ECE 651 – Software Engineering
Week Twelve – Evolution, Distributed and SOA

April 6th, 2016

Ric Telford
Adjunct Associate Professor

Founder, Telford Ventures

Duke | PRATT SCHOOL OF ENGINEERING

04/06/16 – Week 12 Overview

- Recap / Announcements
- Lecture – Software Evolution
- Break
- Lecture – Distributed Systems
- Questions from Last Week
- About Week 13

Recap

- Project Management
  - Concerned with activities involved in ensuring that software is delivered on time and on schedule and in accordance with the requirements of the organizations developing it.
  - Need to balance *Cost, Schedule and Function*
  - There *are 5 universal project management activities*
  - *Risk Management* is one of those 5 and is concerned with identifying risk and drawing up plans to minimize their effect on a project.
    - Two dimensions of risk classification - type and affect
    - Process consists of identification, analysis, planning and monitoring
- Project Planning
  - Involves breaking down the work into parts and assigning to team members. The project plan is the key document for communication.
  - *Plan-driven development* is a software engineering approach where the process is planned in detail
  - *Project scheduling* is the process of deciding how the work in a project will be organized as separate tasks.
  - In *Agile Planning*, planning is done in increments based on progress and customer input and consists of *Release planning* and *Iteration Planning.*
  - *Estimation techniques* are used in planning to make effort and cost estimates. These are either based on Experience or Algorithms.

Announcements

- You are invited to attend the DUhatch Showcase at 7pm, April 14th, Schiciano
- For anyone that has not earned your full 5 points of **extra credit**, you can do that by attending, BUT you MUST stay for all presentations (first hour) and **sign-in and sign-out** with me
- Free food!

DUhatch COMPANY SHOWCASE Spring 2016

DUhatch Business Incubator @ the Pratt School of Engineering,

Welcomes you to the DUhatch Company Showcase- Spring 2016!

The DUhatch Company Showcase is a biannual event organized by DUhatch, the student business incubator at Duke University, to showcase the work being done by the current batch of student companies, who have start-ups in areas as diverse as embedded sensor technology, nutrition, image processing and so on. Guests will include executives, entrepreneurs from RTP and the Duke Entrepreneurship network. There'll be great people to meet and some amazing free food!

WHEN: 14 APRIL, THURSDAY

WHERE: SCHICIANO AUDITORIUM from 7 PM-9 PM

SEE YOU THERE!

Chap 9 – Software Evolution -Topics covered

- 9.0
- 9.1 - Evolution processes
- 9.2 - Legacy systems
- 9.3 - Software maintenance

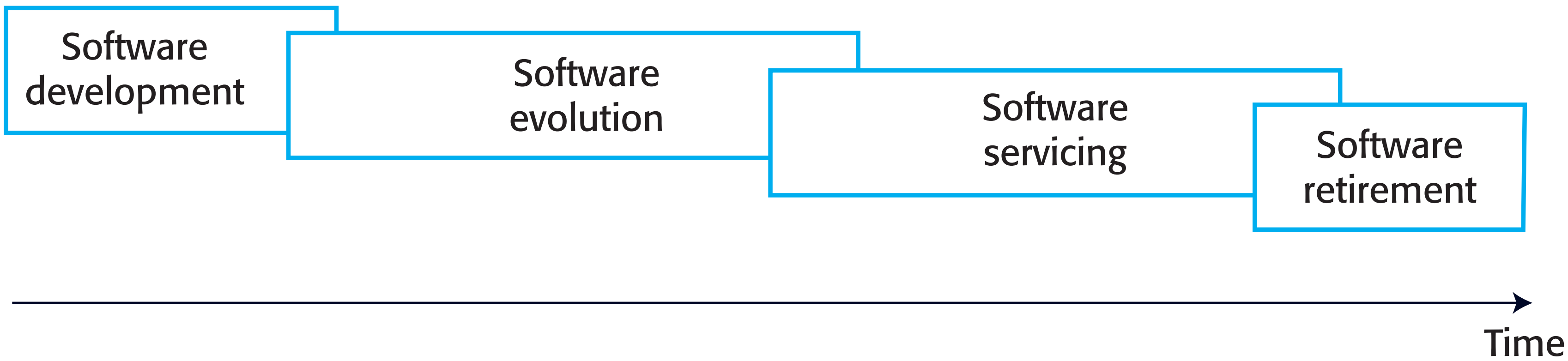
Software change

- Software change is inevitable
  - New requirements emerge when the software is used;
  - The business environment changes;
  - Errors must be repaired;
  - New computers and equipment is added to the system;
  - The performance or reliability of the system may have to be improved.
- A key problem for all organizations is implementing and managing change to their existing software systems.

Importance of evolution

- Organizations have huge investments in their software systems - they are critical business assets.
- To maintain the value of these assets to the business, they must be changed and updated.
- The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.

Evolution and servicing



Evolution and servicing

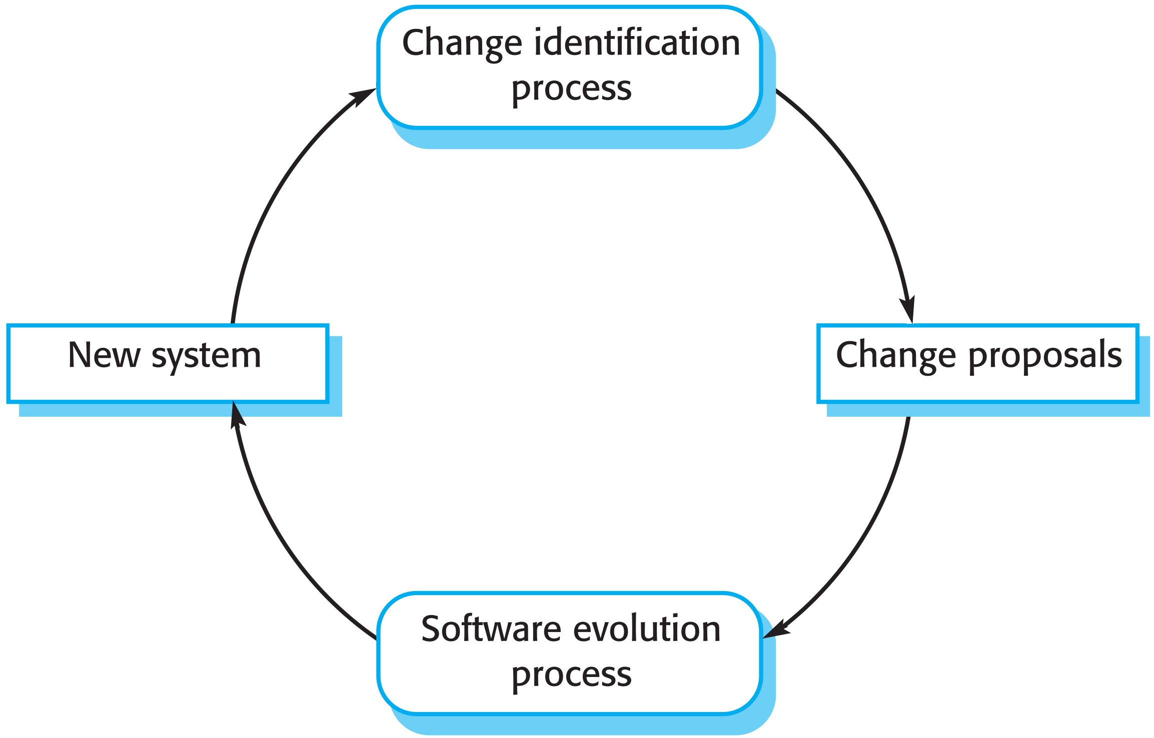
- *Evolution*
  - The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.
- *Servicing*
  - At this stage, the software remains useful but the only changes made are those required to keep it operational i.e. bug fixes and changes to reflect changes in the software's environment. No new functionality is added.
- Phase-out
  - The software may still be used but no further changes are made to it.

