SOFTWARE DESIGN (YAZILIM TASARIMI)

Week 5

Principles of Design

- *Design emerges from a need to solve a problem.*
- *Design is a creative process to meet the need to solve the problem.*
- *Design has two phases:*
  - *Inspiration phase*
  - *Expression or communication phase.*
- *Design is a bottom-up process.*

Design: Inspiration Phase

- *The original (i.e., inspired) idea or thought is disordered or chaotic.*
- *It is not expressible/communicatible as inspired.*
- *A powerful communication basis is required to express the inspired thought(s) in an organized manner.*

Design: Communication Phase

- *Idea or thought is disorganized as inspired.*
- *Words form a tool to communicate inspired thoughts.*
- *Words are components to express/communicate thoughts in a comprehensible manner.*
- *Designer is not confined in her/his inspirations; but restricted to the expessive power of the words s/he selects to communicate her/his thought(s).*
- *The quality of the communicated thought depends upon the expressive power of the words and the potential of the designer to use words.*

Conclusion from Last Page

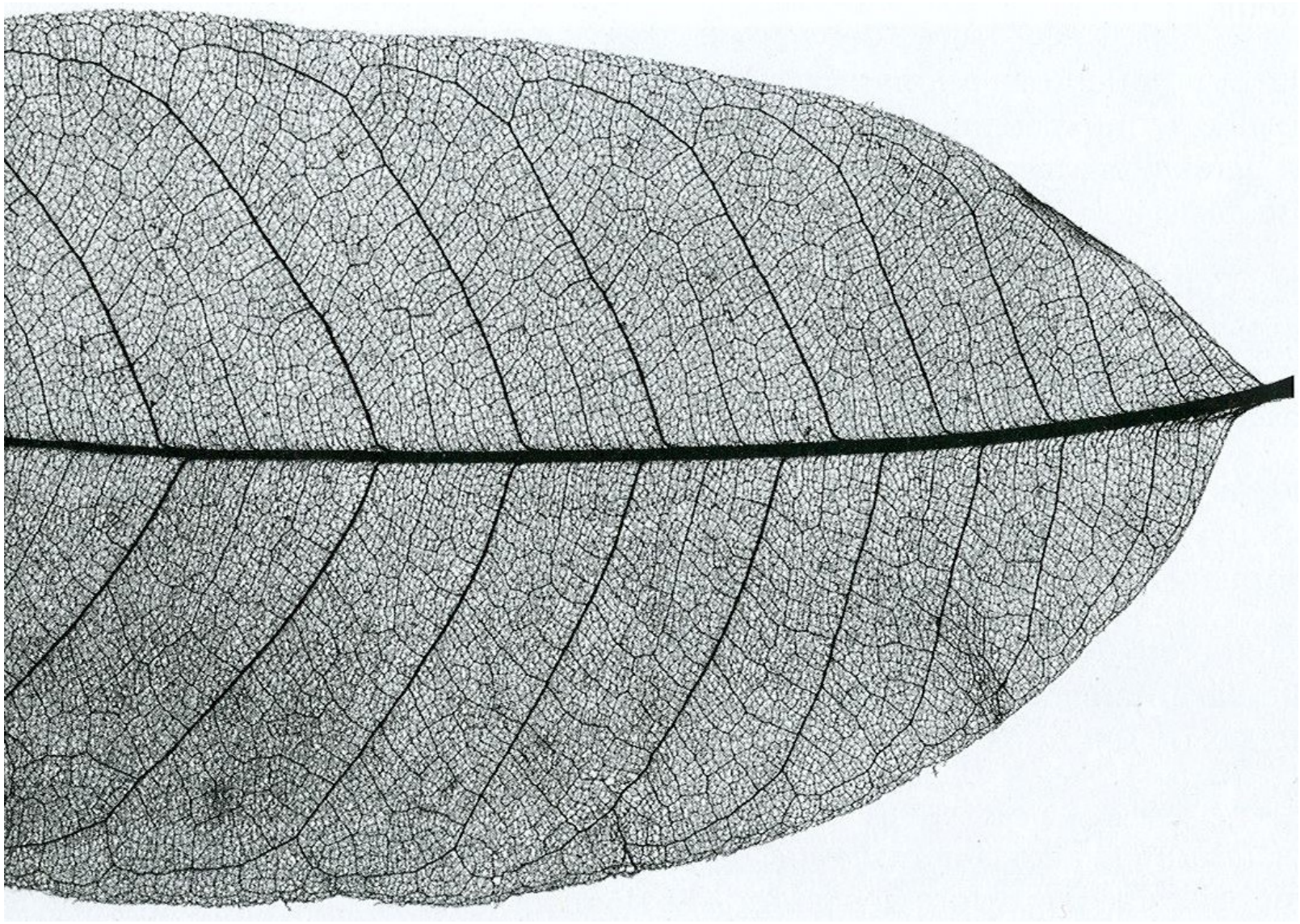
It is inevitable that the designer should have a well-established basis of knowledge on the set of communication tool(s) to use them to their best.

