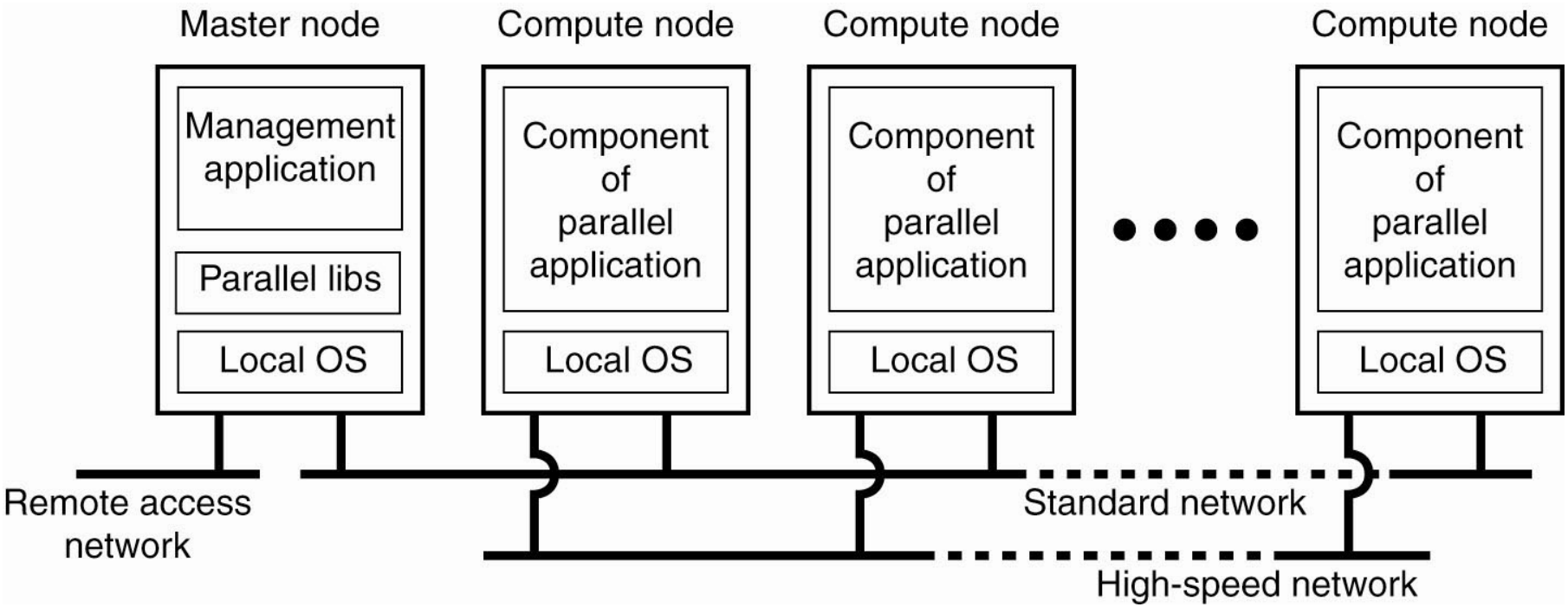

Figure 1-5. An example of dividing the DNS name space into zones.

# Pitfalls when Developing Distributed Systems

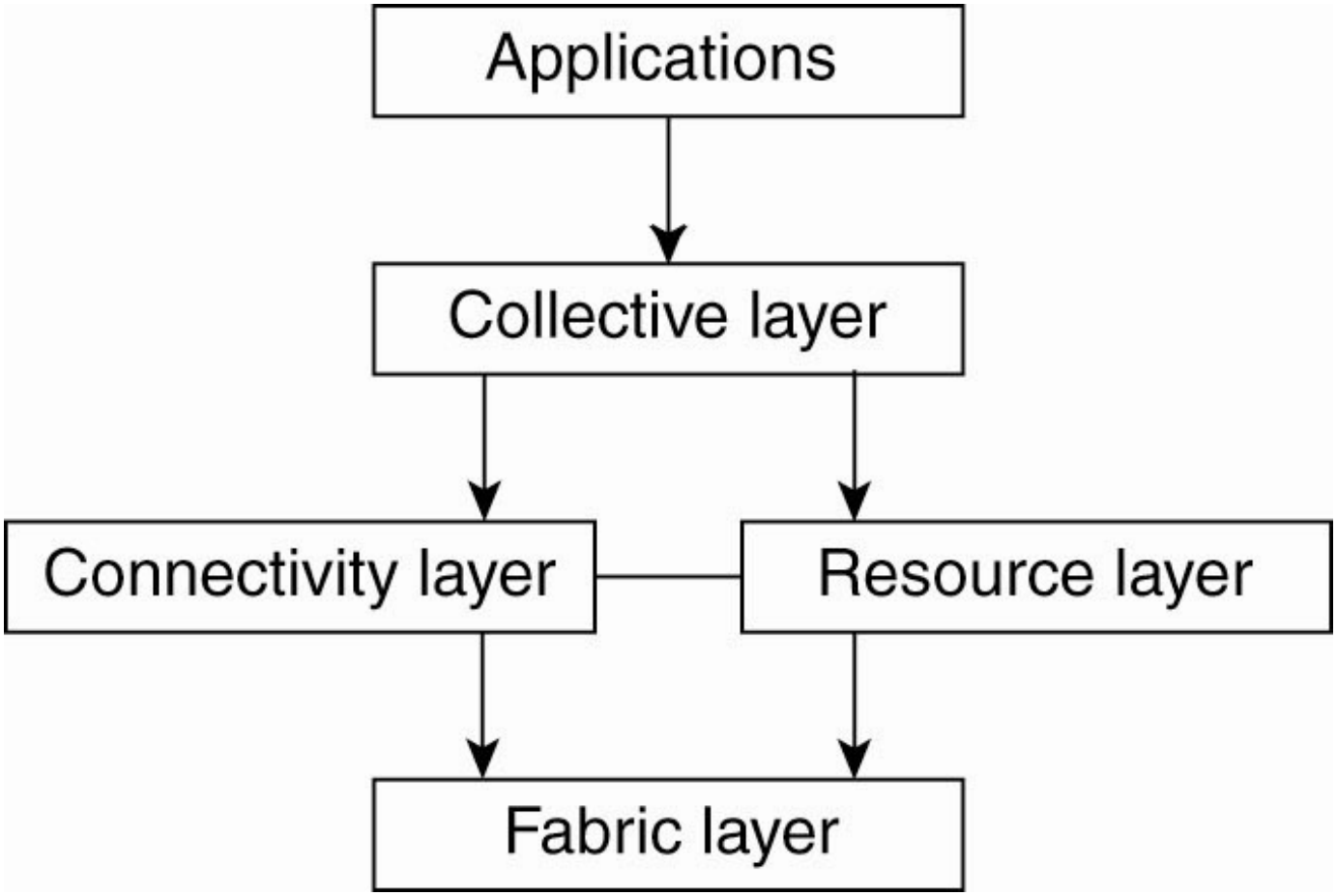
✧ False assumptions made by first time developer:

- The network is reliable.
- The network is secure.
- The network is homogeneous.
- The topology does not change.
- Latency is zero.
- Bandwidth is infinite.
- Transport cost is zero.
- There is one administrator.

# Cluster Computing Systems



# Grid Computing Systems




✧ Figure 1-7. A layered architecture for grid computing systems.

# Transaction Processing Systems (1)

✧ Figure 1-8. Example primitives for transactions.