Evolution processes

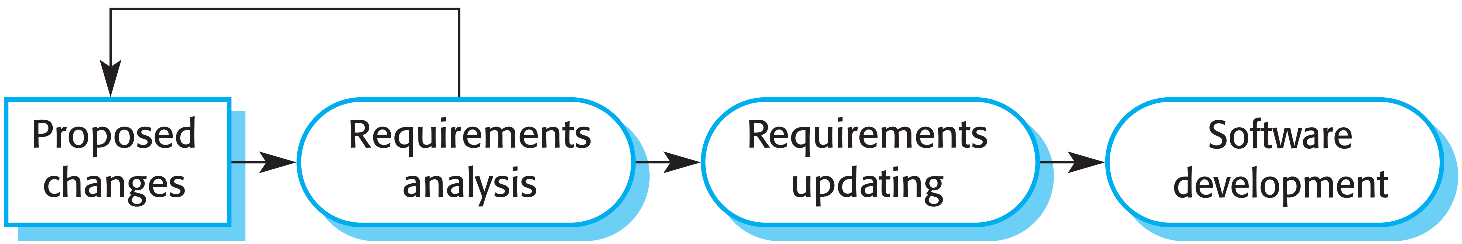
Evolution processes

- Software evolution processes depend on
  - The type of software being maintained;
  - The development processes used;
  - The skills and experience of the people involved.
- Proposals for change are the driver for system evolution.
  - Should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.
- Change identification and evolution continues throughout the system lifetime.

Change identification and evolution processes



Change implementation

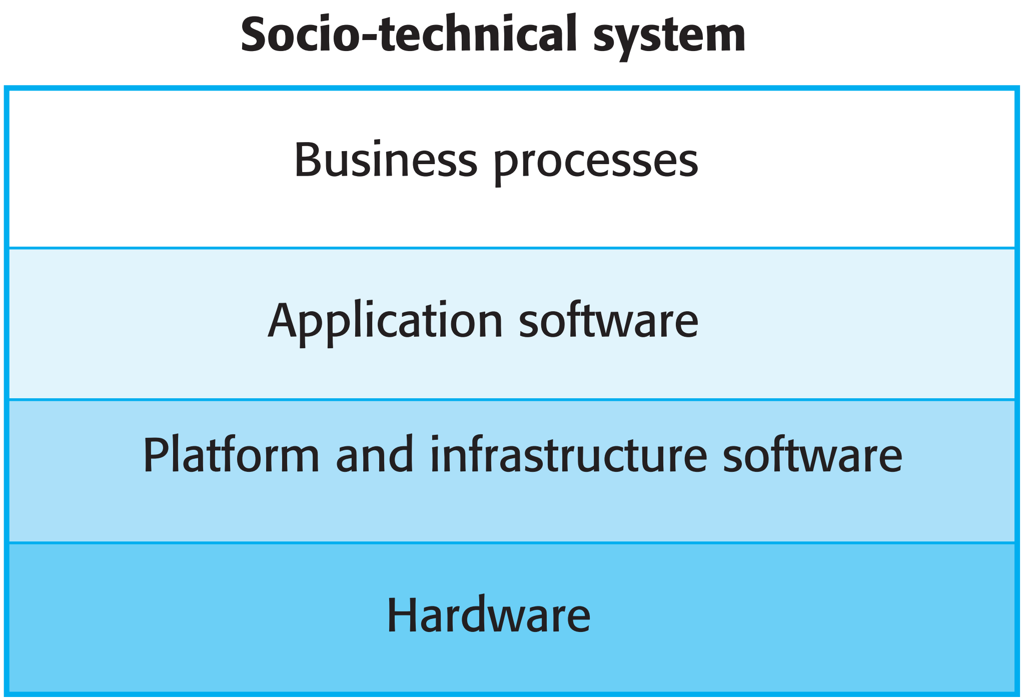


Legacy systems

Legacy systems

- *Legacy systems* are older systems that rely on languages and technology that are no longer used for new systems development.
- Legacy software may be dependent on older hardware, such as mainframe computers and may have associated legacy processes and procedures.
- Legacy systems are not just software systems but are broader socio-technical systems that include hardware, software, libraries and other supporting software and business processes.

Legacy system layers



Legacy system replacement

- Legacy system replacement is risky and expensive so businesses continue to use these systems
- System replacement is risky for a number of reasons
  - Lack of complete system specification
  - Tight integration of system and business processes
  - Undocumented business rules embedded in the legacy system
  - New software development may be late and/or over budget

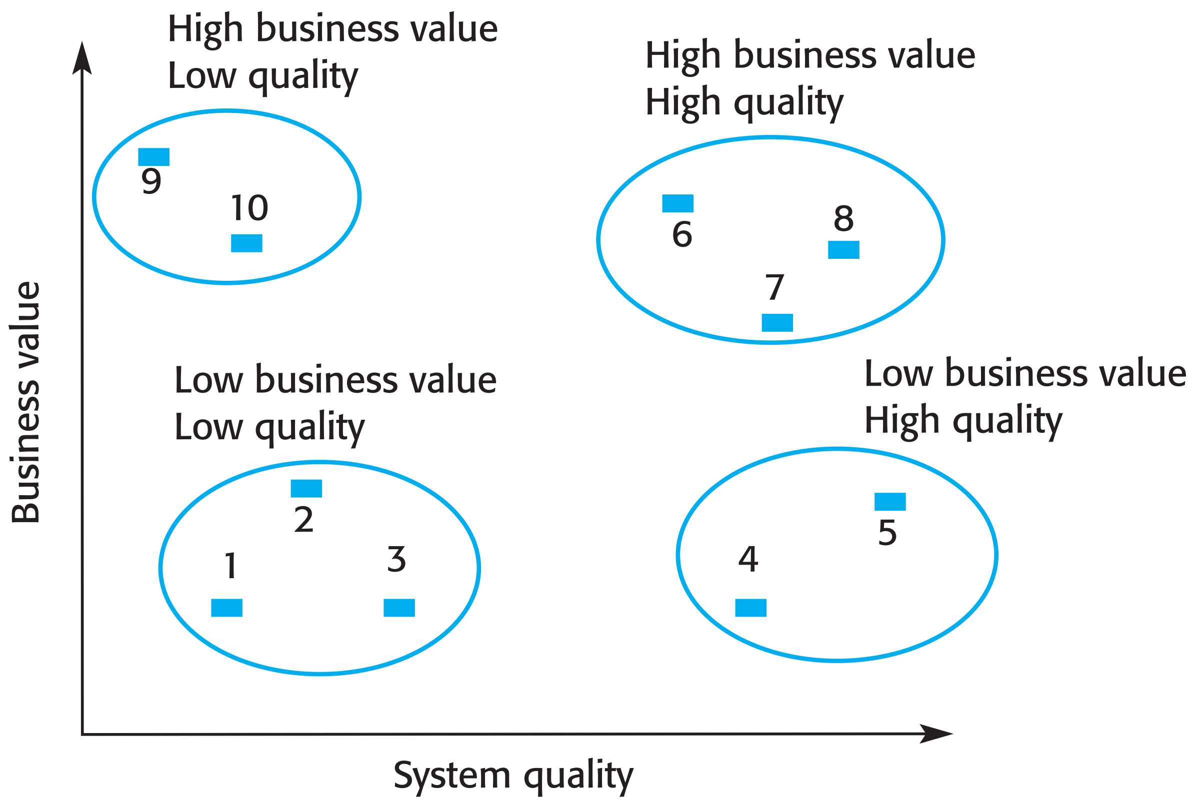
Legacy system change

- Legacy systems are expensive to change for a number of reasons:
  - No consistent programming style
  - Use of obsolete programming languages with few people available with these language skills
  - Inadequate system documentation
  - System structure degradation
  - Program optimizations may make them hard to understand
  - Data errors, duplication and inconsistency

Legacy system management

- Organizations that rely on legacy systems must choose a strategy for evolving these systems
  - Scrap the system completely and modify business processes so that it is no longer required;
  - Continue maintaining the system;
  - Transform the system by re-engineering to improve its maintainability;
  - Replace the system with a new system.
- The strategy chosen should depend on the system quality and its business value.