Nature's Tradition of Perfect Design

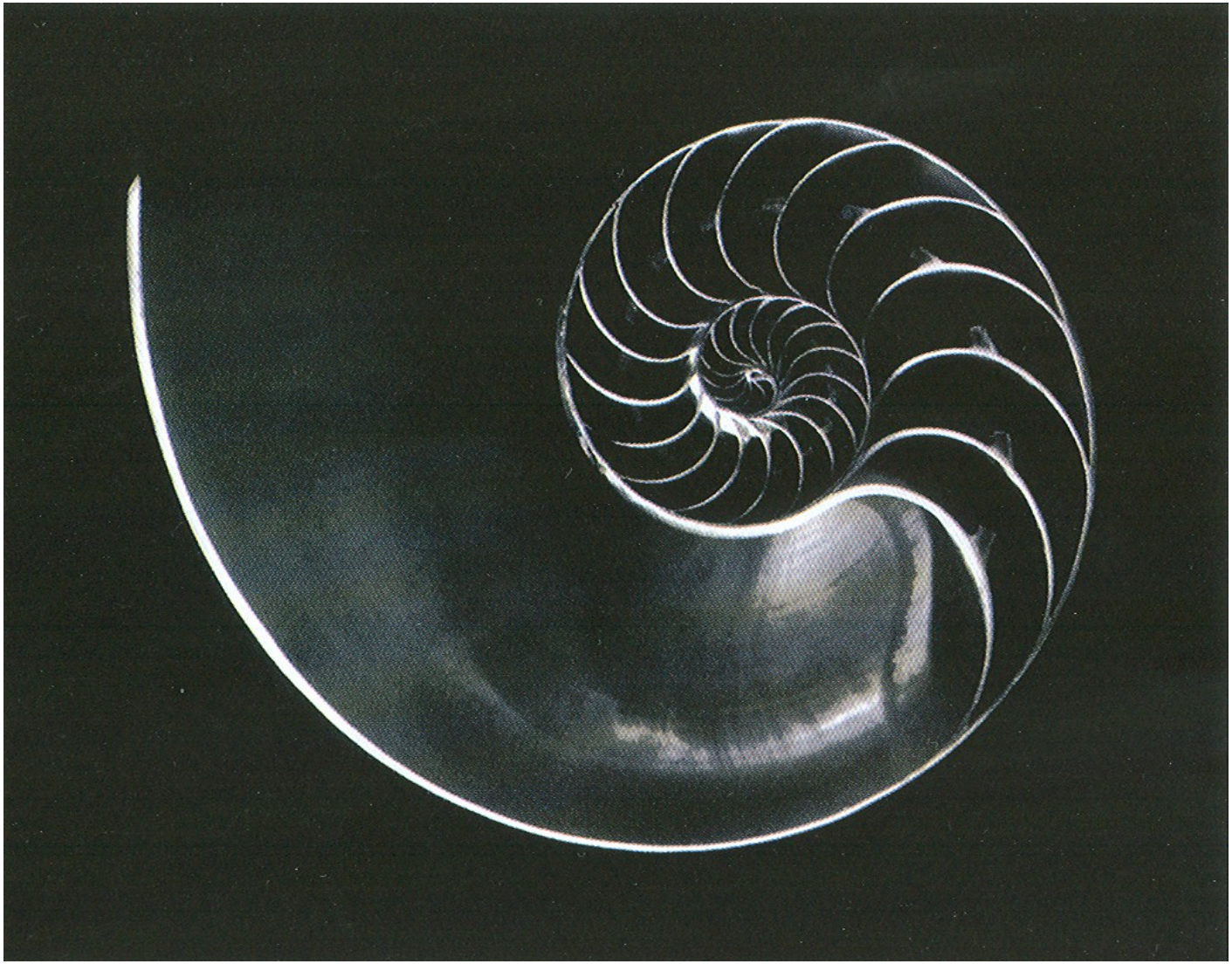
- *Nature designs perfectly since nature bases its designs upon the (to a significant extent unknown) internal reasons originating from an incredible level of balance in an enourmously diverse system of adaptable (evolving) organisms.*

