Chapter 18

Distributed Software Engineering

Topics covered

- Distributed systems characteristics and issues
- Models of component interaction
- Client–server computing
- Architectural patterns for distributed systems
- Software as a service

Distributed systems

- **Virtually all large computer-based systems are now distributed systems.**
- Processing is distributed over several computers rather than confined to a single machine.
- Appears to the user as a single, coherent system (more or less...).

Distributed system characteristics / advantages

- Resource sharing **(hardware and software)**
- Openness **(standard protocols allow equipment and software from different vendors to be combined)**
- Concurrency **(parallel processing to enhance performance)**
- Scalability **(increased throughput by adding new resources up to capacity of network)**
- Fault tolerance **(potential to continue in operation after a fault has occurred)**

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 generally 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 scalable?

(cont'd)

Design issues (cont'd)

- ***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 how they are distributed.
- In practice, this is impossible because parts of the system are independently managed and because of network delays. (It's 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 developed in any programming language can be integrated into the system and can inter-operate with one another.
- Web service standards for *service-oriented architectures* were developed to be open standards.

Scalability

- The ability of a system to deliver a high quality service as demands on the system increase...
  - ***Size***: *…*by adding more resources to cope with an increasing numbers of users.
  - ***Distribution***: *…*by geographically dispersing components without degrading performance.
  - ***Manageability***: *…*by effectively managing a system as it increases in size, even if its components are located in independent organizations.
- There is a distinction between scaling-up and scaling-out...

(cont'd)

Scaling-up vs. scaling-out

- Scaling-up means replacing resources with more powerful ones.
- Scaling-out means adding additional resources
- Scaling-out is often more cost effective, but usually requires that the system support concurrent processing.

Security

- The number of ways that distributed systems may be attacked is significantly greater than that for centralized systems.
- If a part of the system is compromised then the attacker may be able to use this as a “back door” into other parts of the system.
- Difficulties can arise when different organizations own parts of the system. Security policies and mechanisms may be incompatible.

(cont’d)

Types of attack to defend against

- **Interception** of communications between parts of the system resulting in a loss of confidentiality.
- **Interruption of services,** e.g., a *denial of service attack* whereby a node is flooded with spurious service requests so that it cannot deal with valid ones.
- **Modification** of data or services in the system by an attacker.
- **Fabrication**, e.g., of a false password entry that can be used to gain access to a system.

Quality of service (QoS)

- Reflects a system's ability to deliver services dependably and with a response time and throughput that is acceptable to users.
- QoS is particularly critical when a system deals with time-critical data such as audio or video streams (which could become so degraded that it is impossible to understand).

Failure management

- Since failures are inevitable, distributed systems must be designed to be resilient...

"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 failed components, continuing to deliver as many services as possible and, where possible, automatically recovering from failures.

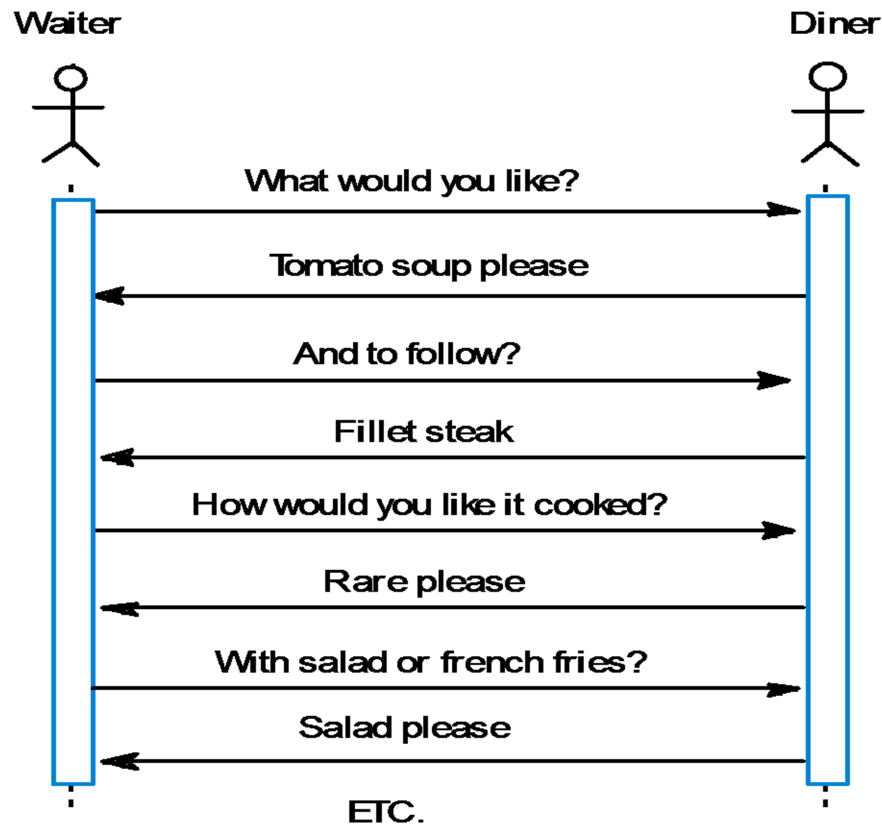
Topics covered

- Distributed systems characteristics and issues
- **Models of component interaction**
- Client–server computing
- Architectural patterns for distributed systems
- Software as a service

Models of component interaction

- There are 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. (synchronous)
  - **Message-based interaction,** involves the computer sending information about what is required to another computer. There is no necessity to wait for a response. (non-synchronous)

Procedural interaction between a diner and a waiter via synchronous procedure calls



Message-based interaction between a waiter and the kitchen via XML

```
<starter>
      <dish name = “soup”   type = “tomato” />
      <dish name = “soup”   type = “fish” />
      <dish name = “pigeon salad” />
</starter>
<main course>
      <dish name = “steak”  type = “sirloin”  cooking = “medium” />
      <dish name = “steak”  type = “fillet”  cooking = “rare” />
      <dish name = “sea bass”>
</main course>
<accompaniment>
      <dish name = “french fries”  portions = “2” />
       <dish name = “salad”  portions = “1” />
</accompaniment>
```

Procedural (synchronous) Interaction

- Implemented by Remote Procedure Calls (RPC's).
- One component calls another (via a "stub") as if it were a local procedure or method.
- Middleware intercepts the call and passes it to the remote component which carries out the required computation and returns the result.
- A problem is that both components need to be available at the time of the communication and must know how to refer to each other.