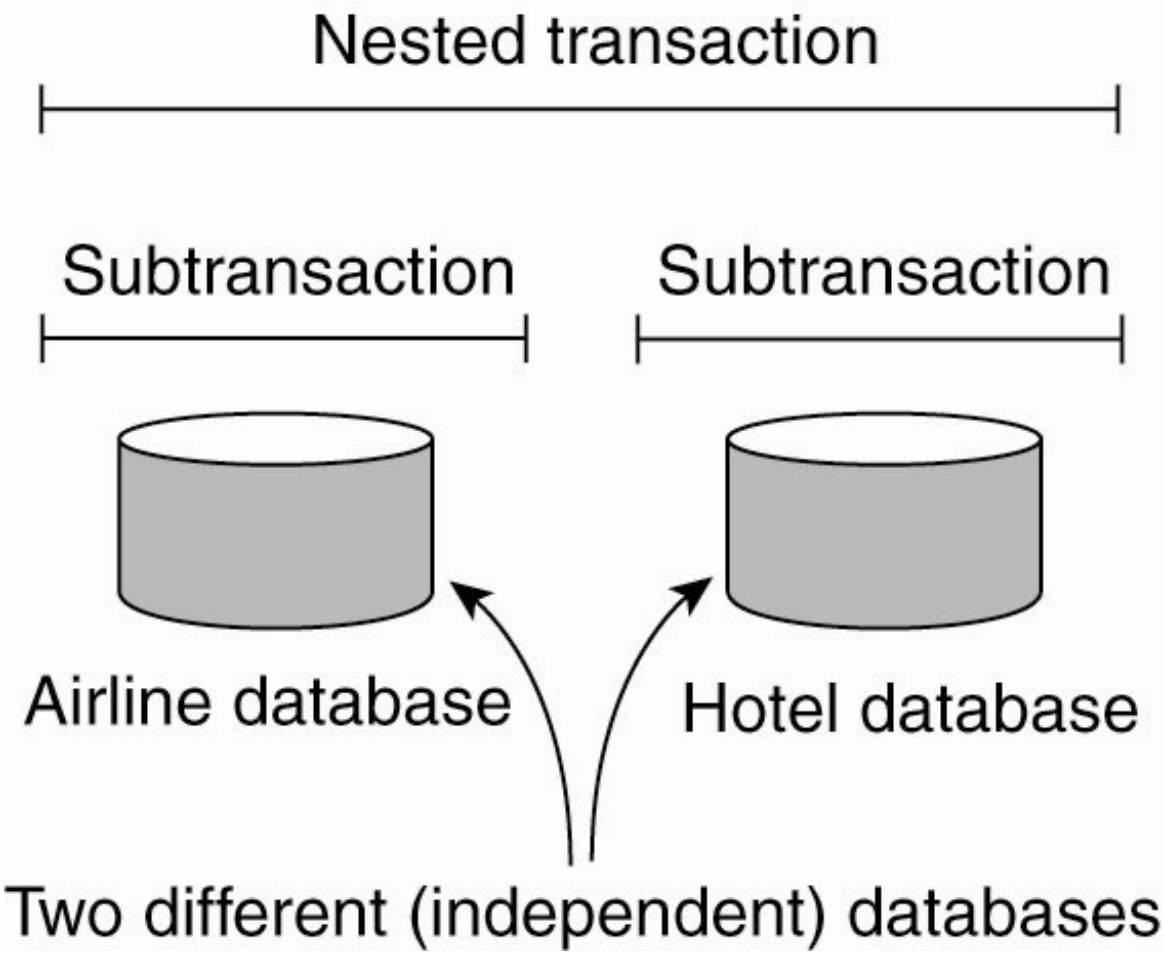
| Primitive | Description |
|---|---|
| BEGIN_TRANSACTION | Mark the start of a transaction |
| END_TRANSACTION | Terminate the transaction and try to commit |
| ABORT_TRANSACTION | Kill the transaction and restore the old values |
| READ | Read data from a file, a table, or otherwise |
| WRITE | Write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

# Transaction Processing Systems (2)

✧ Characteristic properties of transactions:

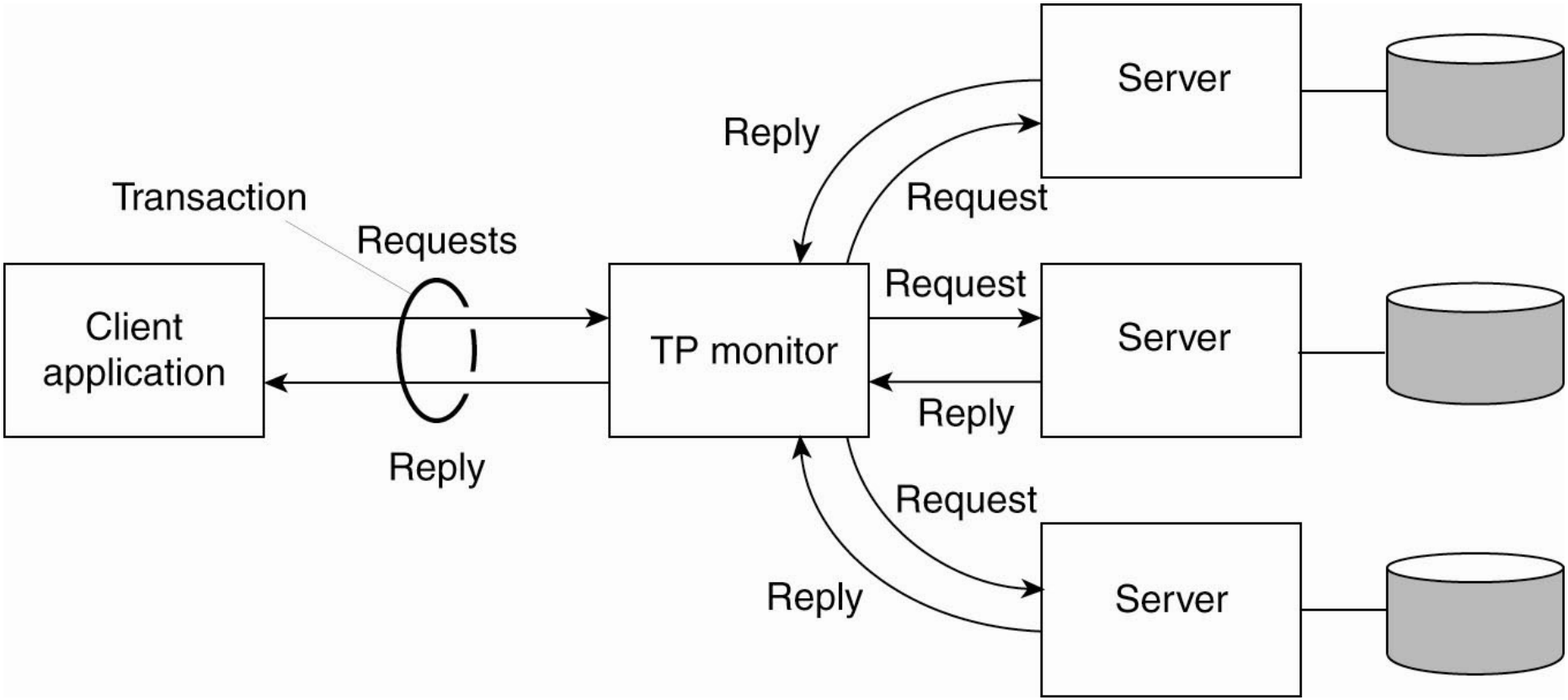
- Atomic: To the outside world, the transaction happens indivisibly.
- Consistent: The transaction does not violate system invariants.
- Isolated: Concurrent transactions do not interfere with each other.
- Durable: Once a transaction commits, the changes are permanent.

# Transaction Processing Systems (3)

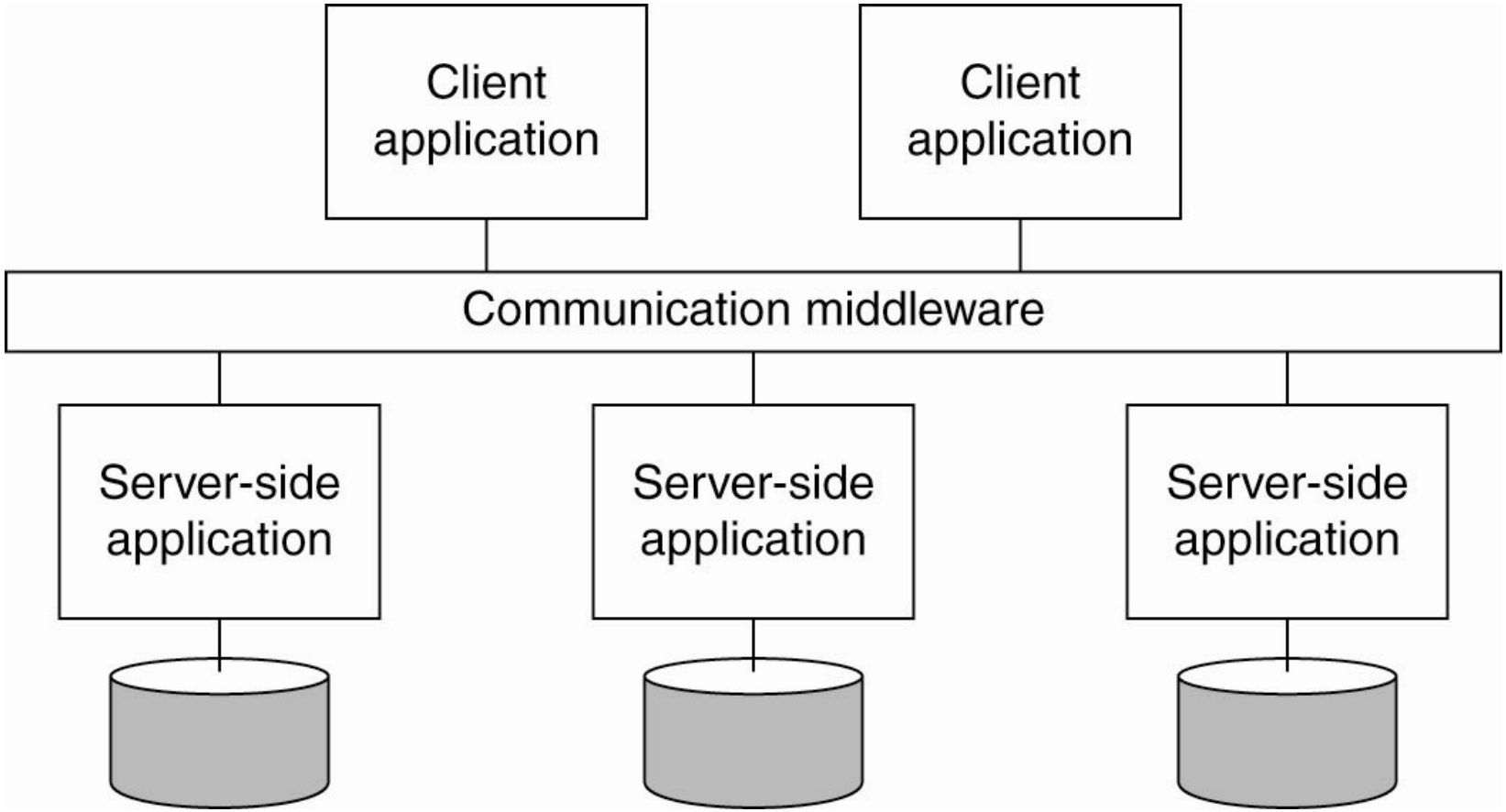



✧ Figure 1-9. A nested transaction.

