Chapter 17 – Distributed software engineering

1

Topics covered

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

2

Distributed systems

- ✧ Virtually all large computer-based systems are now 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.
- ✧ Even apparently self-contained applications on a PC or laptop (e.g. image editors) are essentially distributed systems, executing on a single computer but relying on remote cloud systems for update, storage, etc.
- ✧ Distributed software engineering is therefore very important for enterprise computing systems.

3

Distributed system characteristics/benefits

- ✧ Resource sharing
  - Sharing of hardware and software resources (disks, printers, files, etc.) that are associated with computers on a network.
- ✧ Openness
  - Use of equipment and software from different vendors.
- ✧ Concurrency
  - Concurrent processing to enhance performance.
- ✧ Scalability
  - Increased throughput by adding new resources to cope with new demands on the system.
- ✧ Fault tolerance
  - The ability to continue in operation after a fault has occurred, due to the potential of replicating information.

4

Distributed systems

5

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 by their owners, as is the network.
- ✧ There is no single authority in charge of the distributed system so top-down control is impossible.
- ✧ There is, therefore, an inherent unpredictability in the operation of distributed systems that has to be taken into account when designing such a system.

6

Design issues

- *Transparency.* To what extent should the distributed system appear to the user as a single system? When is it useful for users to understand that the system is distributed?
- *Openness.* Should a system be designed using standard protocols that support interoperability? Not the case is service communication.
- *Scalability.* How can the system be constructed so that it is scalable (i.e. its capacity increases in response to demand)?
- *Security.* How can usable security policies be defined and implemented that apply across independently managed systems?
- *Quality of service.* How should the quality of service be specified and how could acceptable quality be delivered to all users?
- *Failure management.* How can system failures be detected, contained (to cause minimal disruption) and repaired?

7

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.

8

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.

9

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 leads to a more powerful system; scaling out leads to more system instances.

10

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.

11

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.

12

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.
- Ideally, the QoS requirements should be specified in advance and the system designed to deliver them, but:
  - It may not be cost-effective (in terms of needed resources) to continue to provide high quality at peak time (note, though, that cloud computing techniques have now alleviated this issue).
  - QoS parameters may be mutually contradictory: increased reliability may mean reduced throughput, as checking procedures are introduced to ensure data validity.

13

Quality of service

- Quality of service is particularly critical when the system is dealing with time-critical data such as sound or video streams.
  - 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.
  - Systems dealing with sound and video should include QoS negotiation and management components.
  - These should evaluate the QoS requirements against the available resources and, if these are insufficient, negotiate for more resources or for a reduced QoS target.

14

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." (Leslie Lamport)
- 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.

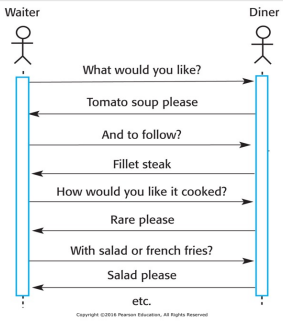
15

Models of interaction

- Two types of interaction between components in a D.S.:
  - 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. Messages usually transmit more information in a single interaction than a procedure call to another machine.
- Consider the ordering of a meal in a restaurant:
  - The waiter has a series of synchronous procedural communications with clients to get individual orders.
  - Then he may have a single asynchronous message-based communication with the kitchen to deliver all the orders.

16

Procedural interaction between a diner and a waiter



17

Message-based interaction between a waiter and the kitchen

```
<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>
<accompaniment>
        <dish name = "french fries" portions = "2" />
        <dish name = "salad" portions = "1" />
</accompaniment>
```


18

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.

19

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.

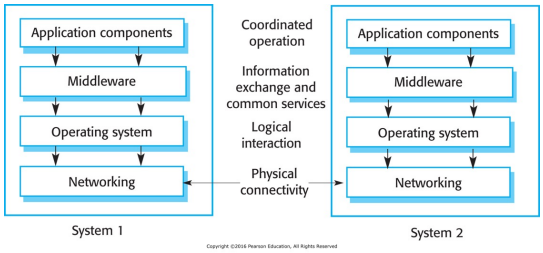
20

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.
- It is usually implemented as a set of libraries, which are installed on each different distributed computer, plus a runtime system to manage communications.