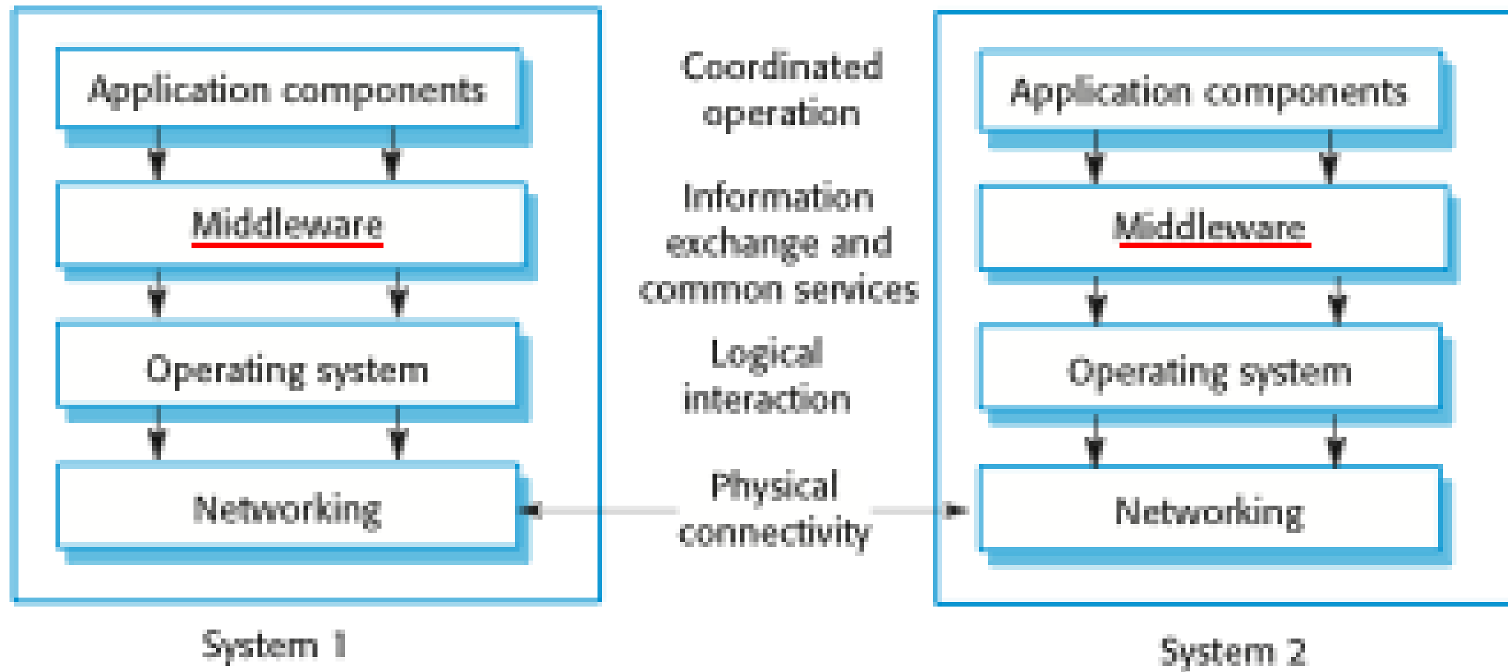
Message-based (asynchronous) Interaction

- Normally involves one component creating a message that details the services required of another.
- Message is sent via middleware to the receiving component which parses the message, carries out the computations, and (possibly) creates a return message with the required results.
- Messages are queued until the receiver is available. Components need NOT refer to each other; middle-ware ensures that messages are passed to the appropriate component.

Middleware

- The components in a distributed system may be implemented in different programming languages and may execute on different processors.
- Models of data, information representation, and protocols for communication may all be different.
- Middleware is software that can manage these diverse components, and ensure that they can communicate and exchange data.
- Examples include: CORBA, DCOM, .NET

Middleware in a distributed system



Types of middleware support

- **Interaction support**: coordinates interactions between different components in the system.
  - Provides *location transparency* whereby it isn't necessary for components to know the physical locations of other components.
- **Common services**: provides services that may be required by different components regardless of their functionality (i.e., security, notification, transaction management, etc.).
  - Supports inter-operability and service consistency.

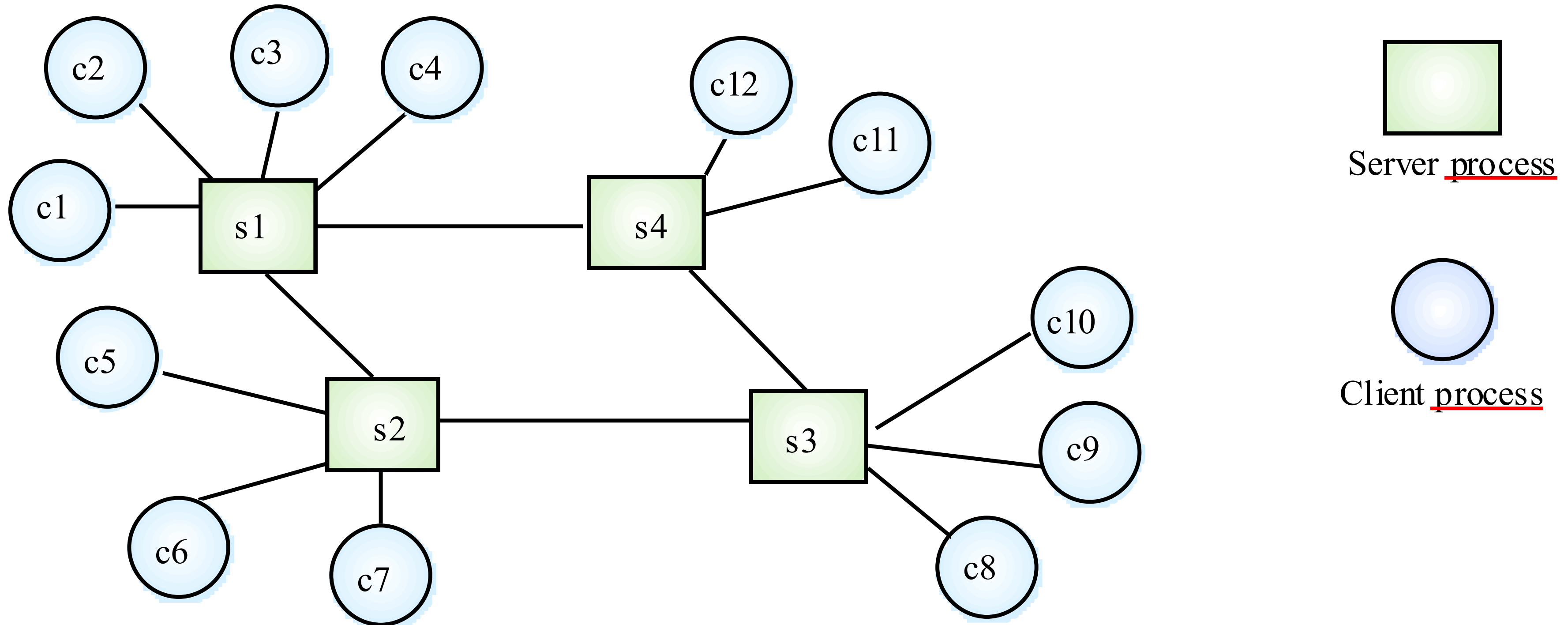
Topics covered

- Distributed systems characteristics and issues
- Models of component interaction
- **Client–server computing**
- Architectural patterns for distributed systems
- Software as a service