Nature's Multi-Levelled Approach to Design

- *Nature has created the tiger to do whatever it achieves to survive in its environment; through **evolution** nature has excelled its design (i.e., all aspects ranging from its most detailed cellular to most general systemic level) to progressively fit its environment.*
- *The same is valid for a leaf or a snail or us.*



Photography: Andreas Feininger



Photography: Andreas Feininger

Nature Designs in a Bottom-up Fashion!

- *Nature's way of design ranging from the most detailed cellular to most general systemic level to provide an organism with all necessary and sufficient properties to survive its current environment accounts for its **bottom-up** approach to design along with its equipping these organisms with evolvability.*

Means of SW Design

- *UML*
- *Prototyping language (if required)*
- *Formal Language, corresponding IDE*
- *Environment: O/S, hardware, network.*
- *Graphic tools (if necessary)*
- *Others?*

SW Design: A Bottom-up Process

- *Designer is required to establish a sufficient background on all tools employed in any part of SW design.*
- *Establishing the required level of background on tools of use, the designer naturally follows a bottom-up approach to SW design.*

Key Points

- *Look for solutions as simple as possible;*
- *Nature's choice is also for as simple as can be. We appreciate this fact from our everyday experience on living organisms that they have the simplest necessary equipment for the given functionality;*
- *Remember! Among a number of solution alternatives to a problem, the best one is the simplest one! (Occam's razor!);*

Key Points

- *Learn all necessary means of SW design (i.e., those we have considered on page 11) to your best and expand your experience on using them;*
- *Just as one who possesses a large vocabulary and is talented (and/or educated) to use words meaningfully and correctly can more powerfully express her/his thoughts than an ordinary person, a SW engineer well equipped with/experienced on necessary tools is more likely to come up with better SW designs in general.*

Architectural Design

Architectural Design

- ... is the design stage for identifying
  - the *sub-systems* making up a system and
  - the framework for sub-system *control and communication*.

Architectural design ...2

- ... is an *early stage* of the *system design process*.
- ... is a *link between specification and design*.
- ... involves *identifying major system components and their communications*.

Advantages of explicit architecture

- ***Stakeholder communication***
  - a *focus of discussion* by system stakeholders.
- ***System analysis***
  - Architectural design at an early stage requires system analysis. This in turn makes analysis possible of *whether the system can meet its non-functional requirements*.
- ***Large-scale reuse***
  - architecture reusable?

Architecture and system characteristics

- ***Performance***
  - Localise critical operations and minimise communications. Use larger rather than finer components.
- ***Security***
  - Use layered architecture with critical assets in the inner layers.
- ***Safety***
  - Localise safety-critical features in a small number of sub-systems.
- ***Availability***
  - Include redundant components and mechanisms for fault tolerance.
- ***Maintainability***
  - Use finer, replaceable components.