# Transaction Processing Systems (4)



# Enterprise Application Integration




✧ Figure 1-11. Middleware as a communication facilitator in enterprise application integration.

## Distributed Pervasive Systems

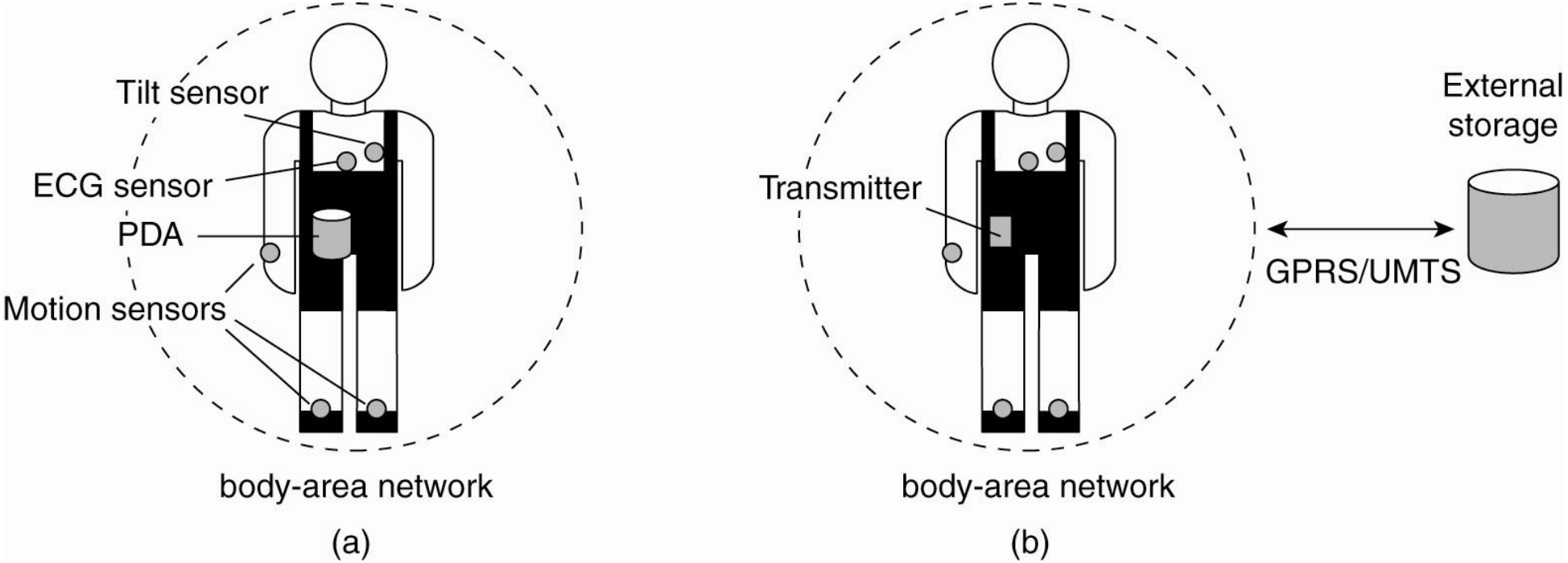
✧Requirements for pervasive systems

- Embrace contextual changes.
- Encourage ad hoc composition.
- Recognize sharing as the default.

## Electronic Health Care Systems (1)

- Questions to be addressed for health care systems:
  - Where and how should monitored data be stored?
  - How can we prevent loss of crucial data?
  - What infrastructure is needed to generate and propagate alerts?
  - How can physicians provide online feedback?
  - How can extreme robustness of the monitoring system be realized?
  - What are the security issues and how can the proper policies be enforced?

# Electronic Health Care Systems (2)



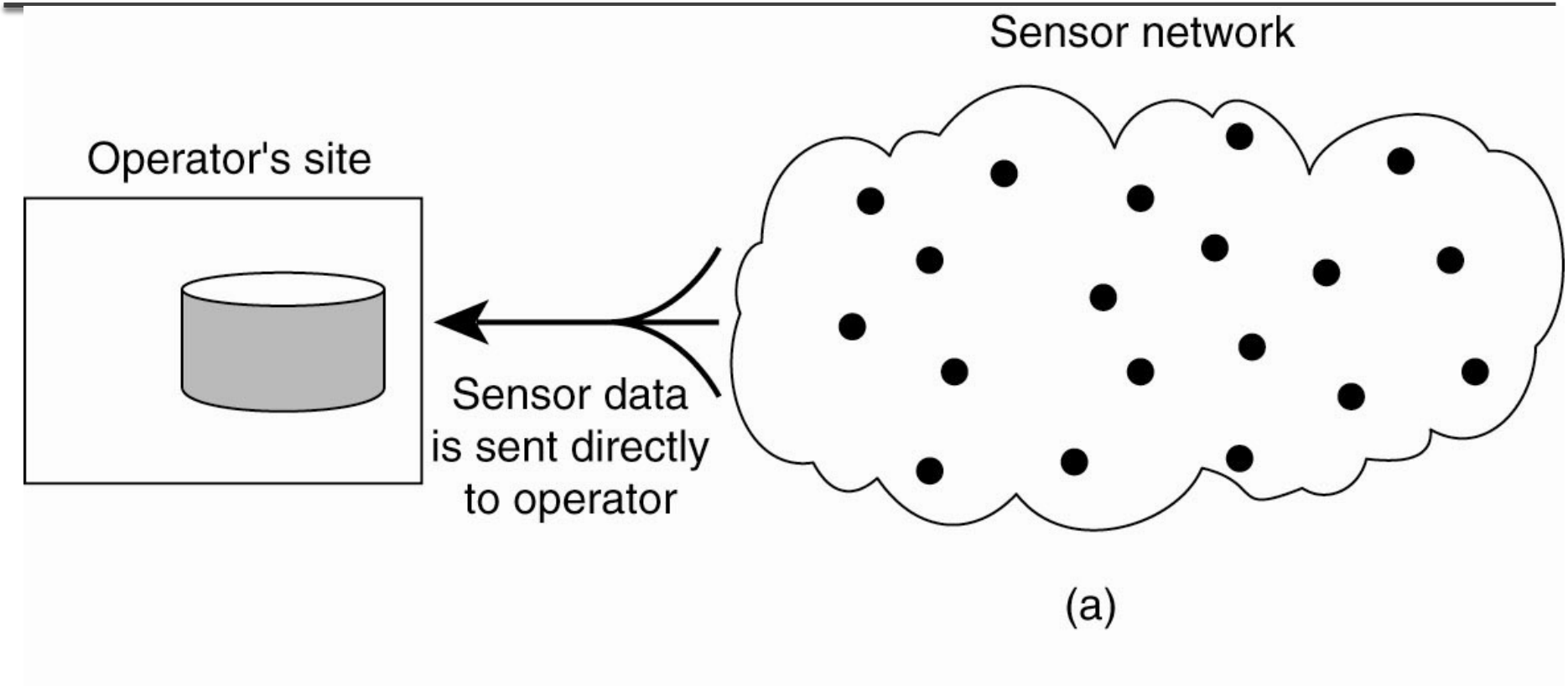

- Figure 1-12. Monitoring a person in a pervasive electronic health care system, using (a) a local hub or
- (b) a continuous wireless connection.

## Sensor Networks (1)

✧Questions concerning sensor networks:

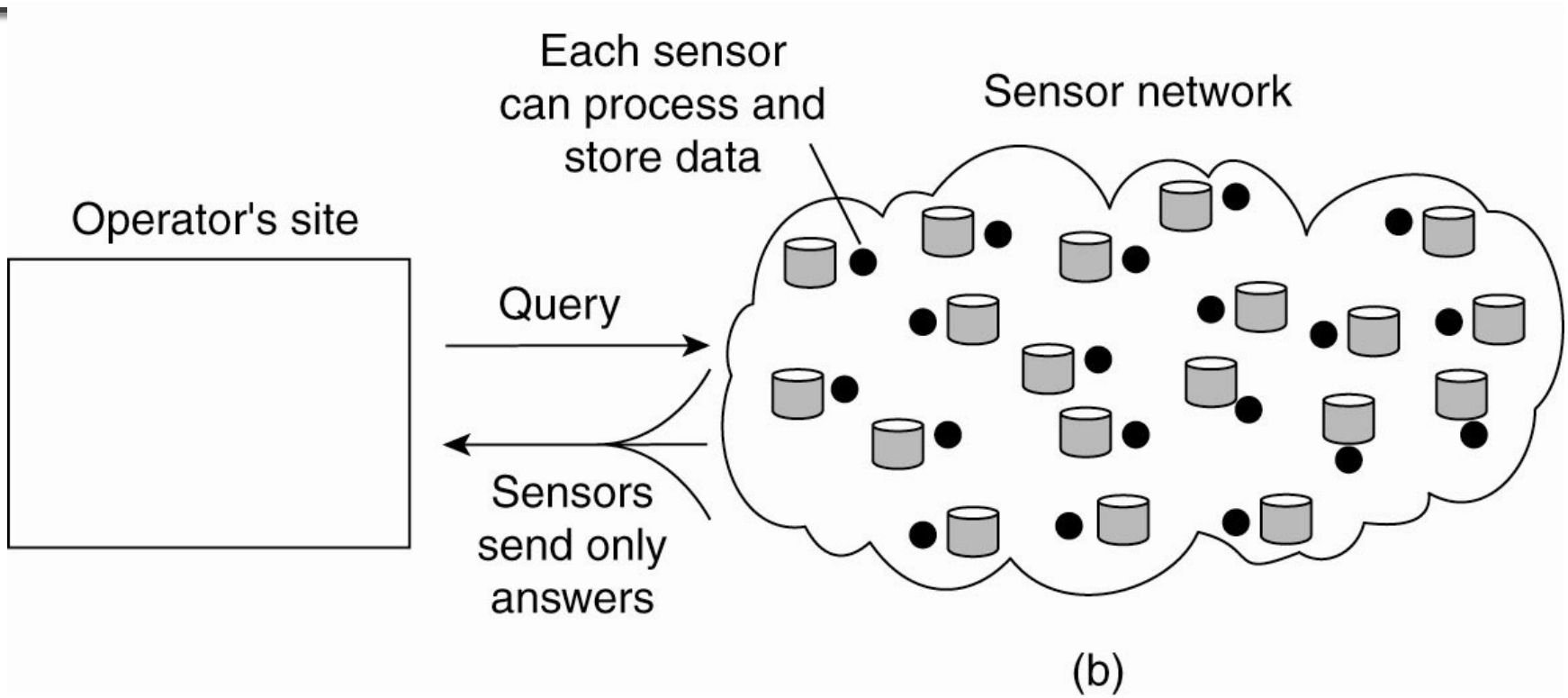
- How do we (dynamically) set up an efficient tree in a sensor network?
- How does aggregation of results take place? Can it be controlled?
- What happens when network links fail?

# Sensor Networks (2)




✧ Figure 1-13. Organizing a sensor network database, while storing and processing data (a) only at the operator's site or …

# Sensor Networks (3)

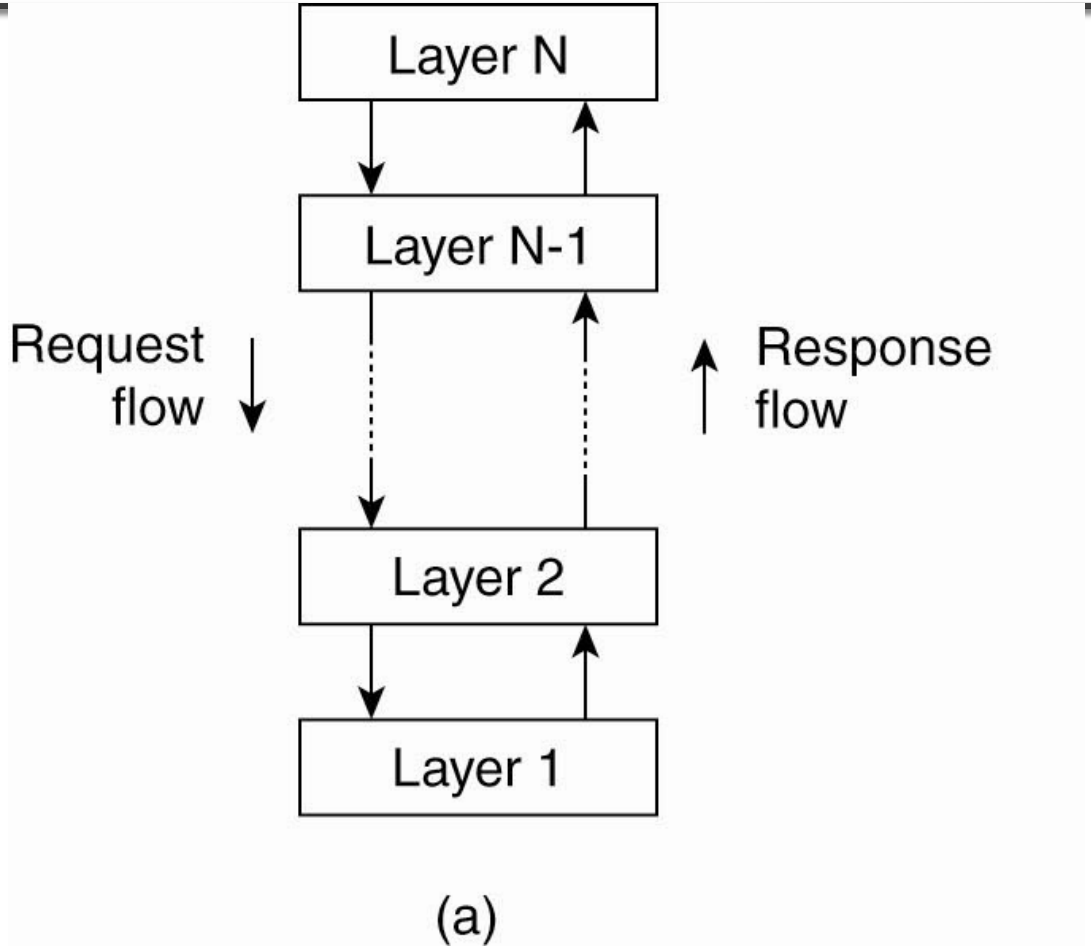



✧ Figure 1-13. Organizing a sensor network database, while storing and processing data … or (b) only at the sensors.

# Architectural Styles (1)

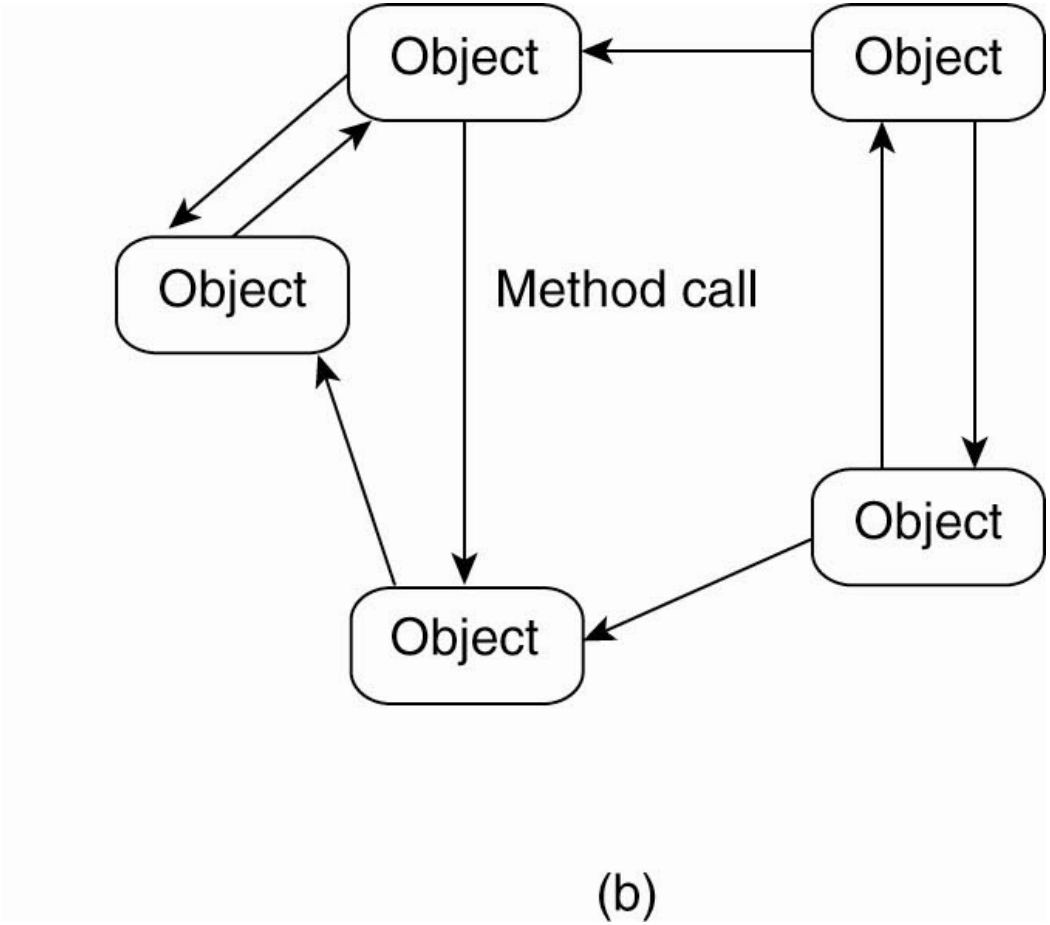
- ✧ Important styles of architecture for distributed systems
- Layered architectures
- Object-based architectures
- Data-centered architectures
- Event-based architectures