21

Middleware in a distributed system



22

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 (such as authentication, authorization, notification, naming, transaction management), components can easily inter-operate and provide user services in a consistent way.

23

Client-server computing

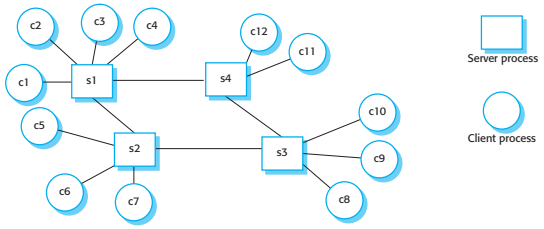
24

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.
- Note that the client-server model is not restricted to D.S. and can also be used as a logical interaction model where client and server are on the same machine.

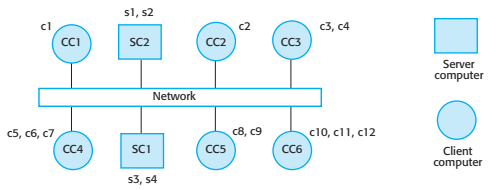
25

Client–server interaction



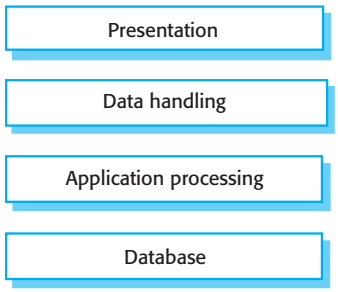
26

Mapping of clients and servers to networked computers



27

Layered architectural model for client–server applications



28

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. Implements checks on the data, generates 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, query services, etc.

29

Architectural patterns for distributed systems

30

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. It may also be used when a large number of independent computations may have to be made.

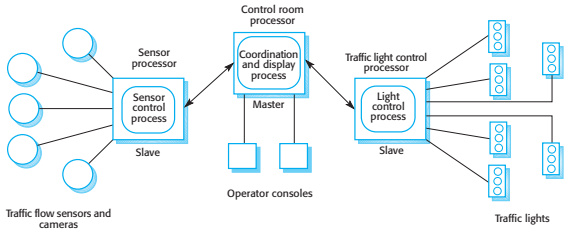
31

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.

32

A traffic management system with a master-slave architecture



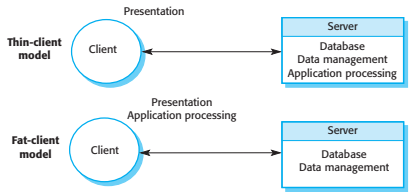
33

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. The client presentation software is usually a web browser, but apps for mobile devices may also be available.
  - 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. In this case, the client software may be a specially written program that is tightly integrated with the server application.

34

Thin- and fat-client architectural models



35

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.
- The advantage of the thin-client model is that it is simple to manage the clients, a major issue when there are many of them and otherwise it may be expensive and difficult to install software on all of them.
- A major disadvantage is that it places a heavy processing load on both the server and the network, as the former is responsible for all computation, which may generate a lot of messages between servers and clients.

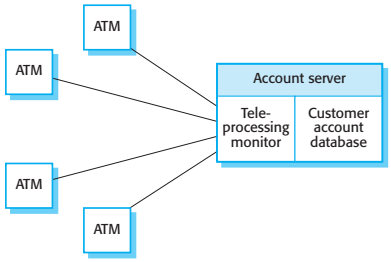
36

Fat-client model

- More processing is delegated to the client as the application processing is locally executed.
- The server is essentially a transaction server that manages all database transactions.
- Data handling is straightforward as there is no need to manage the interaction between the client and the application processing system.
- 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 and maintained on all clients.

37

A fat-client architecture for an ATM system



38

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.