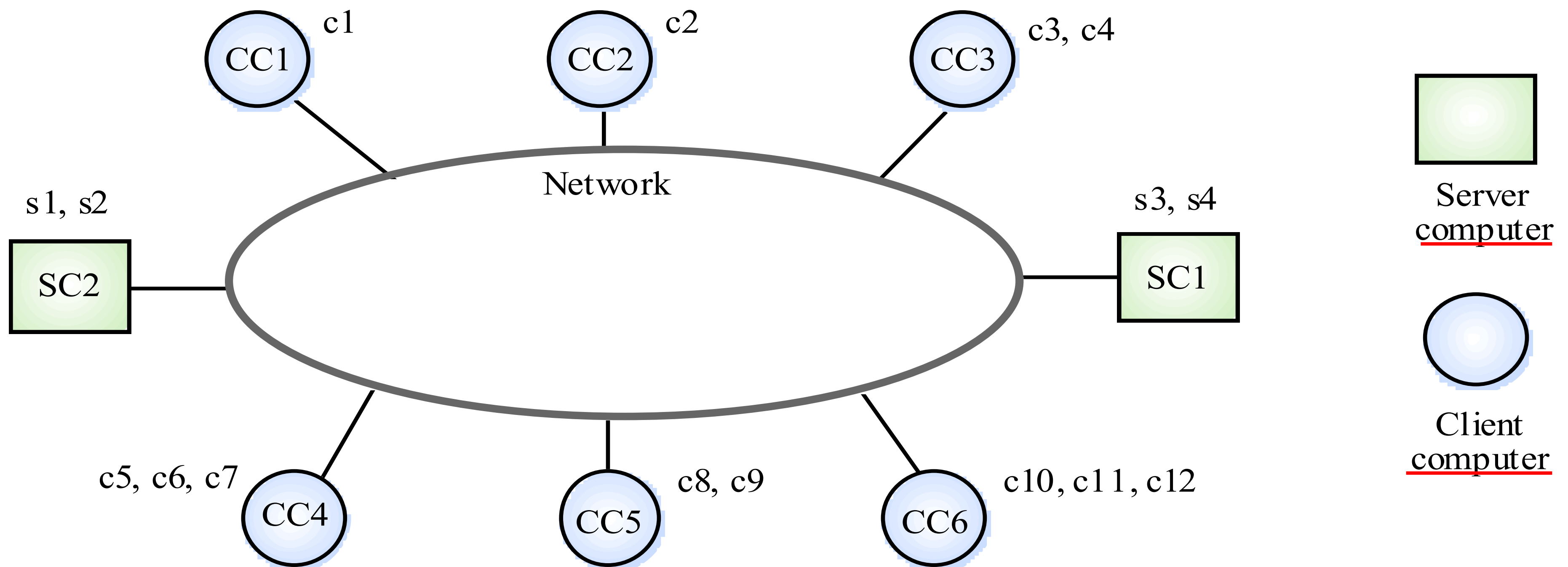
Client-server computing

- Distributed systems that are accessed over the Internet are often organized as client-server (C/S) systems.
- The user interacts with a program running on a local computer (e.g., a web browser or phone-based application) which 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 **clients**.

Processes in a client-server system



Mapping of client and server processes to networked computers



Layered architectural model for client-server applications

Presentation layer

Data management layer

Application processing layer

Database layer

Topics covered

- Distributed systems characteristics and issues
- Models of component interaction
- Client–server computing
- **Architectural patterns for distributed systems**
- Software as a service

Distributed system architectural patterns

- **Master-slave:** used in real-time systems for which guaranteed interaction response times are required.
- **Two-tier client-server:** used for simple client-server systems, and where the system is centralized for security reasons.
- **Multi-tier client-server:** used when there is a high volume of transactions to be processed by the server.
- **Distributed component:** 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:** used when clients exchange locally stored information and the role of the server is to introduce clients to each other.

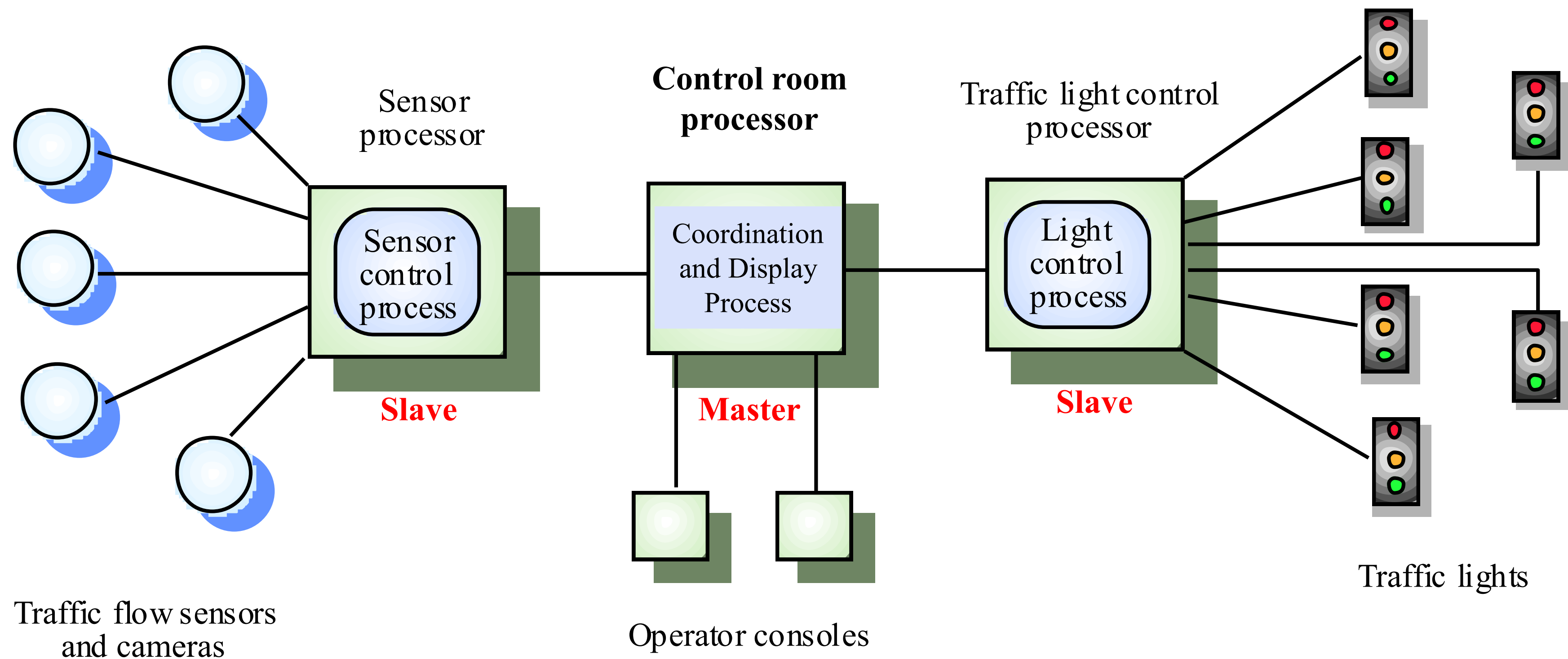
Distributed system architectural patterns

- **Master-slave:** used in real-time systems for which guaranteed interaction response times are required.
- **Two-tier client-server:** used for simple client-server systems, and where the system is centralized for security reasons.
- **Multi-tier client-server:** used when there is a high volume of transactions to be processed by the server.
- **Distributed component:** 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:** used when clients exchange locally stored information and the role of the server is to introduce clients to each other.

Master-slave architectures

- Commonly used in real-time systems with separate processors associated with data acquisition, data processing, and computation and actuator management.
- A **"Master"** process is usually responsible for computation, coordination, and communications; it controls the "slave" processes.
- **"Slave"** processes are dedicated to specific actions, e.g., the acquisition of data from an array of sensors.