# Architectural Styles (2)




✧ Figure 2-1. The (a) layered architectural style and …

# Architectural Styles (3)

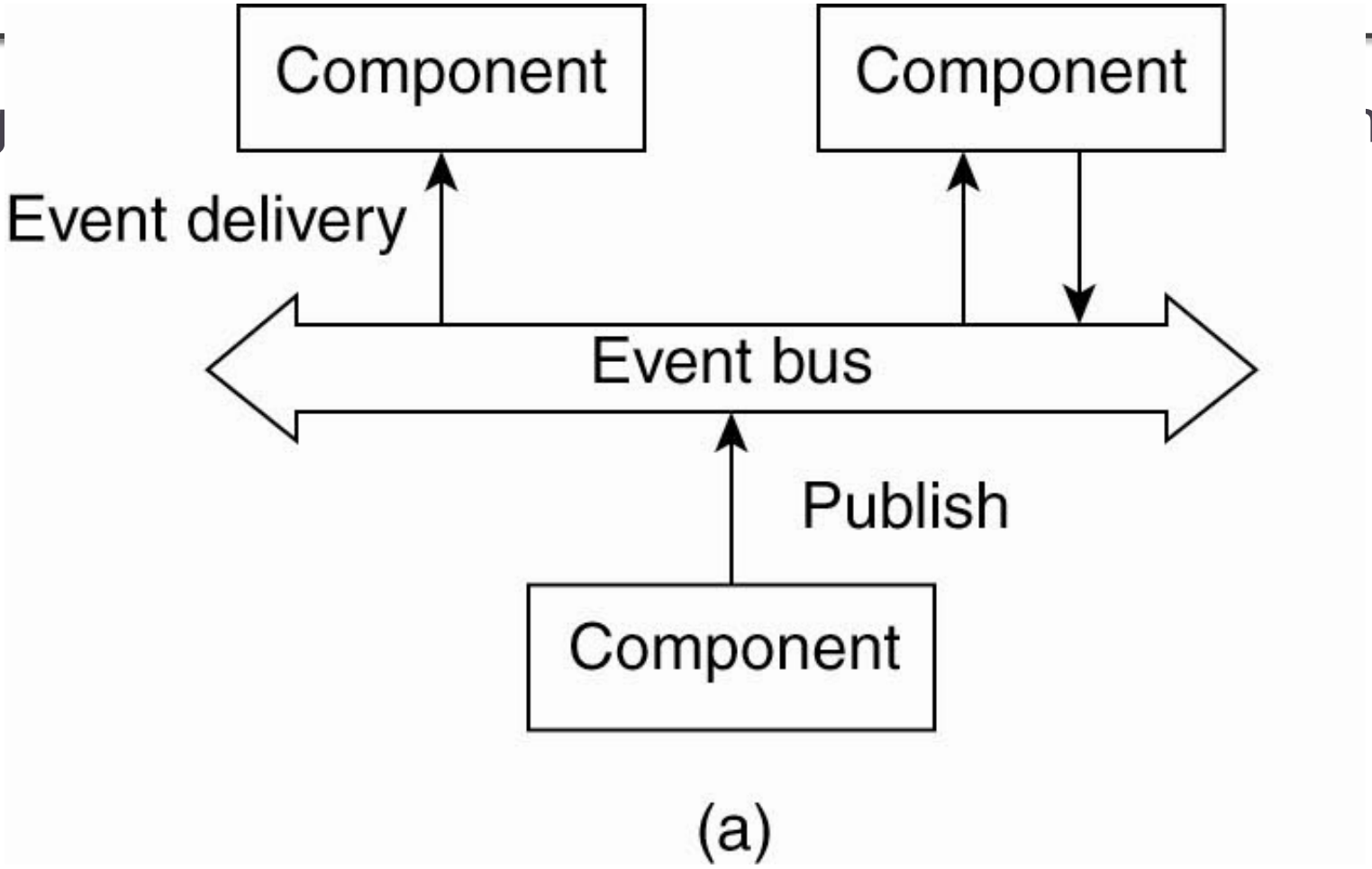



✧ Figure 2-1. (b) The object-based architectural style.

# Architectural Styles (4)

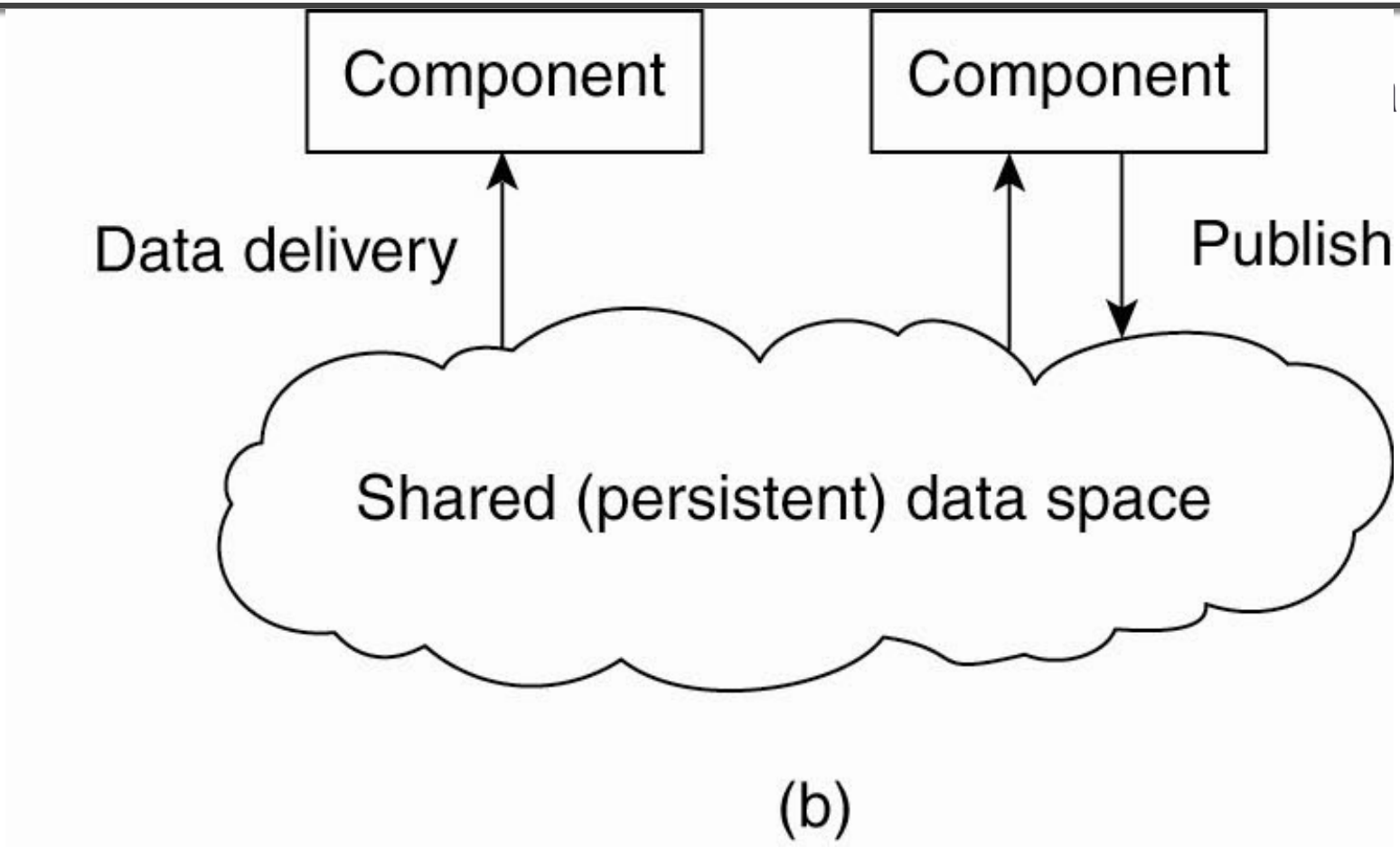
✧ Fig … nd

… Event delivery



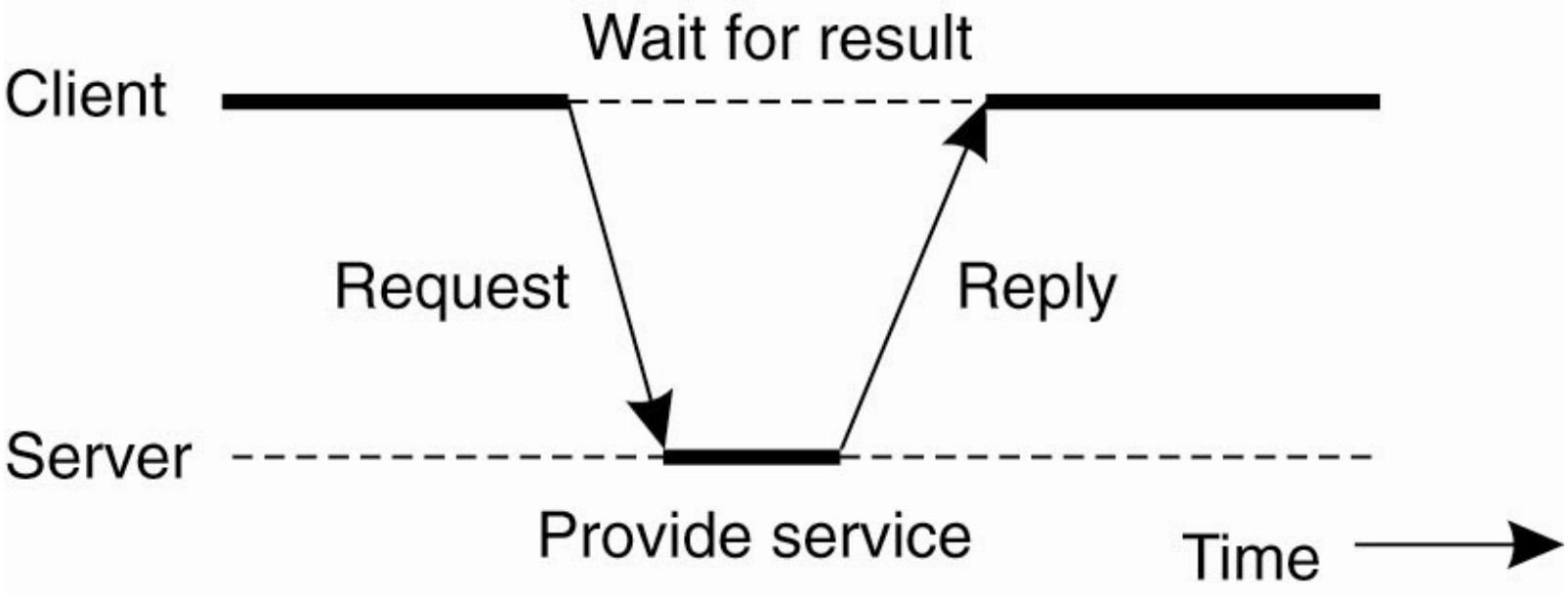
# Architectural Styles (5)

- l style.