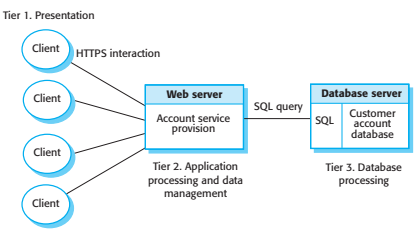
39

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.
- In an internet banking system, the customer database is hosted on a mainframe computer, a web server provides data management services, and the customer's own computer with a browser is the client.

40

Three-tier architecture for an Internet banking system



41

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. 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.

42

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. 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.
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.

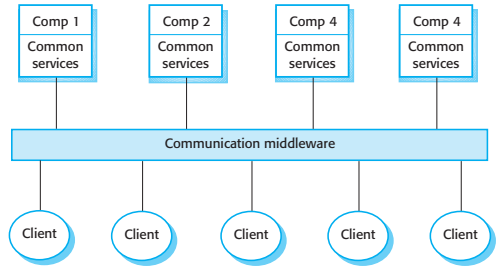
43

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 (such as CORBA, EJB or .NET), which manages component interactions, reconciles differences between types of parameters passed between components, and provides a set of common services that application components can use.

44

A distributed component architecture



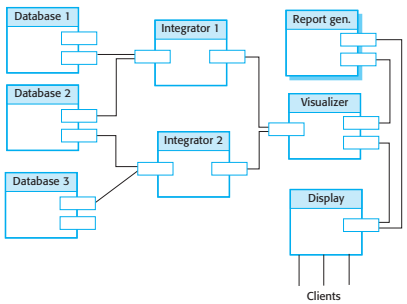
45

Benefits of distributed component architecture

- It allows the system designer to delay decisions on where and how services should be provided. Service-providing components may execute on any node of the network and there is no need to decide in advance whether a service is part of one specific layer or another.
- It is a very open system architecture that allows new resources to be added as required and without disruption to the existing system.
- The system is flexible and scalable and new or replicated objects can be added as the load increases.
- It is possible to reconfigure the system dynamically with objects migrating across the network as required.

46

A distributed component architecture for a data mining system



47

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, as they are not to humans as intuitive as layer client-server systems are (CS systems often reflect the way people interact between themselves in requesting/receiving services).
  - Standardized middleware for distributed component systems has never been accepted by the community. Different vendors (Sun, Microsoft) have developed different, incompatible middleware.
- As a result of these problems, service-oriented architectures are replacing distributed component architectures in many situations.

48

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.
- It contrasts with the client-server model in that it does not make any distinction between servers (providers of services) and clients (receivers of services).

49

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

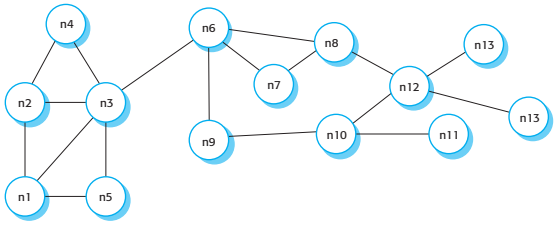
50

P2p architectural models

- The logical network architecture
  - *Decentralized architectures*. They are highly redundant and hence both fault-tolerant and tolerant to nodes disconnecting from the network. But many different nodes may process the same search and there is significant overhead in replicated peer communication.
  - *Semi-centralized architectures*. It involves a server (sometimes called a super-peer) that helps to establish contact between peers on the network or coordinate the results of a computation.
- Application architecture
  - The generic organization of components making up a p2p application.
- Focus here on network architectures

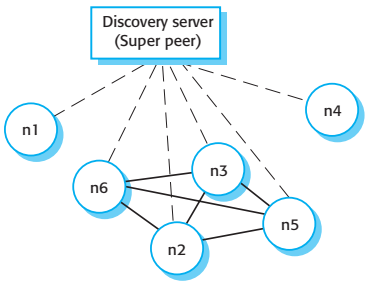
51

A decentralized p2p architecture



52

A semicentralized p2p architecture



53

Use of p2p architecture

- When a system is computationally-intensive and it is possible to separate the processing required into a large number of independent computations (e.g. a drug cancer discovery program that analyses a huge number of molecules; each molecule can be considered separately without the need for the peers to communicate).
- When a system primarily involves the exchange of information between individual computers on a network and there is no need for this information to be centrally-stored or managed (e.g. file sharing systems that allow peers to exchange local files such as music and video or phone systems that support voice and video).