A traffic management system with a master-slave architecture



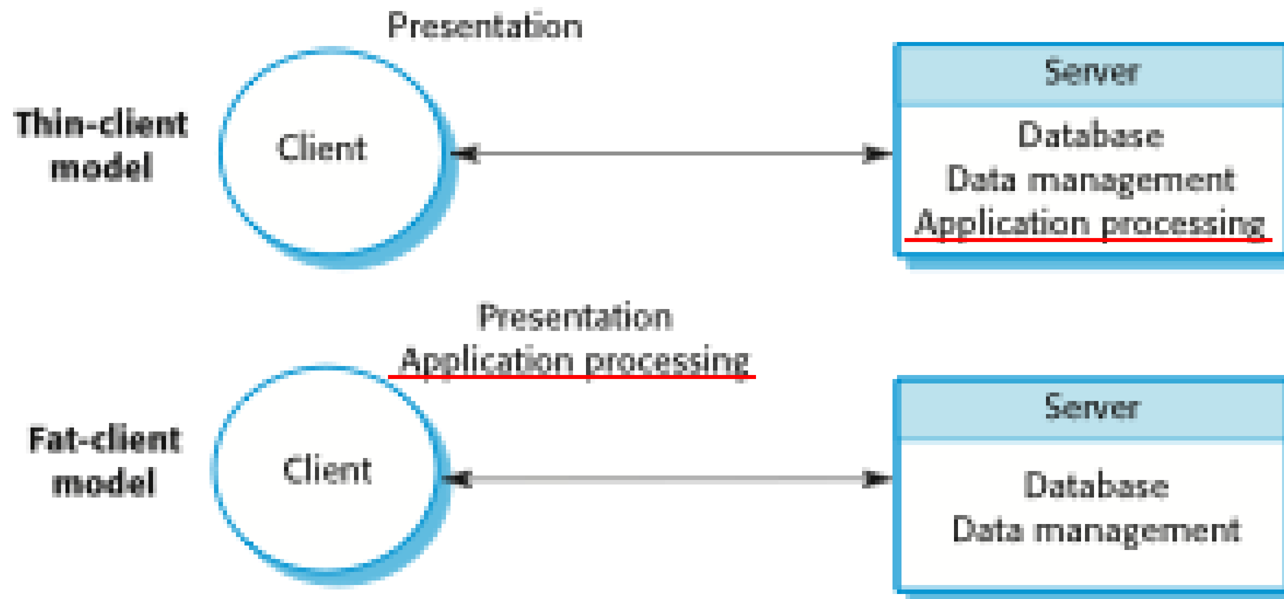
Distributed system architectural patterns

- **Master-slave:** used in real-time systems for which guaranteed interaction response times are required.
- **Two-tier client-server:** used for simple client-server systems, and where the system is centralized for security reasons.
- **Multi-tier client-server:** used when there is a high volume of transactions to be processed by the server.
- **Distributed component:** 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:** used when clients exchange locally stored information and the role of the server is to introduce clients to each other.

Two-tier C/S architecture

- System is implemented as a single logical server plus some number of clients that use that server.
  - Thin-client model: presentation layer is implemented on the client; all other layers (data management, application processing and database) are on the server.
  - Fat-client model: some or all application processing is implemented on the client; data management and database functions are on the server.

Thin- and fat-C/S architectures



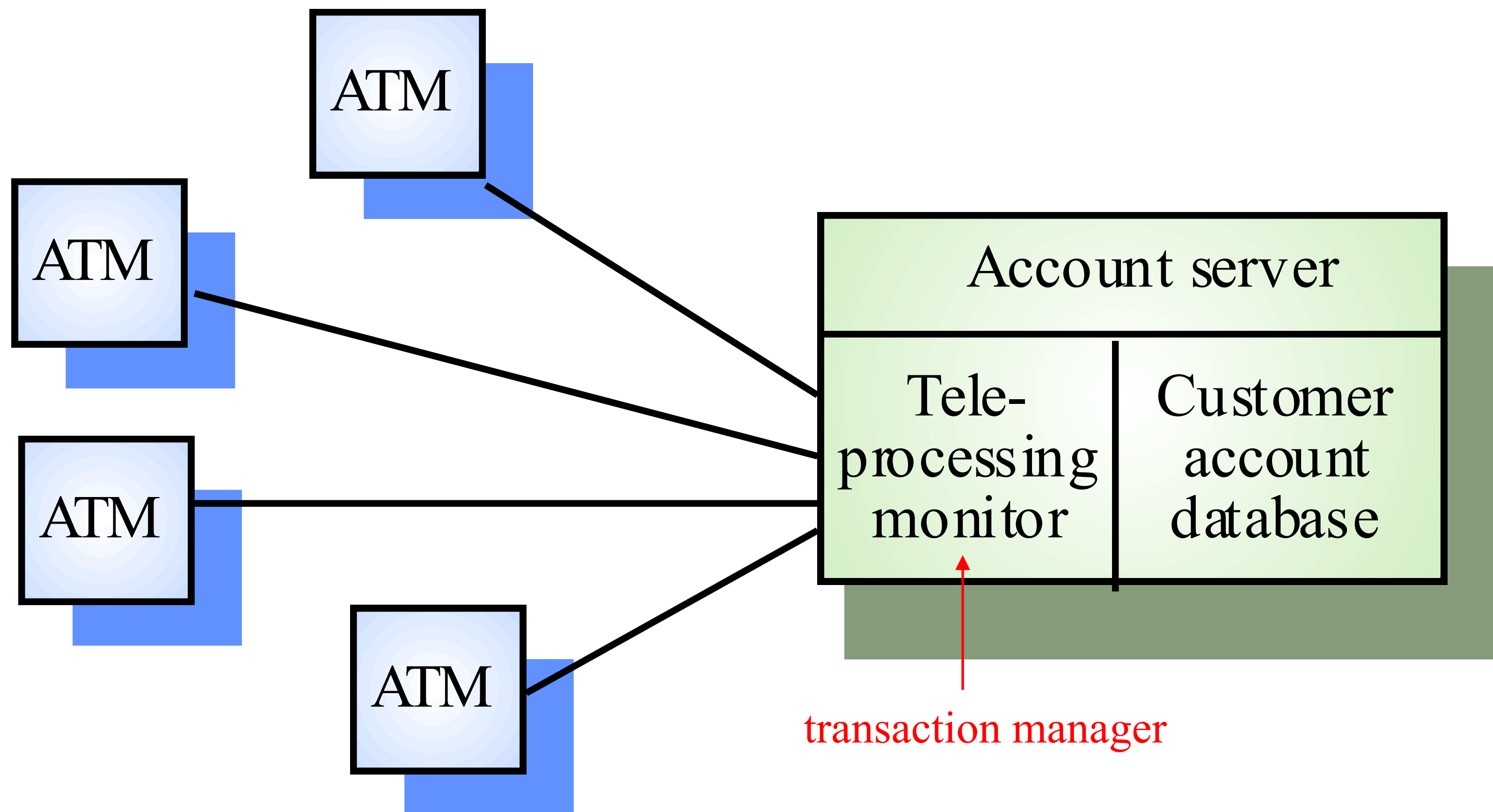
Thin-client model

- Often **used when centralized legacy systems evolve to a C/S architecture;** a graphical interface is implemented on clients and the legacy system acts as a server.
- Disadvantage: places a **heavy processing load on both the server and the network**.

Fat-client model

- **More processing is delegated to the client.**
- Most suitable for new C/S systems where client capabilities are known in advance.
- System **management is more complex**. When application functionality changes, updates are required on each client.