# Centralized Architectures

✧ Figure 2-3. General interaction between a client and a server.



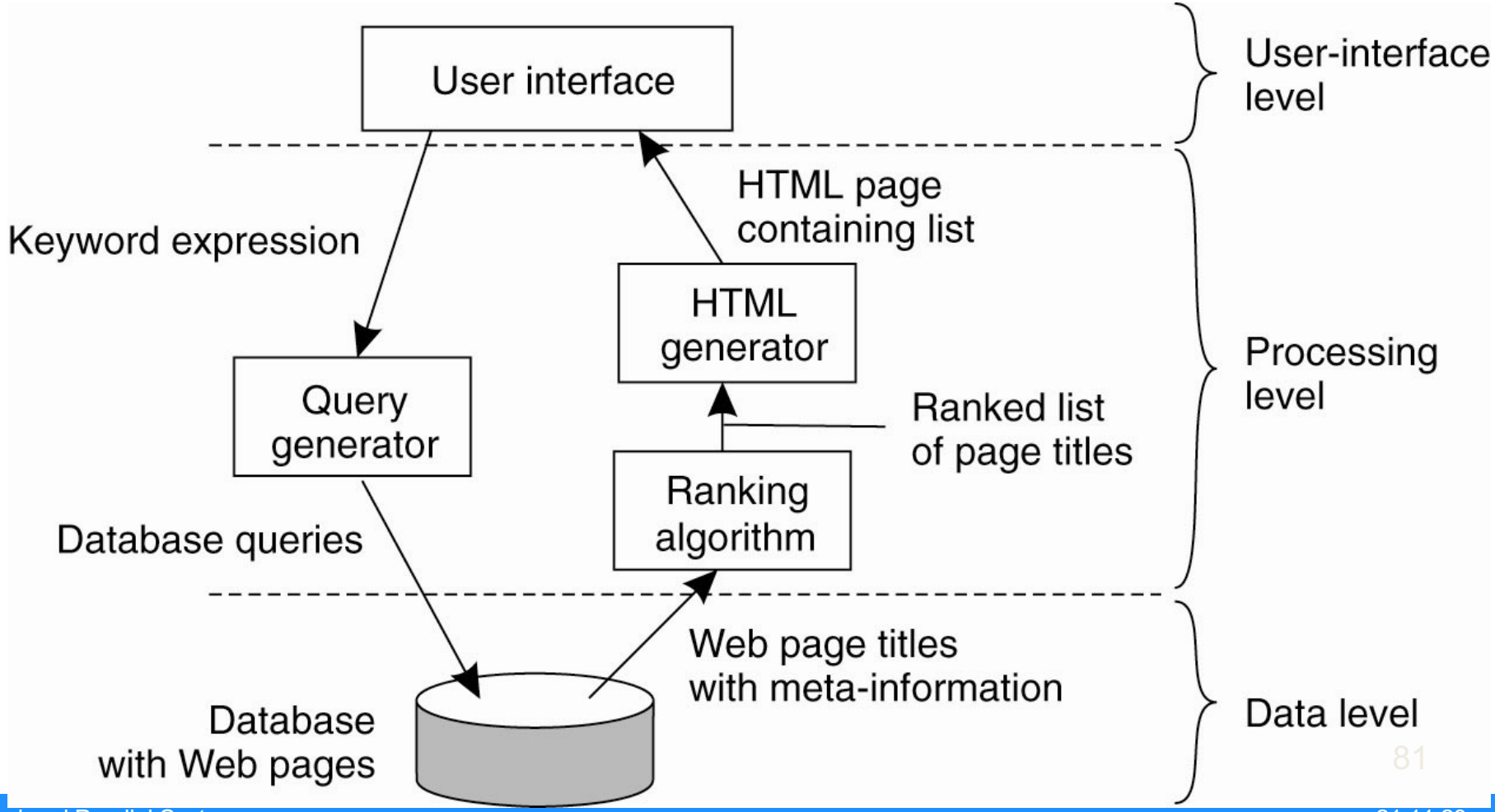
# Application Layering (1)

✧Recall previously mentioned layers of architectural style

- The user-interface level
- The processing level
- The data level

# Application Layering (2)

✧ Figure 2-4. The simplified organization of an Internet search engine into three different layers.