Figure 9.13 An example of a legacy system assessment



Legacy system categories

- Low quality, low business value
  - These systems should be scrapped.
- Low-quality, high-business value
  - These make an important business contribution but are expensive to maintain. Should be re-engineered or replaced if a suitable system is available.
- High-quality, low-business value
  - Replace with COTS, scrap completely or maintain.
- High-quality, high business value
  - Continue in operation using normal system maintenance.

Software maintenance

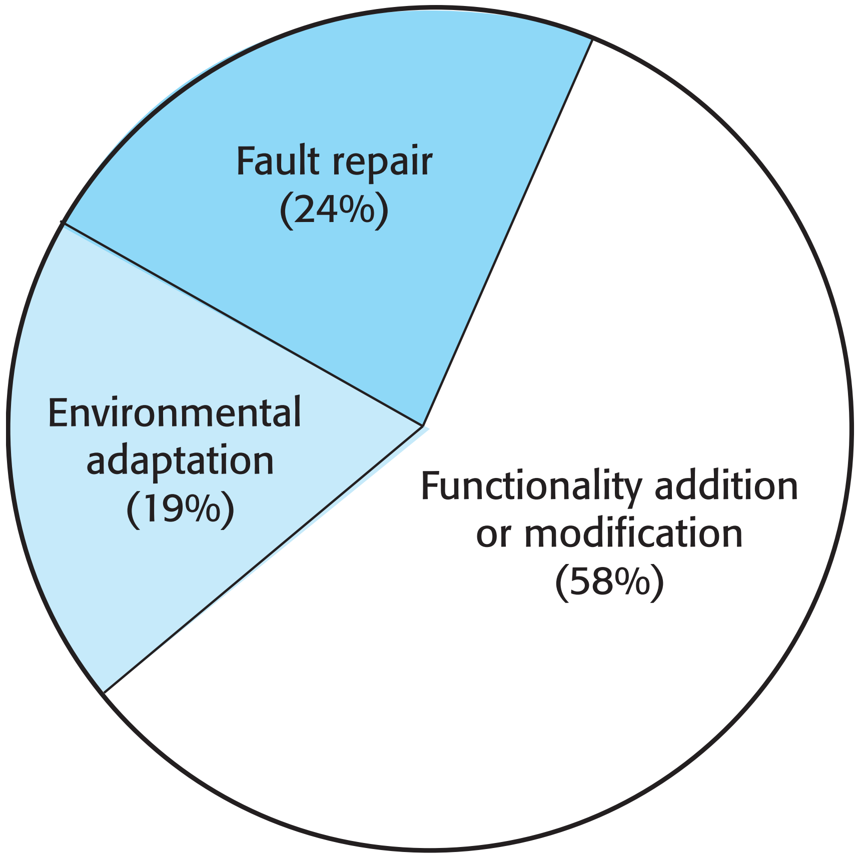
Software maintenance

- Modifying a program after it has been put into use.
- The term is mostly used for changing custom software. Generic software products are said to evolve to create new versions.
- Maintenance does not normally involve major changes to the system's architecture.
- Changes are implemented by modifying existing components and adding new components to the system.

Types of maintenance

- *Fault repairs*
  - Changing a system to fix bugs/vulnerabilities and correct deficiencies in the way meets its requirements.
- *Environmental adaptation*
  - Maintenance to adapt software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- *Functionality addition* and modification
  - Modifying the system to satisfy new requirements.

Maintenance effort distribution



Maintenance costs

- Usually greater than development costs (2* to 100* depending on the application).
- Affected by both technical and non-technical factors.
- Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.
- Ageing software can have high support costs (e.g. old languages, compilers etc.).

Maintenance costs

- It is usually more expensive to add new features to a system during maintenance than it is to add the same features during development
  - A new team has to understand the programs being maintained
  - Separating maintenance and development means there is no incentive for the development team to write maintainable software
  - Program maintenance work is unpopular
    - Maintenance staff are often inexperienced and have limited domain knowledge.
  - As programs age, their structure degrades and they become harder to change

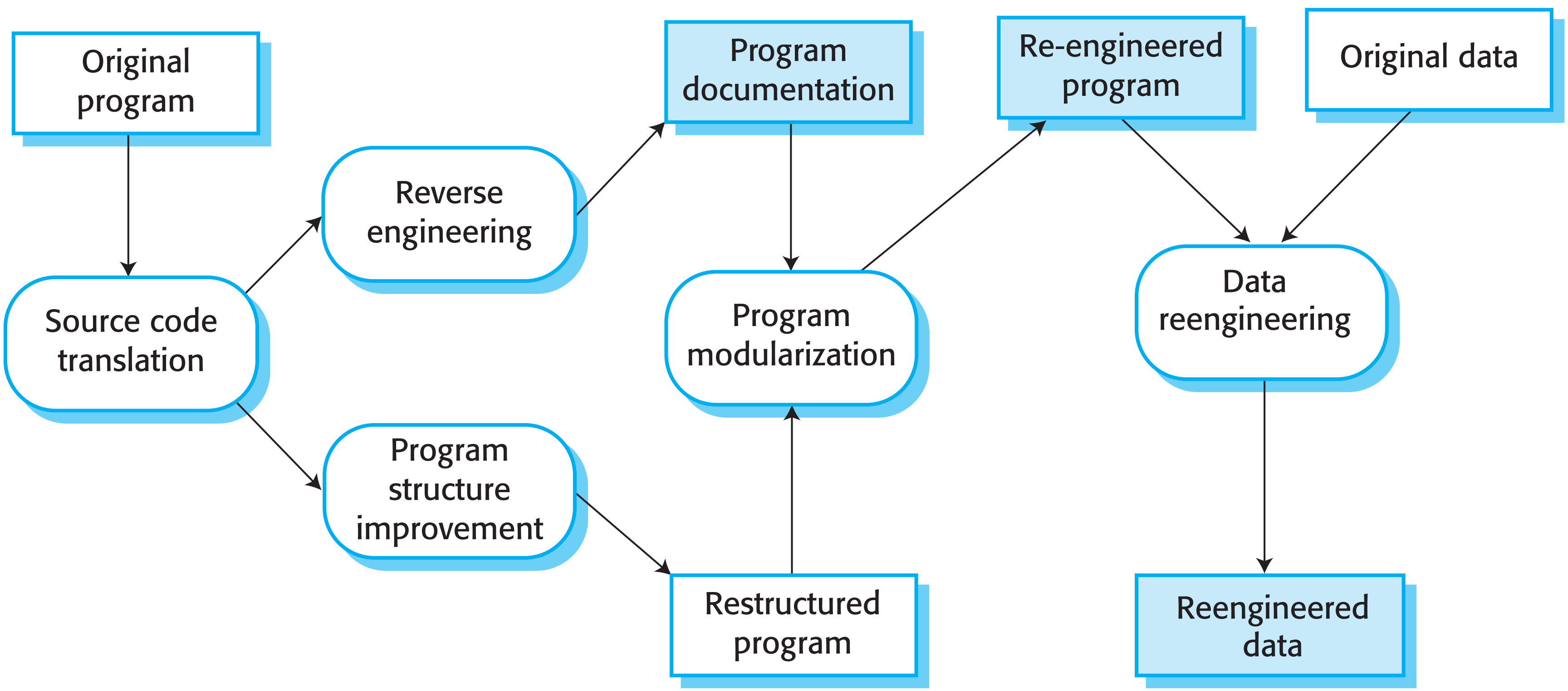
Software reengineering

- Restructuring or rewriting part or all of a legacy system without changing its functionality.
- Applicable where some but not all sub-systems of a larger system require frequent maintenance.
- Reengineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.

Advantages of reengineering

- Reduced risk
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- Reduced cost
  - The cost of re-engineering is often significantly less than the costs of developing new software.

The reengineering process



Reengineering process activities

- Source code translation
  - Convert code to a new language.
- Reverse engineering
  - Analyse the program to understand it;
- Program structure improvement
  - Restructure automatically for understandability;
- Program modularisation
  - Reorganise the program structure;
- Data reengineering
  - Clean-up and restructure system data.