A fat-client C/S architecture for an ATM system



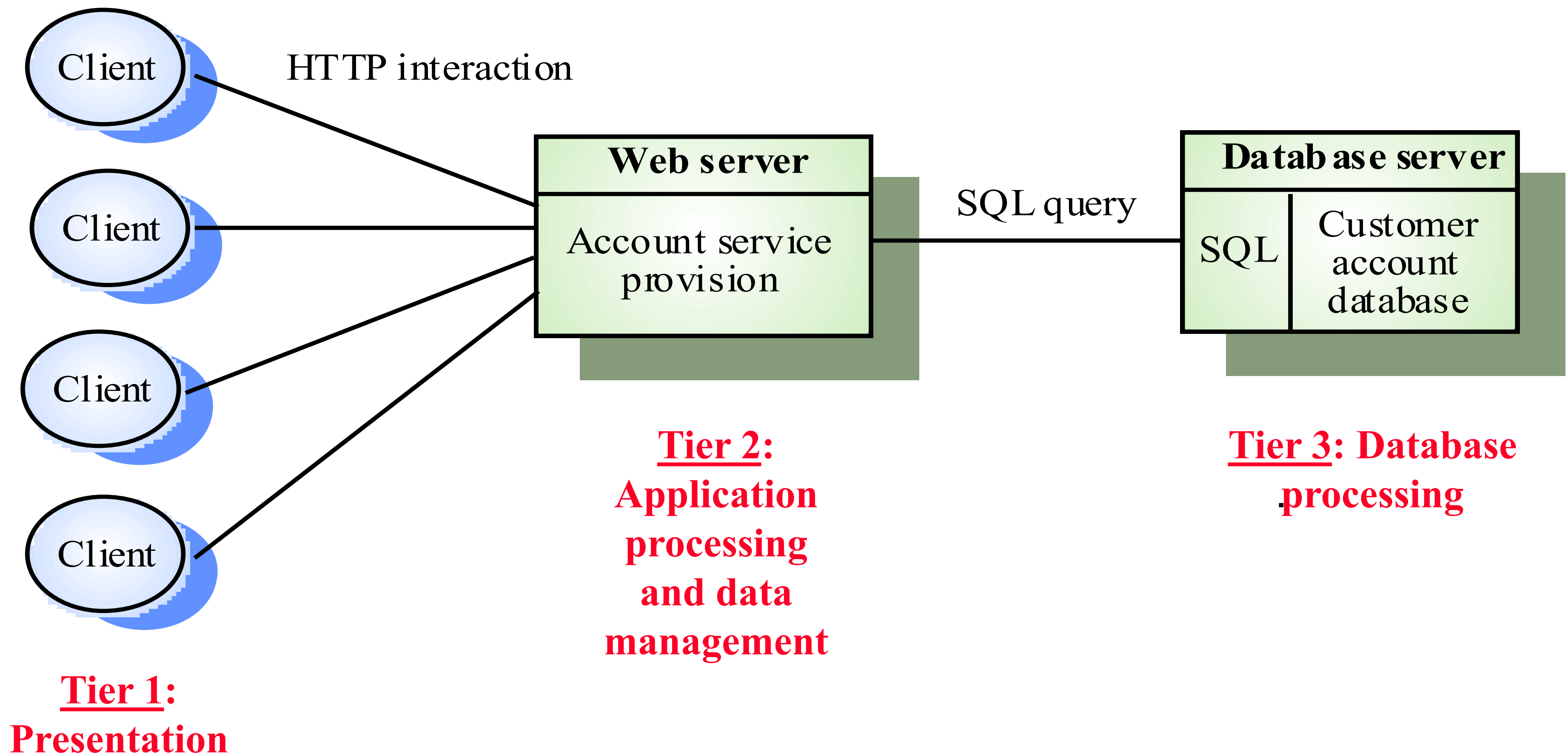
Distributed system architectural patterns

- **Master-slave:** used in real-time systems for which guaranteed interaction response times are required.
- **Two-tier client-server:** used for simple client-server systems, and where the system is centralized for security reasons.
- **Multi-tier client-server:** used when there is a high volume of transactions to be processed by the server.
- **Distributed component:** 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:** used when clients exchange locally stored information and the role of the server is to introduce clients to each other.

Multi-tier C/S architecture

- Each layer may execute on a separate processor.
- Allows for:
  - better performance and scalability than the thin-client, approach, and is
  - simpler to manage than a fat-client approach. **(Why?)**

Three-tier architecture for an internet banking system



Use of C/S 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. **Computationally intensive applications** such as compilers with little or no data management. **Data-intensive applications (browsing and querying) with non-intensive application processing**. Browsing the Web is the most common example of a situation where this architecture is used.

(cont’d)

Use of C/S 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. Applications where **computationally intensive processing of data (e.g., data visualization) is required**. **Mobile applications where internet connectivity cannot be guaranteed**. Some local processing using cached information from the database is therefore possible.

Use of C/S architectural patterns

Architecture	Applications
Multi-tier client–server architecture	**Large-scale applications** with hundreds or thousands of clients. Applications where **both the data and the application are volatile**. Applications where **data from multiple sources are integrated**.

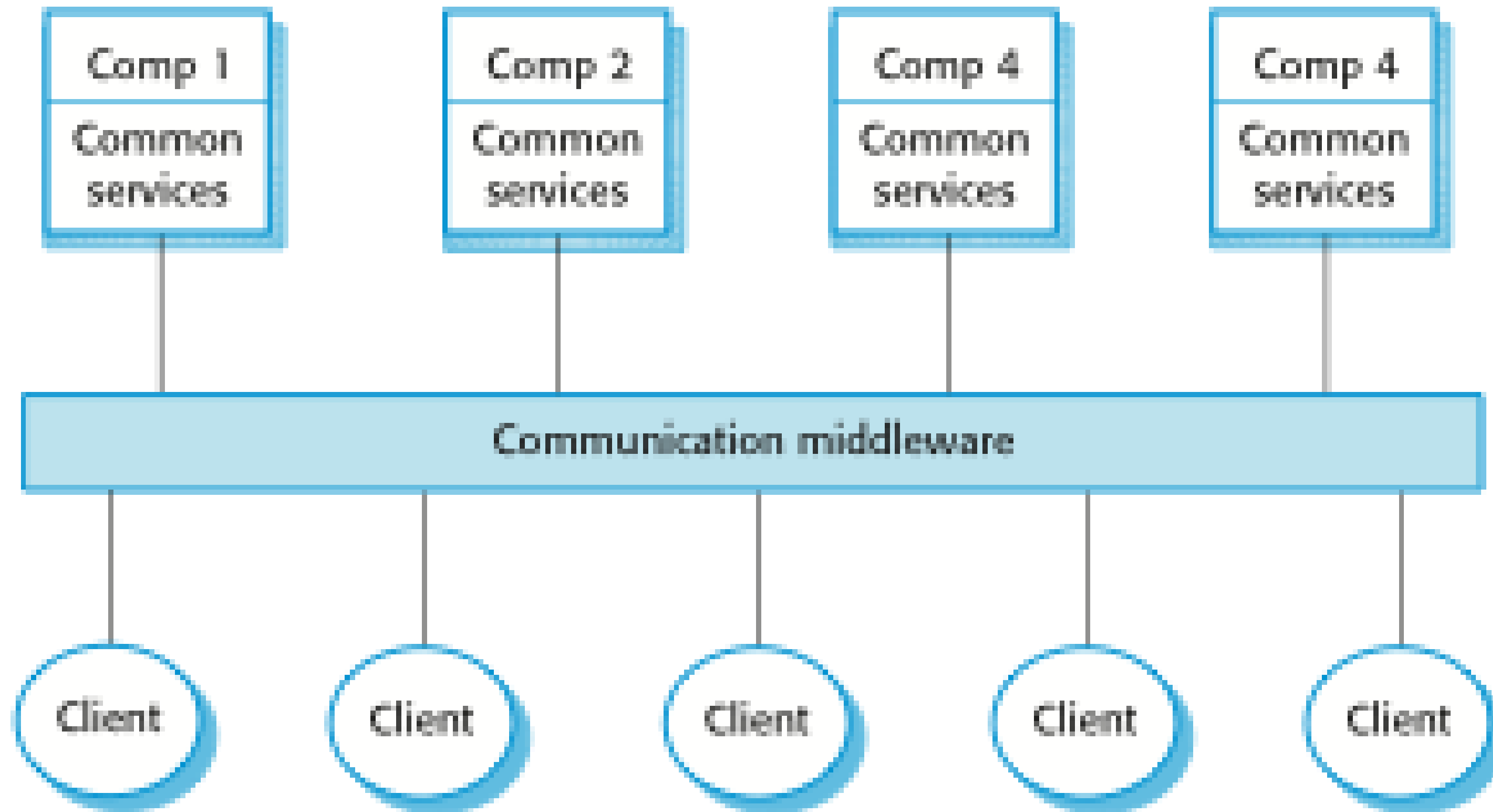
Distributed system architectural patterns