54

Security issues in p2p system

- Security concerns are the principal reason why p2p architectures are not widely used.
- The lack of central management means that malicious nodes can be set up to deliver spam and malware to other nodes in the network.
- P2P communications require careful setup to protect local information and if not done correctly, then this is exposed to other peers.

55

Software as a service

56

Software as a service

- Software as a service (SaaS) involves hosting the software remotely and providing access to it over the Internet.
- This contrasts with the typical client-server model, where the customer “owns” the software application and it is his responsibility to purchase and manage a server on which this application will run (either in a thin- or fat-client fashion).
- As an example, compare a client-server email application such as Outlook or Mac Mail with a SaaS email application such as Yahoo or Gmail.

57

Key elements of SaaS

- Software is deployed on a server (or more commonly a number of servers) and is accessed through a web browser. It is not deployed on a local PC.
- The software is owned and managed by a software provider, rather than the organizations using the software.
- Users may pay for the software according to the amount of use they make of it or through an annual or monthly subscription. Sometimes, the software is free for anyone to use but users must then agree to accept advertisements, which fund the software service.

58

SaaS and SOA

- Software as a service is a way of providing functionality on a remote server with client access through a web browser. The server maintains the user’s data and state during an interaction session. Transactions are usually long transactions e.g. editing a document.
- Service-oriented architecture is an approach to structuring a software system as a set of separate, stateless services. These may be provided by multiple providers and may be distributed. Typically, transactions are short transactions where a service is called, does something then returns a result.

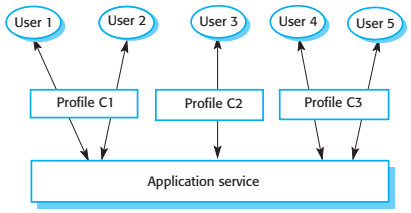
59

Implementation factors for SaaS

- *Configurability* How do you configure the software for the specific requirements of each organization?
- *Multi-tenancy* How do you present each user of the software with 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?

60

Configuration of a software system offered as a service



61

Service configuration

- *Branding*, where users from each organization, are presented with an interface that reflects their own organization.
- *Business rules and workflows*, where each organization defines its own rules that govern the use of the service and its data.
- *Database extensions*, where each organization defines how the generic service data model is extended to meet its specific needs.
- *Access control*, where service customers create individual accounts for their staff and define the resources and functions that are accessible to each of their users.

62

Multi-tenancy

- Multi-tenancy is a situation in which many different users access the same system and the system architecture is defined to allow the efficient sharing of system resources.
- It must appear to each user that they have the sole use of the system.
- Multi-tenancy involves designing the system so that there is an absolute separation between the system functionality and the system data.

63

A multitenant 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

64

Scalability

- Develop applications where each component is implemented as a simple stateless service that may be run on any server.
- Design the system using asynchronous interaction so that the application does not have to wait for the result of an interaction (such as a read request).
- 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 in the database when only part of a record is in use.

65

Key points

- The benefits of distributed systems are that they can be scaled to cope with increasing demand, 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 include transparency, openness, scalability, security, quality of service and failure management.
- 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.

66

Key points

- Architectural patterns for distributed systems include master-slave architectures, two-tier and multi-tier client-server architectures, distributed component architectures and peer-to-peer architectures.
- Distributed component systems require middleware to handle component communications and to allow components to be added to and removed from the system.
- Peer-to-peer architectures are decentralized with no distinguished clients and servers. Computations can be distributed over many systems in different organizations.
- Software as a service is a way of deploying applications as thin client- server systems, where the client is a web browser.