Refactoring

- *Refactoring* is the process of making improvements to a program to slow down degradation through change.
- You can think of refactoring as ‘preventative maintenance’ that reduces the problems of future change.
- Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
- When you refactor a program, you should not add functionality but rather concentrate on program improvement.

Refactoring and reengineering

- *Re-engineering* takes place after a system has been maintained for some time and maintenance costs are increasing. You use automated tools to process and re-engineer a legacy system to create a new system that is more maintainable.
- *Refactoring* is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

‘Bad smells’ in program code

- Duplicate code
  - The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.
- Long methods
  - If a method is too long, it should be redesigned as a number of shorter methods.
- Switch (case) statements
  - These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.

‘Bad smells’ in program code

- Data clumping
  - Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. These can often be replaced with an object that encapsulates all of the data.
- Speculative generality
  - This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

Key points

- Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- For custom systems, the costs of software maintenance usually exceed the software development costs.
- The process of software evolution is driven by requests for changes and includes change impact analysis, release planning and change implementation.
- Legacy systems are older software systems, developed using obsolete software and hardware technologies, that remain useful for a business.

Key points

- It is often cheaper and less risky to maintain a legacy system than to develop a replacement system using modern technology.
- The business value of a legacy system and the quality of the application should be assessed to help decide if a system should be replaced, transformed or maintained.
- There are 3 types of software maintenance, namely bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.

Key points

- Software re-engineering is concerned with re-structuring and re-documenting software to make it easier to understand and change.
- Refactoring, making program changes that preserve functionality, is a form of preventative maintenance.

Chap 17 – Distributed Software Engineering - Topics covered

- 17.0
- 17.1 - Distributed systems
- 17.2 - Client–server computing
- 17.3 - Architectural patterns for distributed systems
- 17.4 - Software as a service

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.
- Distributed software engineering is therefore very important for enterprise computing 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.

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.
  - 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.

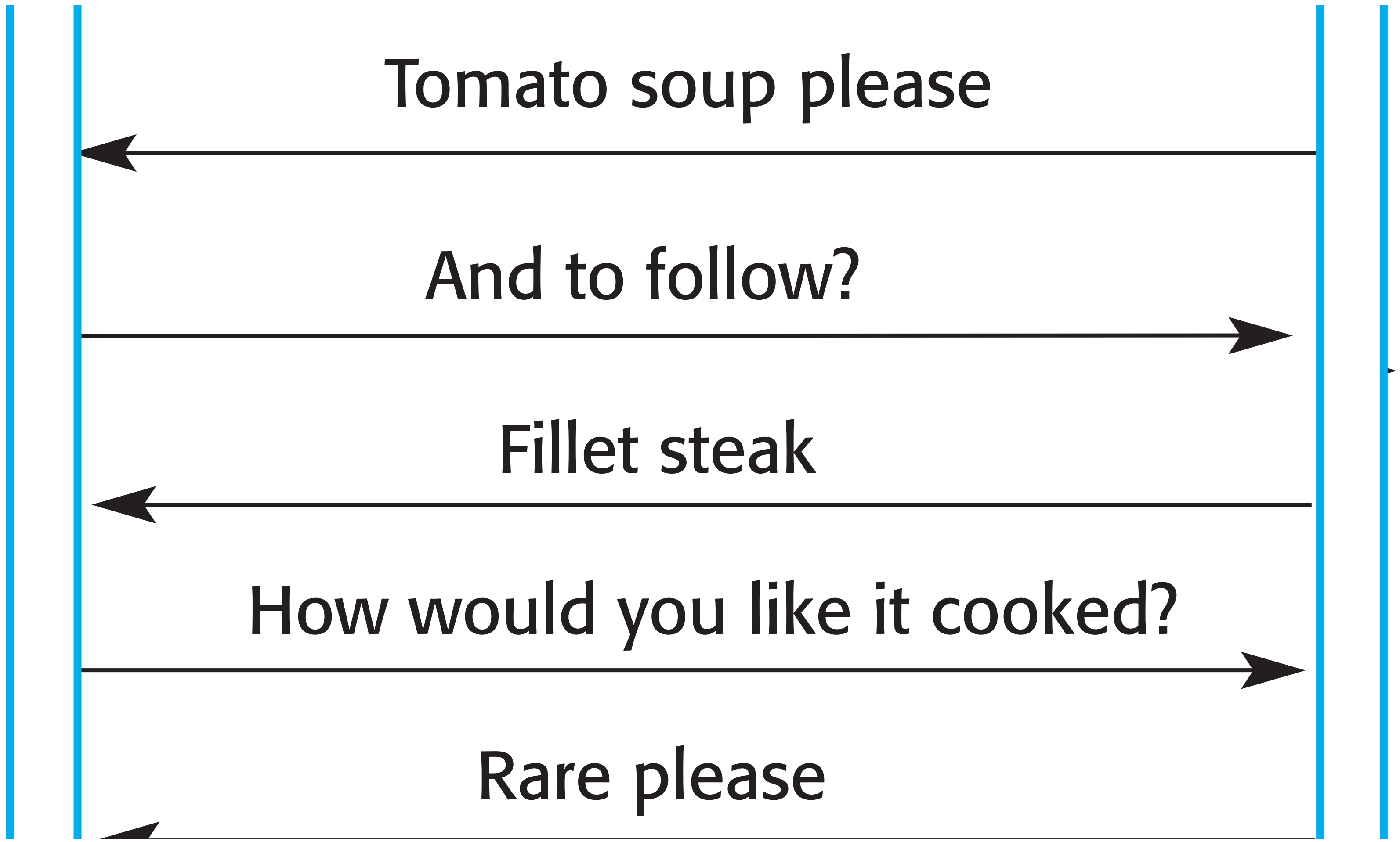
  "You know that you have a distributed system when the crash of a system that you've never heard of stops you getting any work done."

- 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



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>

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.