- **Master-slave:** used in real-time systems for which guaranteed interaction response times are required.
- **Two-tier client-server:** used for simple client-server systems, and where the system is centralized for security reasons.
- **Multi-tier client-server:** used when there is a high volume of transactions to be processed by the server.
- **Distributed component:** 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:** used when clients exchange locally stored information and the role of the server is to introduce clients to each other.

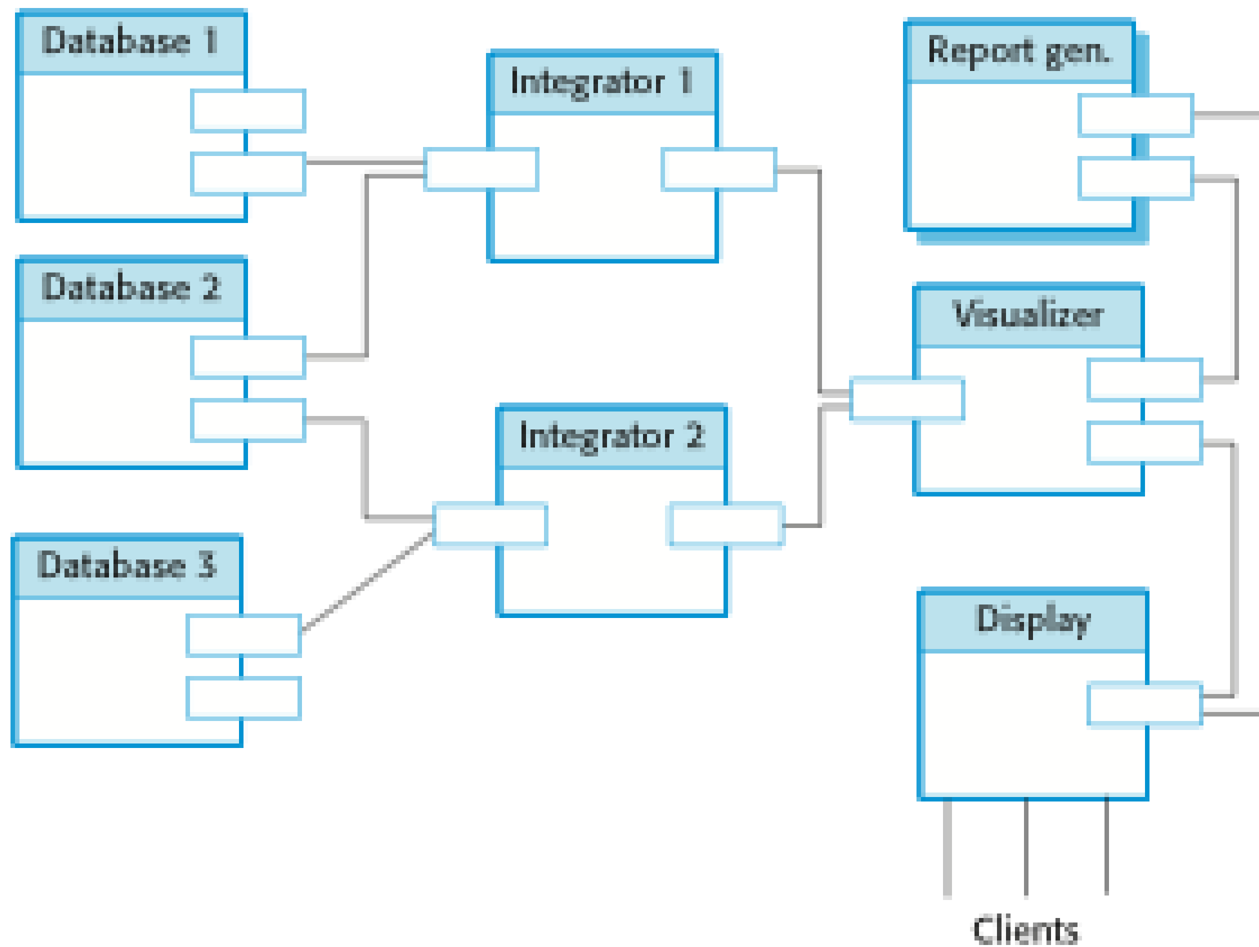
Distributed component architectures

- System is designed as a set of services, without attempting to allocate these services to layers in the system.
- Each service (or group of related services) is implemented as a separate component or object.
- Components communicate via middleware using remote procedure or method calls.

Generic distributed component architecture



A distributed component architecture of a DATA MINING SYSTEM



Advantages of distributed component architectures

- Allows developers to delay decisions on where and how services should be provided. **(Service-providing components may execute on any network node.)**
- Very open architecture – new resources can be added as required.
- System is dynamically reconfigurable – components can migrate across the network as required. **(Thus improving performance.)**
- Therefore: **flexible** and **scalable**.

Disadvantages of distributed component architectures

- Less intuitive/natural than C/S: more difficult to visualize, understand, and design.
- Competing middleware standards: vendors, such as Microsoft and Sun, have developed different, incompatible middleware systems.

As a result, service-oriented architectures (SOA's) are replacing distributed component architectures in many situations.

Distributed system architectural patterns

- **Master-slave:** used in real-time systems for which guaranteed interaction response times are required.
- **Two-tier client-server:** used for simple client-server systems, and where the system is centralized for security reasons.
- **Multi-tier client-server:** used when there is a high volume of transactions to be processed by the server.
- **Distributed component:** 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:** used when clients exchange locally stored information and the role of the server is to introduce clients to each other.