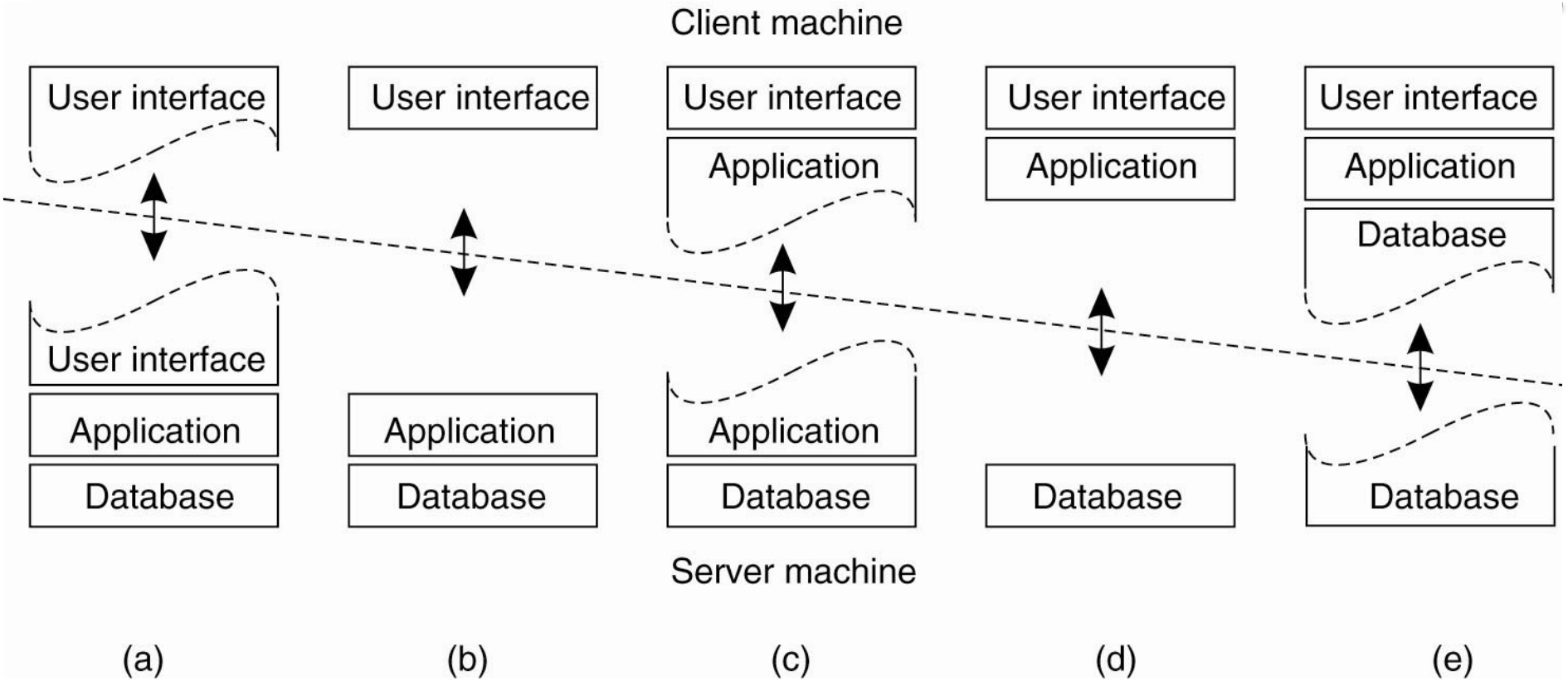


# Multitiered Architectures (1)

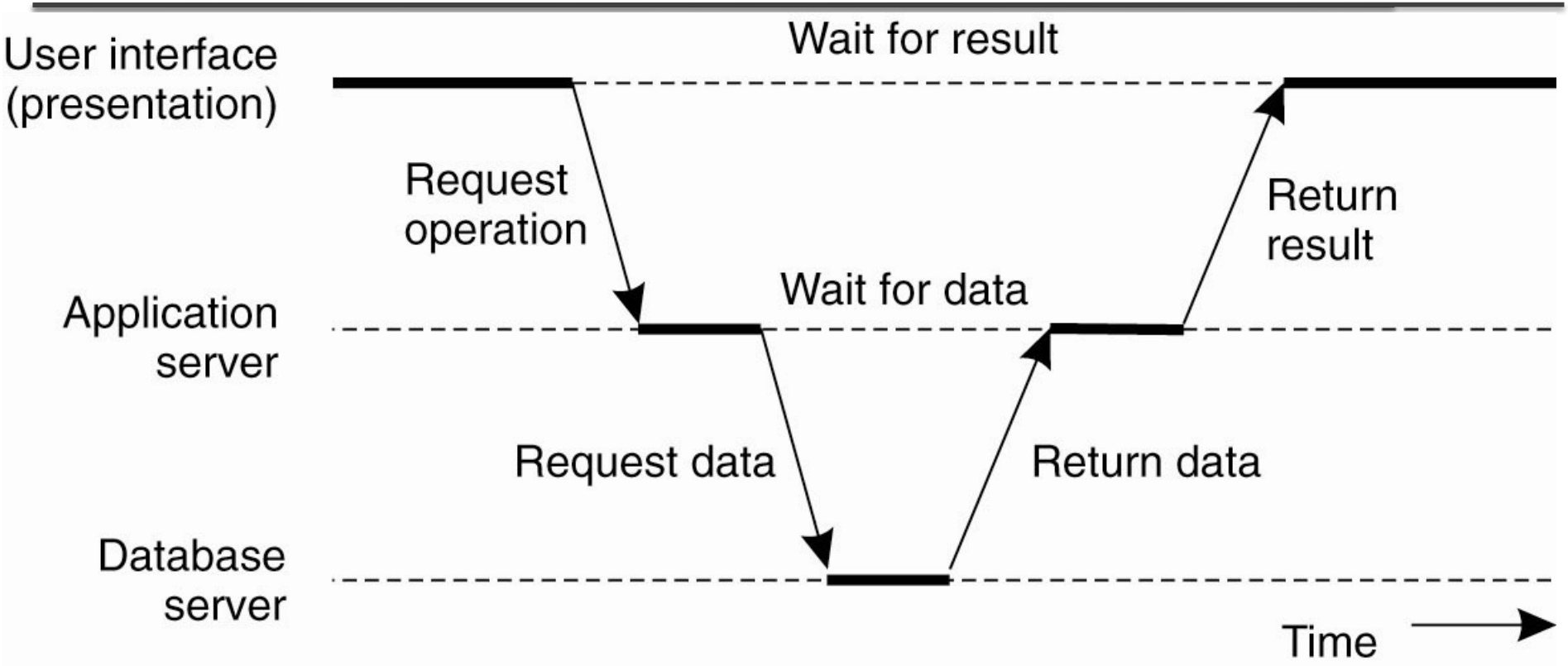
✧ The simplest organization is to have only two types of machines:

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

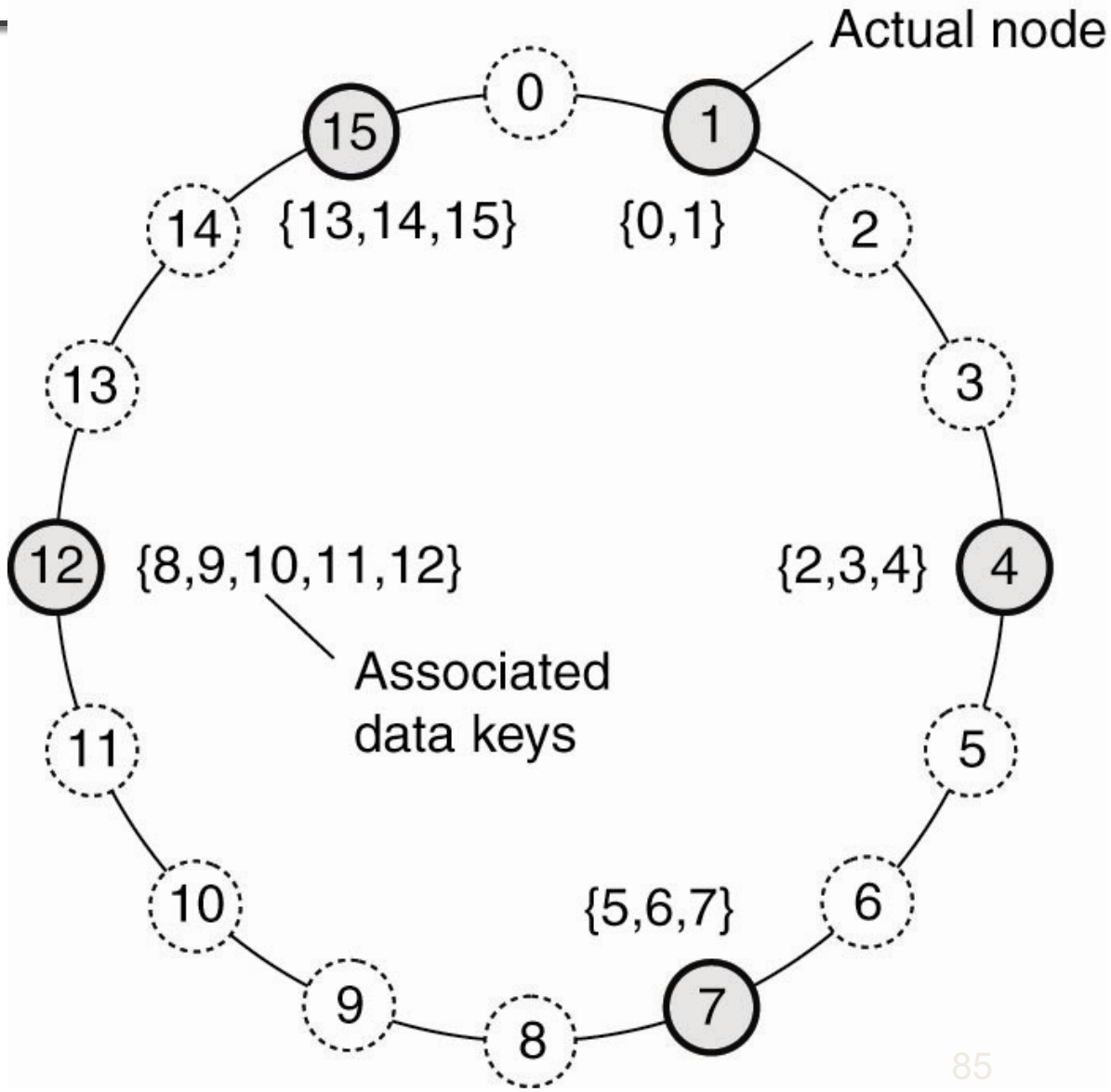
# Multitiered Architectures (2)