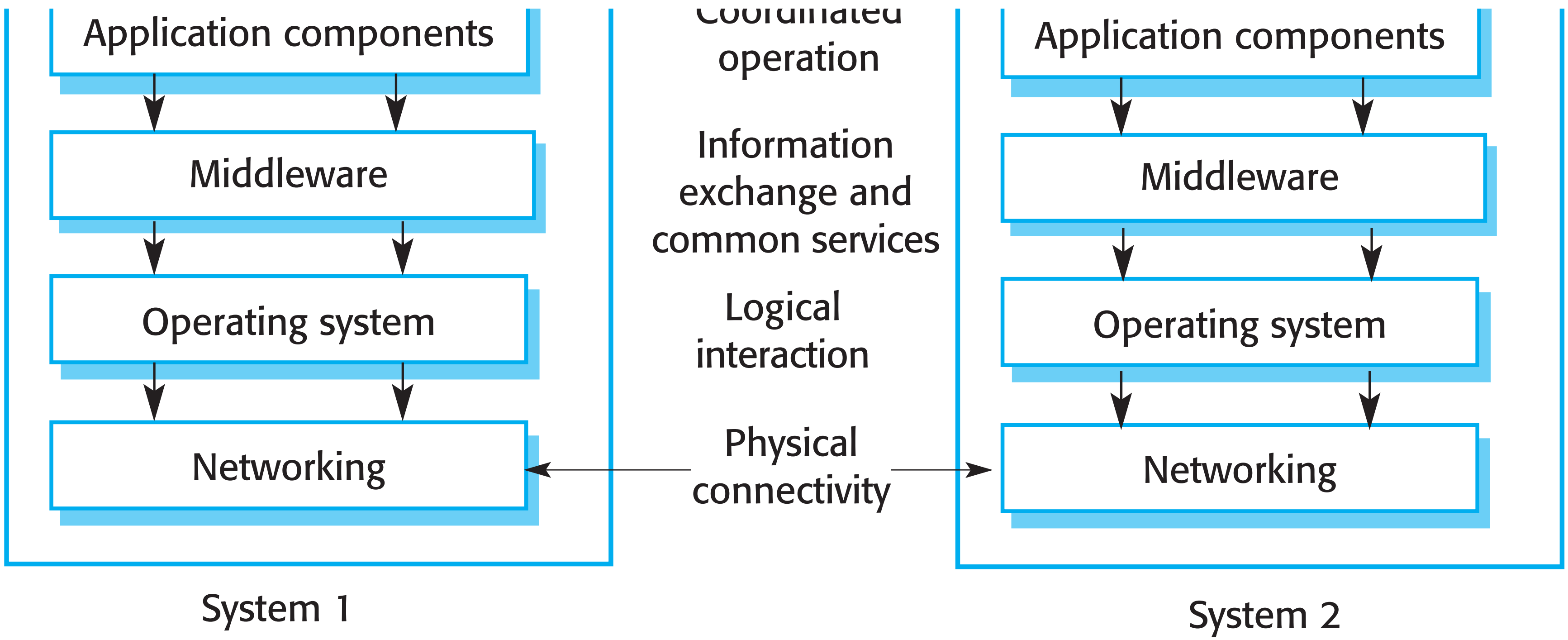
Message passing

- 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

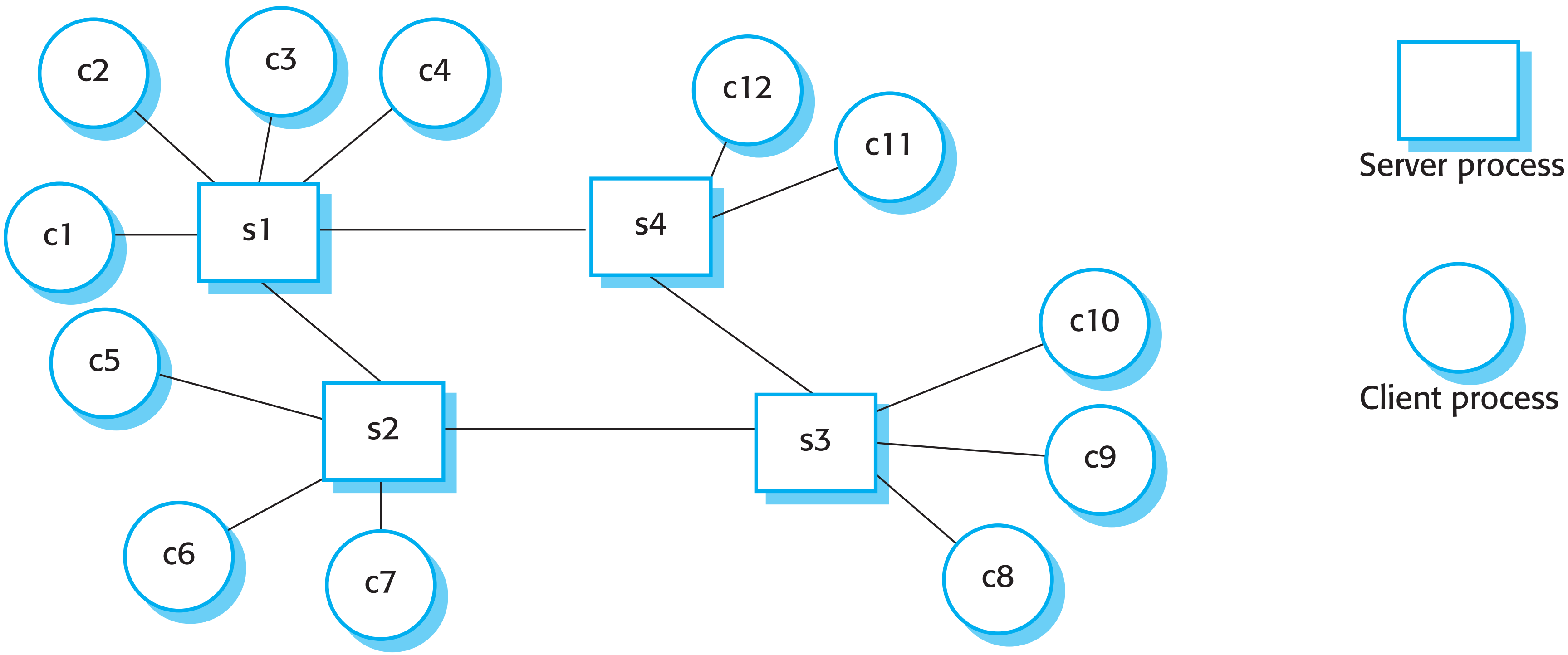


Client-server computing

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



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

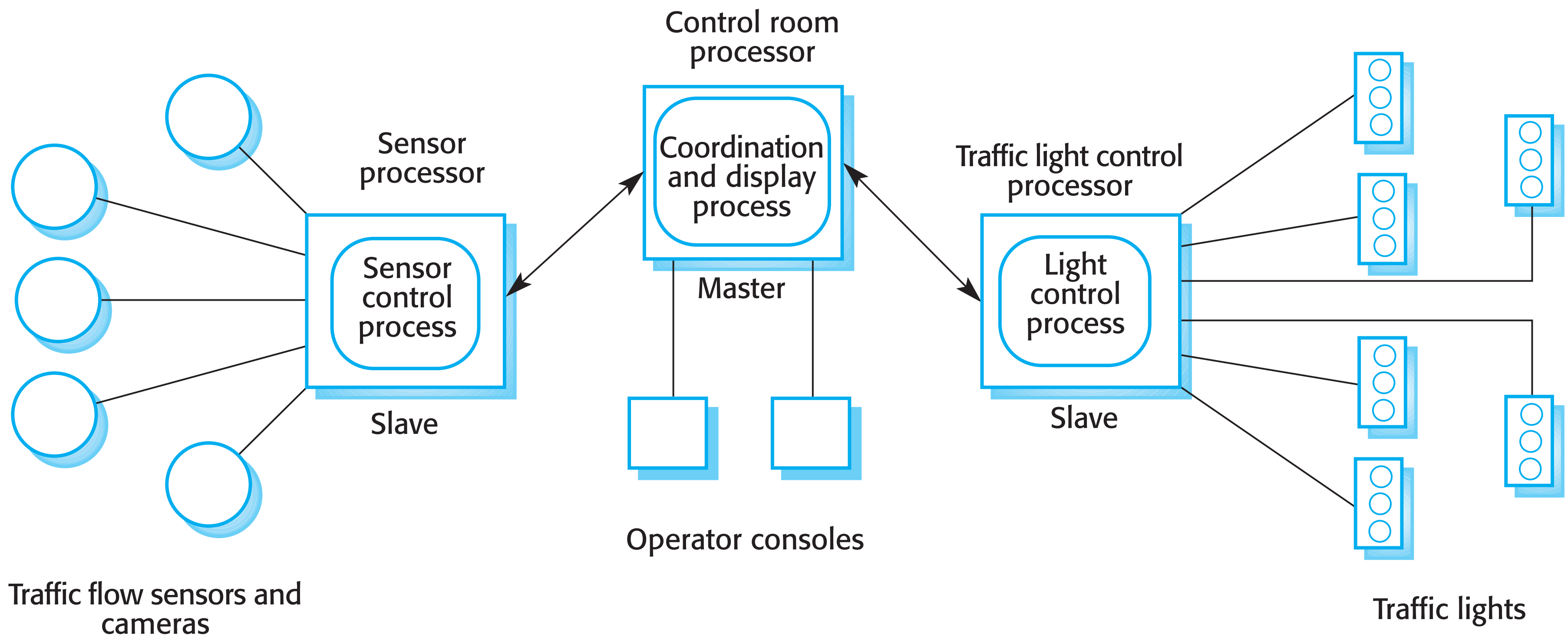
Architectural patterns

- 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.