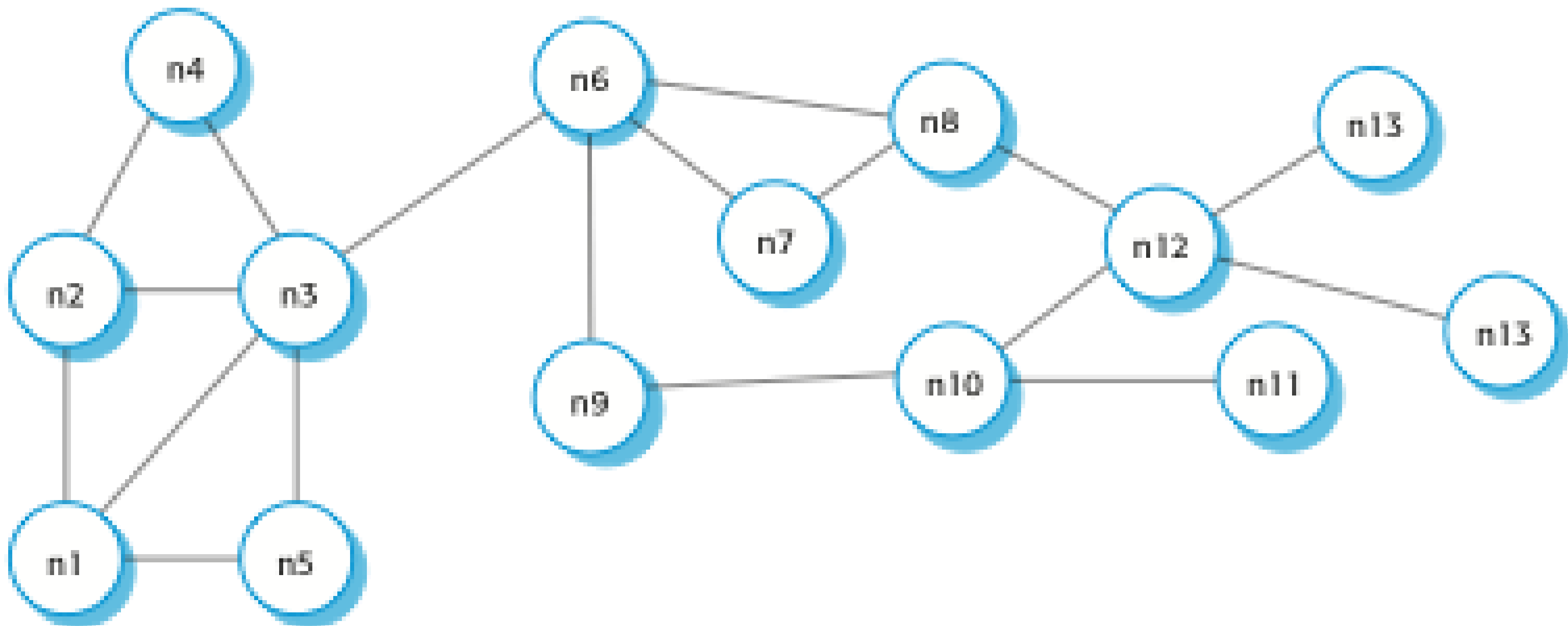
Peer-to-peer (p2p) architectures

- Decentralized systems utilize the power and storage of a large number of networked computers *running the same application*.
- Computations may therefore be carried out by any node in the network.
- Most systems have been personal in nature (e.g., file sharing on PCs), but the number of business and scientific applications is increasing.
  (Folding@home, SETI@home, VOIP, etc.)

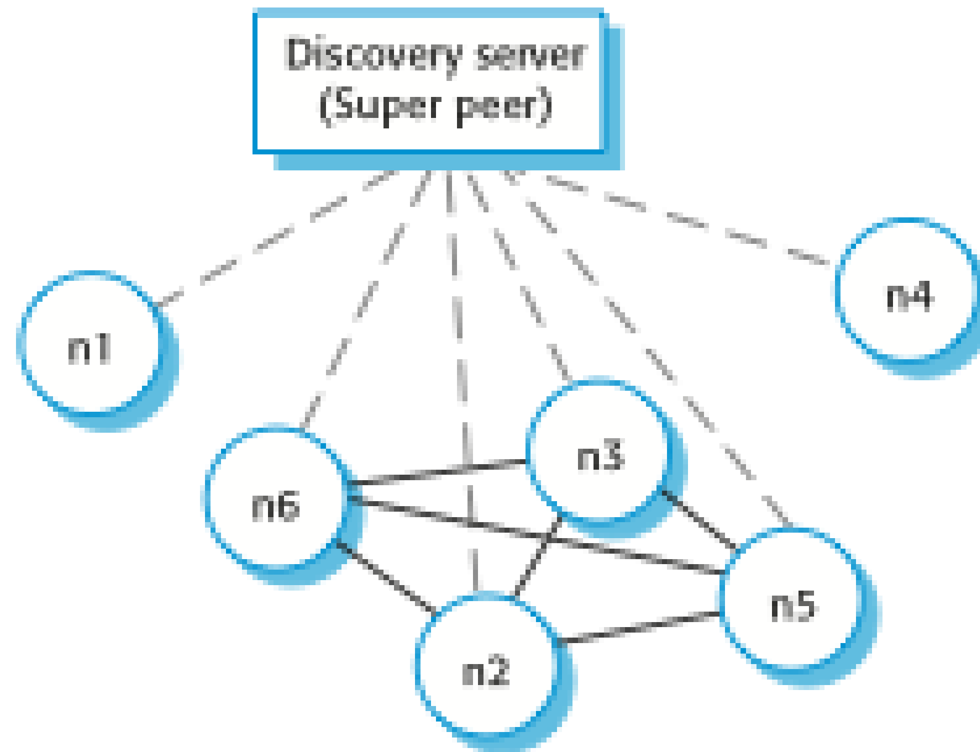
Appropriate uses of p2p

1. Computationally intensive applications where processing can be efficiently distributed among a large number of networked computers that need not communicate with one another. (E.g., Folding@home)
2. Applications where processing involves a large number of networked computers exchanging data that need not be centrally stored or managed. (E.g., file sharing)

A decentralized p2p architecture



A “semi-centralized” p2p architecture



P2p problems

- The major concerns that have inhibited p2p use are **security** and **trust**.
- When p2p nodes interact with one another, any resources could potentially be accessed.
- Problems may also occur if peers deliberately behave in a malicious way.

Topics covered

- Distributed systems characteristics and issues
- Models of component interaction
- Client–server computing
- Architectural patterns for distributed systems
- **Software as a service**

Software as a Service (SaaS)

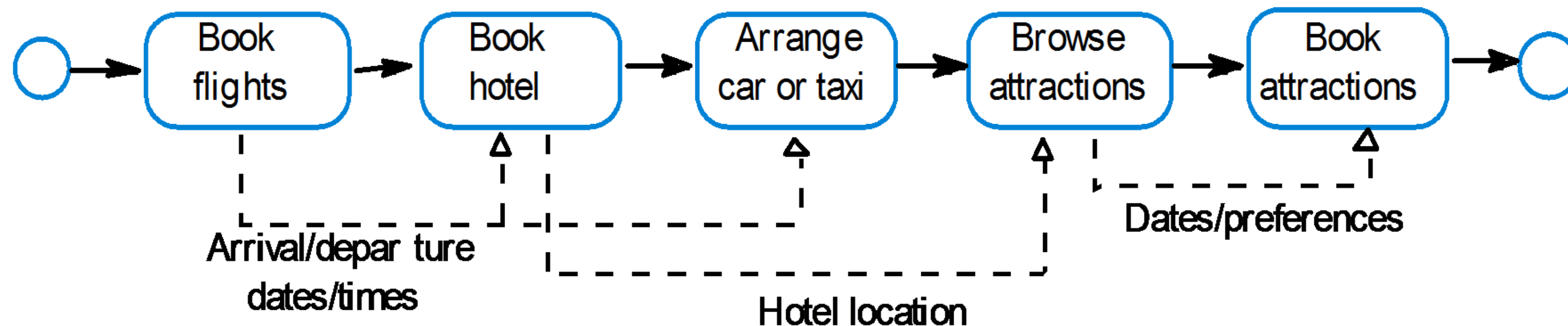
- Involves hosting software remotely on servers ("the cloud") with access provided over the Internet via web browsers.
- The server maintains user data and state during a trans-action session.
- Applications are owned and managed by a software pro-vider rather than users.
- Users may pay for access according to the amount of use, or through an annual or monthly subscription.
- If free, users are exposed to advertisements which fund the software service.

Differences between SaaS and "Service-oriented architecture" (SOA)

- SOA (Chap. 19) is an implementation technology for structuring a system as a set of separate, stateless services.
- Services may be owned and managed by multiple providers and may be distributed.
- Existing services may be composed and configured to create new *composite* services and applications.
- The basis for service composition is often a workflow.