# Multitiered Architectures (3)

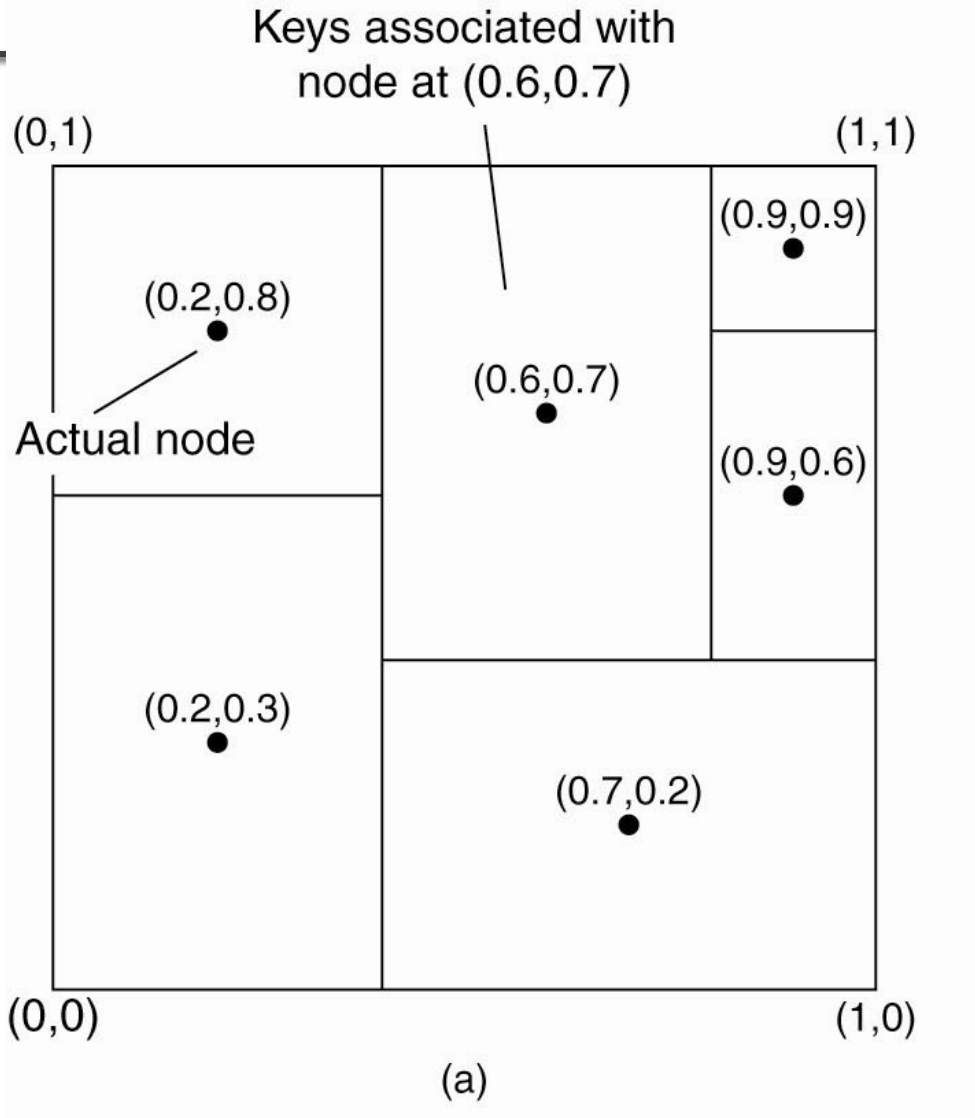


# Structured Peer-to-Peer Architectures (1)



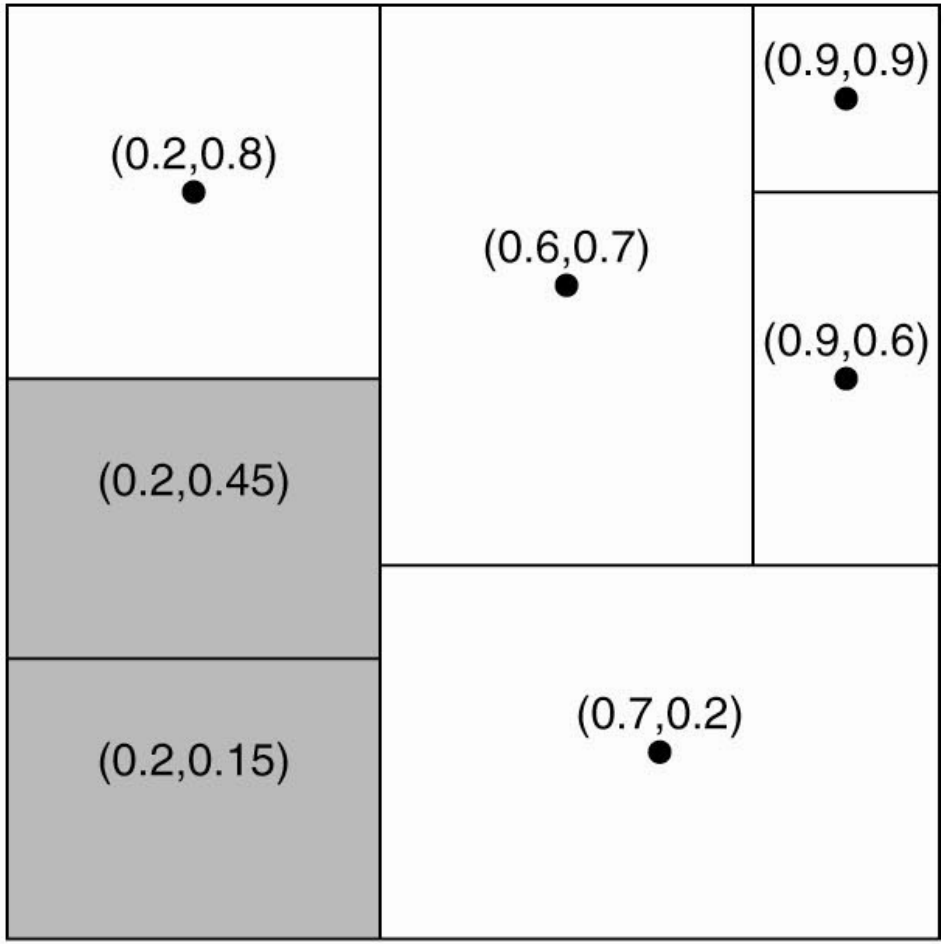

✧ Figure 2-7. The mapping of data items onto nodes in Chord.

# Structured Peer-to-Peer Architectures (2)




✧ Figure 2-8. (a) The mapping of data items onto nodes in CAN.

# Structured Peer-to-Peer Architectures (3)




✧ Figure 2-8. (b) Splitting a region when a node joins.

# Unstructured Peer-to-Peer Architectures (1)

**Actions by active thread (periodically repeated):**

```
select a peer P from the current partial view;
if PUSH_MODE {
    mybuffer = [(MyAddress, 0)];
    permute partial view;
    move H oldest entries to the end;
    append first c/2 entries to mybuffer;
    send mybuffer to P;
} else {
    send trigger to P;
}
if PULL_MODE {
    receive P's buffer;
}
construct a new partial view from the current one and P's buffer;
increment the age of every entry in the new partial view;
```

(a)

✧ Figure 2-9. (a) The steps taken by the active thread.

# Unstructured Peer-to-Peer Architectures (2)

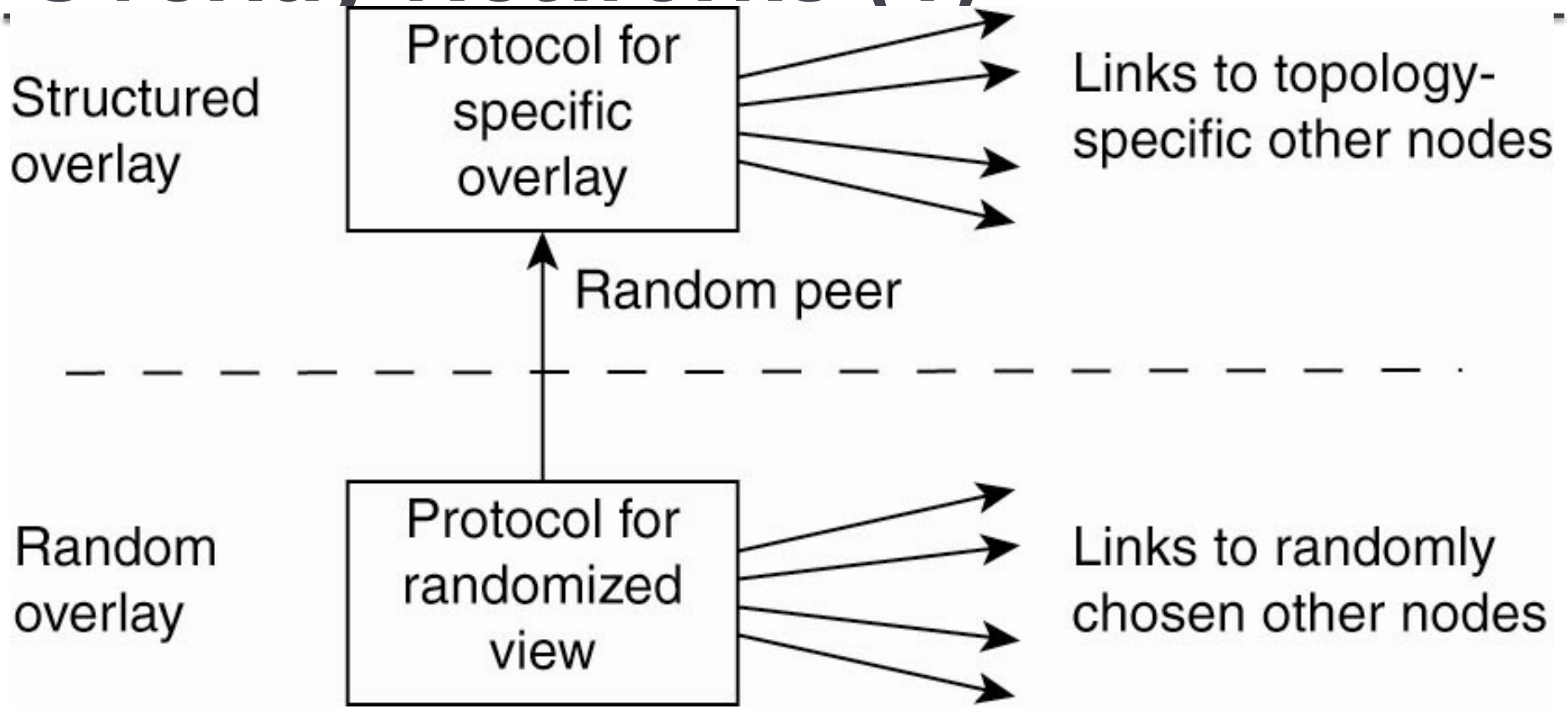
**Actions by passive thread**:

```
receive buffer from any process Q;
if PULL_MODE {
        mybuffer = [(MyAddress, 0)];
        permute partial view;
        move H oldest entries to the end;
        append first c/2 entries to mybuffer;
        send mybuffer to P;
}
construct a new partial view from the current one and P's buffer;
increment the age of every entry in the new partial view;
```

(b)

✧ Figure 2-9. (b) The steps take by the passive thread

# Topology Management of Overlay Networks (1)




✧ Figure 2-10. A two-layered approach for constructing and maintaining specific overlay topologies using techniques from unstructured peer-to-peer systems.

# Superpeers

✧ Figur to a super