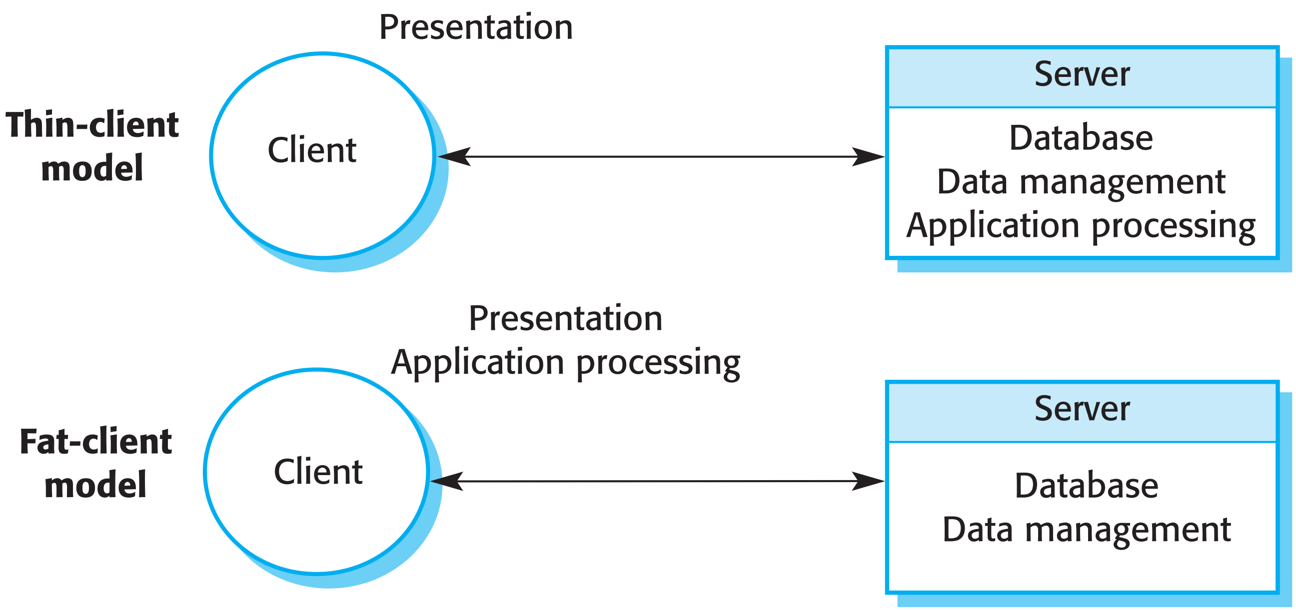
A traffic management system with a master-slave architecture



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



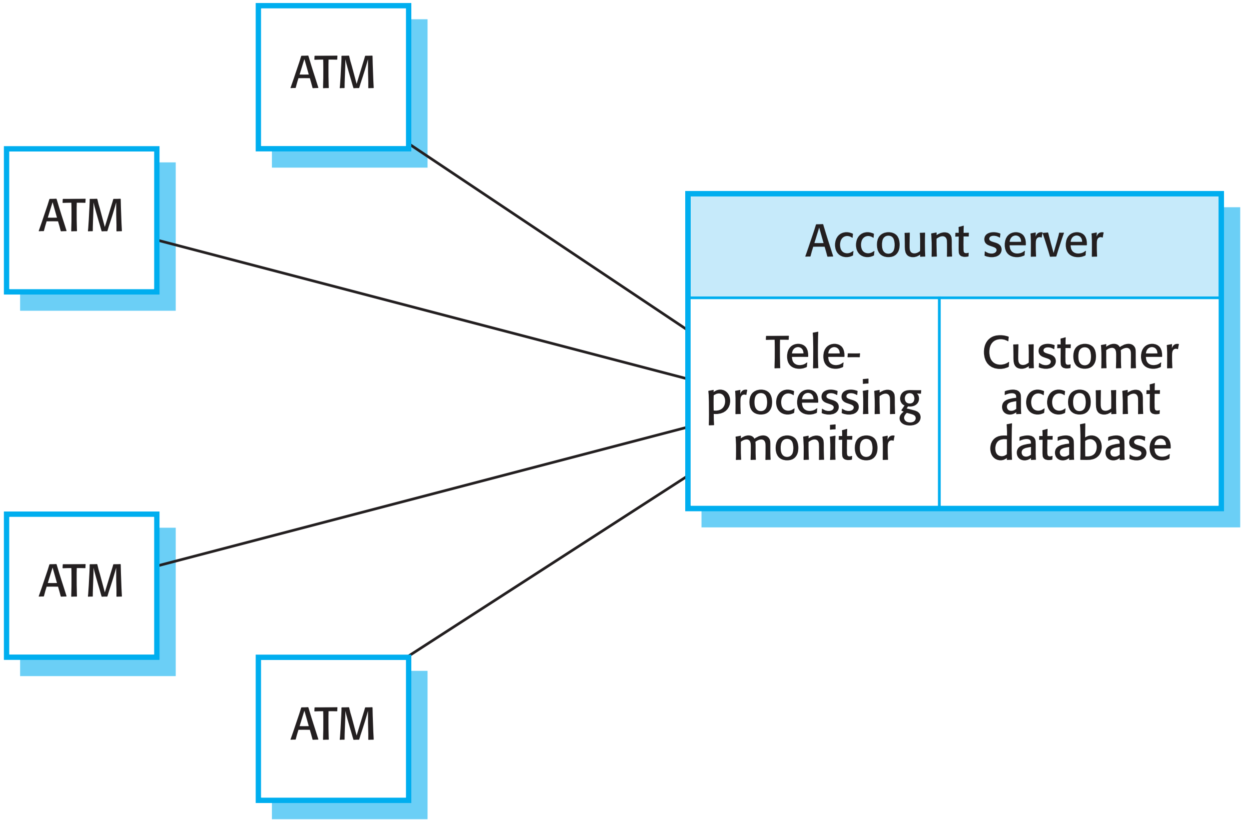
Thin client model

- 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



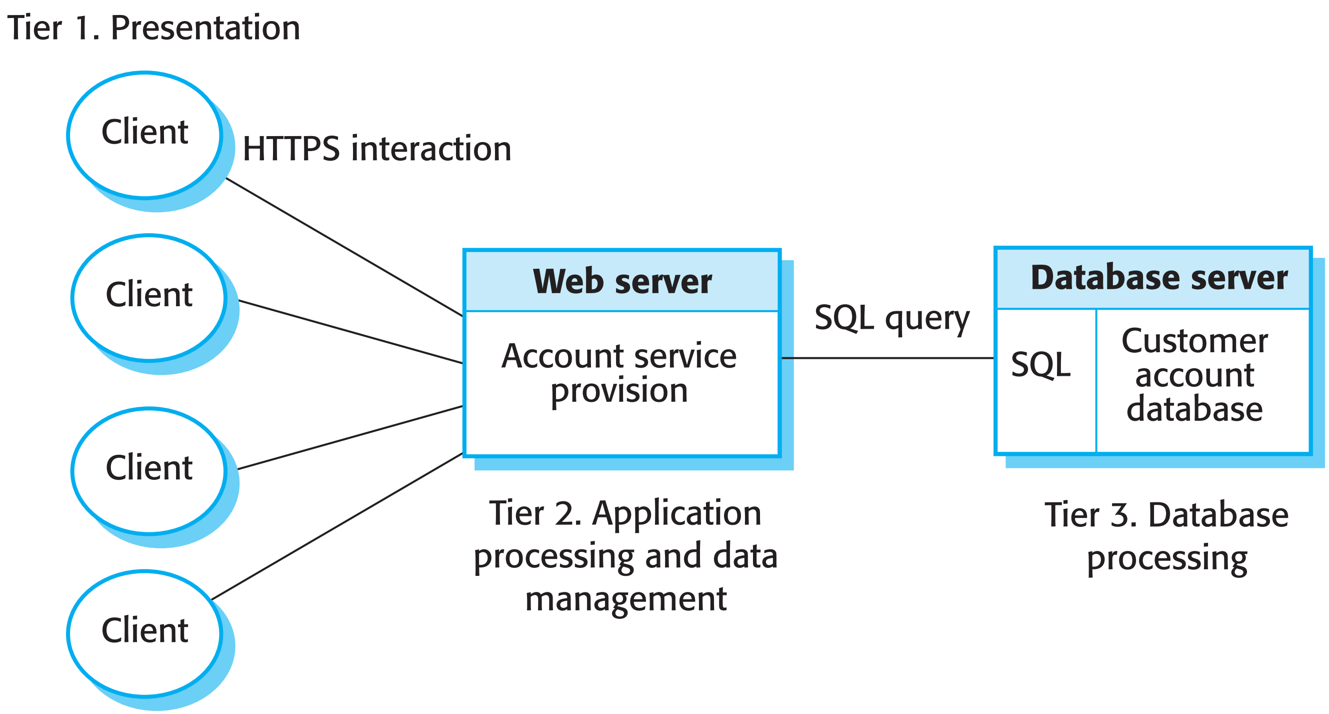
Thin and fat clients

- 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. Computationally intensive applications such as compilers with little or no data management. 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.