Differences between SaaS and "Service-oriented architecture" (SOA) (cont'd)

- Example SOA for a "Vacation Package" workflow:



Differences between SaaS and "Service-oriented architecture" (SOA) (cont'd)

- Individual SOA transactions are typically "brief," whereby a service is called, does something, and returns a result.
- SaaS transactions, in contrast, are usually "long," e.g., editing a document.

(This is a very simplistic generalization – it is not a reliable discriminator!)

Differences between SaaS and “Service-oriented architecture” (SOA) (cont’d)

- In summary:
  - SaaS is simply a **general software delivery method** whereby a software system is hosted remotely on a provider’s server (a “cloud”) – e.g., web-based e-mail.
  - SOA is a **specific implementation strategy for designing and building software products** through the composition of existing capabilities and services.
  - Thus, *delivering* tax capabilities *over the web* is SaaS, while *enabling a tax application to integrate with IRS services for tax form checking and e-filing* is SOA.

Implementation considerations for SaaS

- ***Service configuration***: How do you configure the software for the specific requirements of different users/organizations?
- ***Multi-tenancy:*** How do you give users the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?
- ***Scalability:*** How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

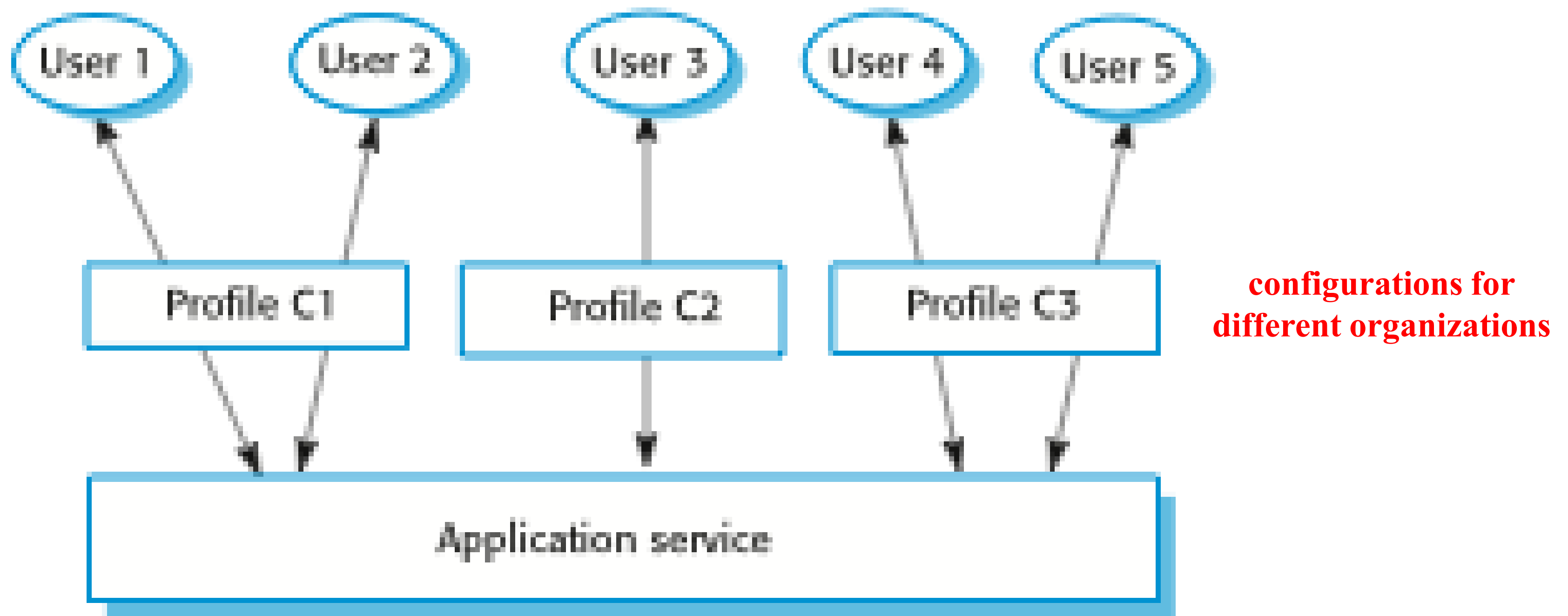
Implementation considerations for SaaS

- ***Service configuration***: How do you configure the software for the specific requirements of different users/organizations?
- ***Multi-tenancy:*** How do you give users the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?
- ***Scalability:*** How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

SaaS service configuration may support:

- Branding: users can be presented with an interface that reflects their own organization.
- Business rules and workflows: rules that govern the use of the service and its data can be organization specific.
- Database extensions: each organization can define extensions to the generic service data model that meet its specific needs.
- Access control: organizations can create individual accounts for their staff and define the resources and functions that are accessible to each of them.

Configuration of a software system offered as a service



configurations for different organizations

Implementation considerations for SaaS

- ***Service configuration***: How do you configure the software for the specific requirements of different users/organizations?
- ***Multi-tenancy:*** How do you give users the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?
- ***Scalability:*** How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

Multi-tenancy

- Multi-tenancy involves defining a system architecture that:
  1. allows many different users to access and efficiently share system resources, and
  2. gives each of those users the impression that he is the sole user of the system.
- This requires designing the system so that there is an absolute separation between the system functionality and the user data and state .

Tenant identifiers in a multi-tenant database

Tenant	Key	Name	Address
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

Implementation considerations for SaaS

- ***Service configuration***: How do you configure the software for the specific requirements of different users/organizations?
- ***Multi-tenancy:*** How do you give users the impression that they are working with their own copy of the system while, at the same time, making efficient use of system resources?
- ***Scalability:*** How do you design the system so that it can be scaled to accommodate an unpredictably large number of users?

General guidelines for achieving scalability

- Develop applications where components are implemented as simple, stateless services that can run on *any* server.
- Design the system using *message-based (non-synchronous) interaction* so that the application does not have to wait for the result of an inter-action (such as a read request).

(cont'd)

General guidelines for achieving scalability (cont'd)

- Manage resources such as network and database connections as a ***pool*** so that no single server is likely to run out of resources.
- Design your database to allow ***fine-grain locking.*** That is, do not lock out whole records when only *part* of a record is in use.

Key points

- **Important benefits of distributed systems:** they can be scaled to cope with increasing demand, they can continue to provide user services if parts of the system fail, and they enable resources to be shared.
- **Issues to be considered in the design of distributed systems:** transparency, openness, scalability, security, quality of service and failure management.

(cont'd)

Key points (cont'd)

- Client-server systems are **structured into layers**, with the presentation layer implemented on a client computer. Servers provide data management, application, and database services.
- Client-server systems **may have several tiers**, with different layers of the system distributed to different computers.

(cont'd)

Key points (cont'd)

- **Architectural patterns for distributed systems include:** master-slave architectures, two-tier and multi-tier C/S architectures, distributed component architectures, and peer-to-peer architectures.
- **Distributed component systems** are designed as a set of services, without attempting to allocate these services to layers. Middleware handles component communication.

(cont'd)

Key points (cont'd)

- **Peer-to-peer architectures** support decentralized systems that utilize the power and storage of a large number of networked computers *running the same application*.
- **Software as a service (SaaS)** is a way of deploying applications as thin-client C/S systems where the software is hosted remotely on servers ("the cloud") and the client is a web browser.

Chapter 18

Distributed Software Engineering