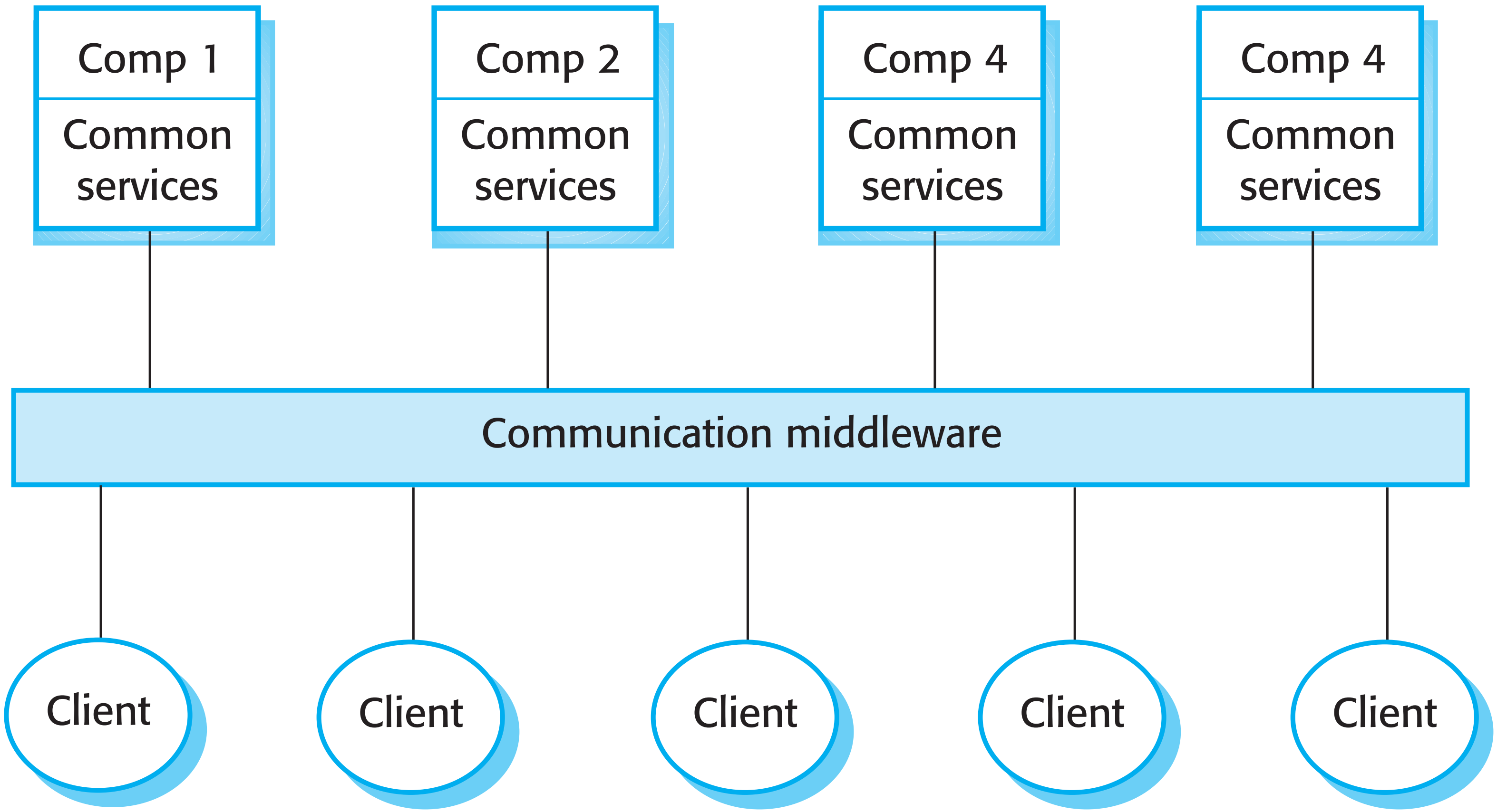
Use of client–server architectural patterns

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. 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.

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.

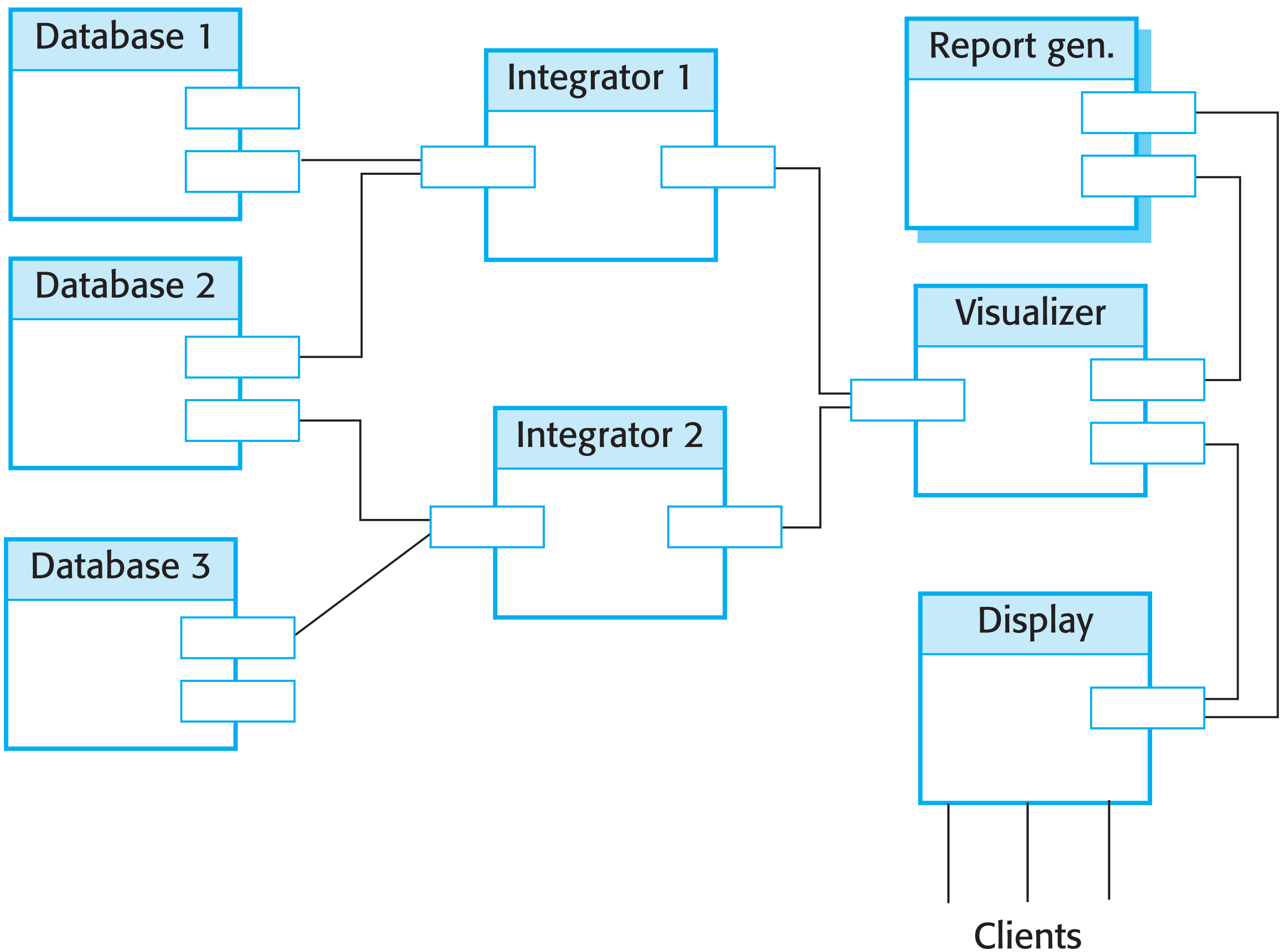
A distributed component architecture



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.

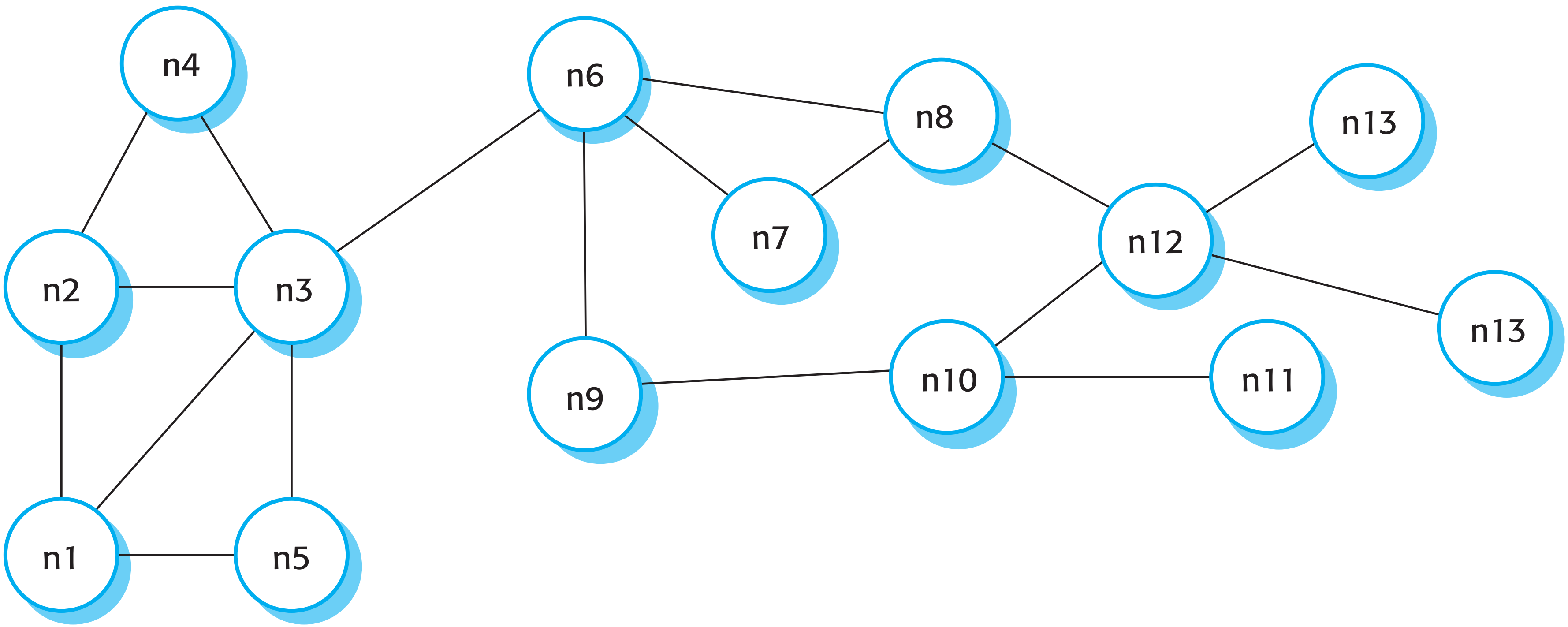
Peer-to-peer systems

- 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

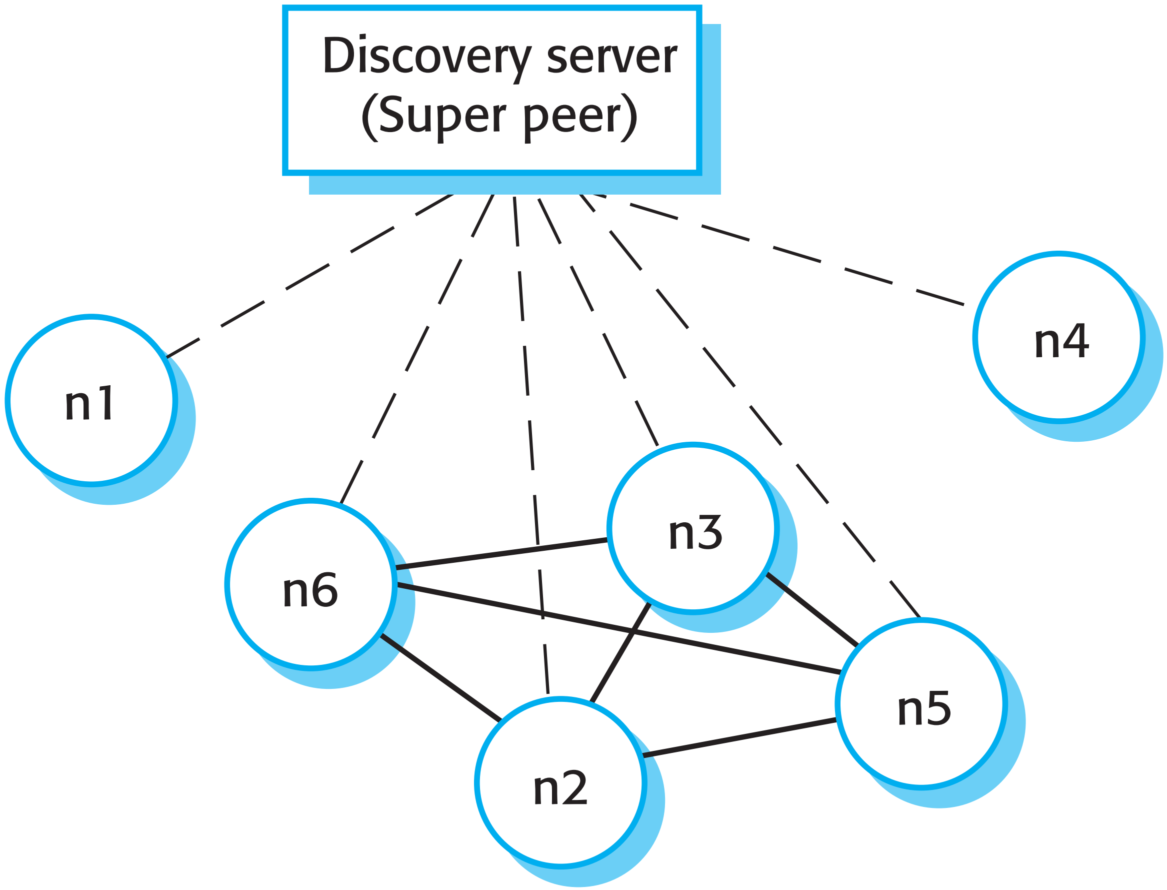
P2p architectural models

- The logical network architecture
  - Decentralised architectures;
  - Semi-centralised architectures.
- Application architecture
  - The generic organisation of components making up a p2p application.
- Focus here on network architectures.

A decentralized p2p architecture



A semicentralized p2p architecture



Software as a service

Software as a service

- Software as a service (SaaS) involves hosting the software remotely and providing access to it over the Internet.
  - 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.

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?

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.

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.

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

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.

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.

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.

About Week 13 – April 13th

- Topics will be SOA, Web and Mobile
- HW 8 – Checkpoint Meetings.
  - Schedule second checkpoint with TA – DUE April 15th
  - No need to respond in Sakai – we will track and grade.
- Finish up your Documentation Deliverable
- Soon you should be starting User Testing / System Testing