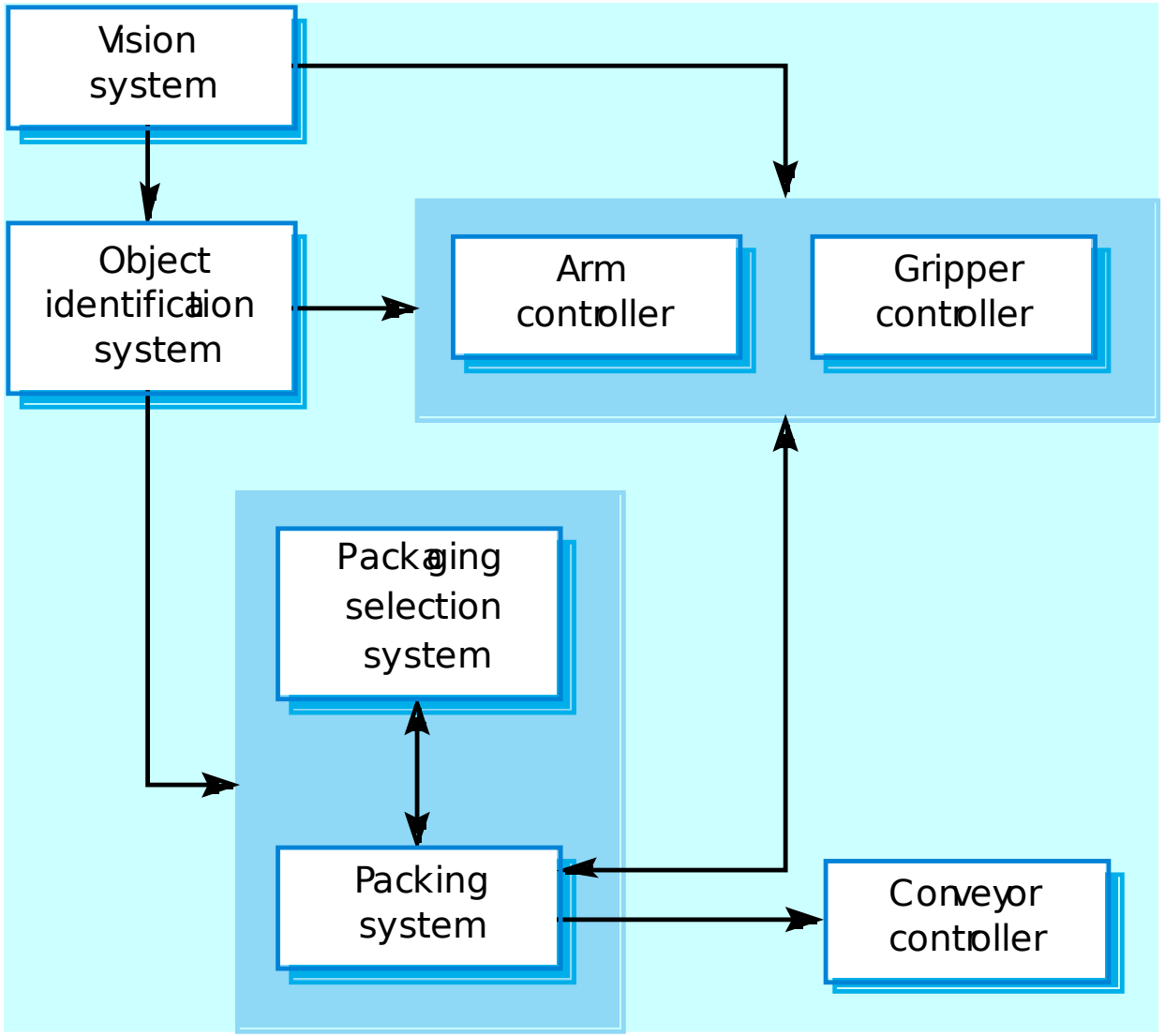
Architectural conflicts

- A ***high performance and maintainable*** system?
- ***Availability*** (i.e., redundant data) and ***security*** (i.e., layered architecture) together?
- A ***safe*** (safety-related features means more communication) and ***high performance*** system?

System structuring

- decomposing the system into *interacting sub-systems*.
- normally expressed as a *block diagram* presenting an overview of the system structure.
- showing how sub-systems
  - *share data*,
  - *are distributed* and
  - *interface with each other*.

Packing robot control system



Architectural design decisions

- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- How will the system be decomposed into modules?
- What control strategy should be used?
- How will the architectural design be evaluated?
- How should the architecture be documented?

Architectural models

- ***Static structural model*** that shows the major system components.
- ***Dynamic process model*** that shows the process structure of the system.
- ***Interface model*** that defines sub-system interfaces.
- ***Relationships model*** such as a data-flow model that shows sub-system relationships.
- ***Distribution model*** that shows how sub-systems are distributed across computers.

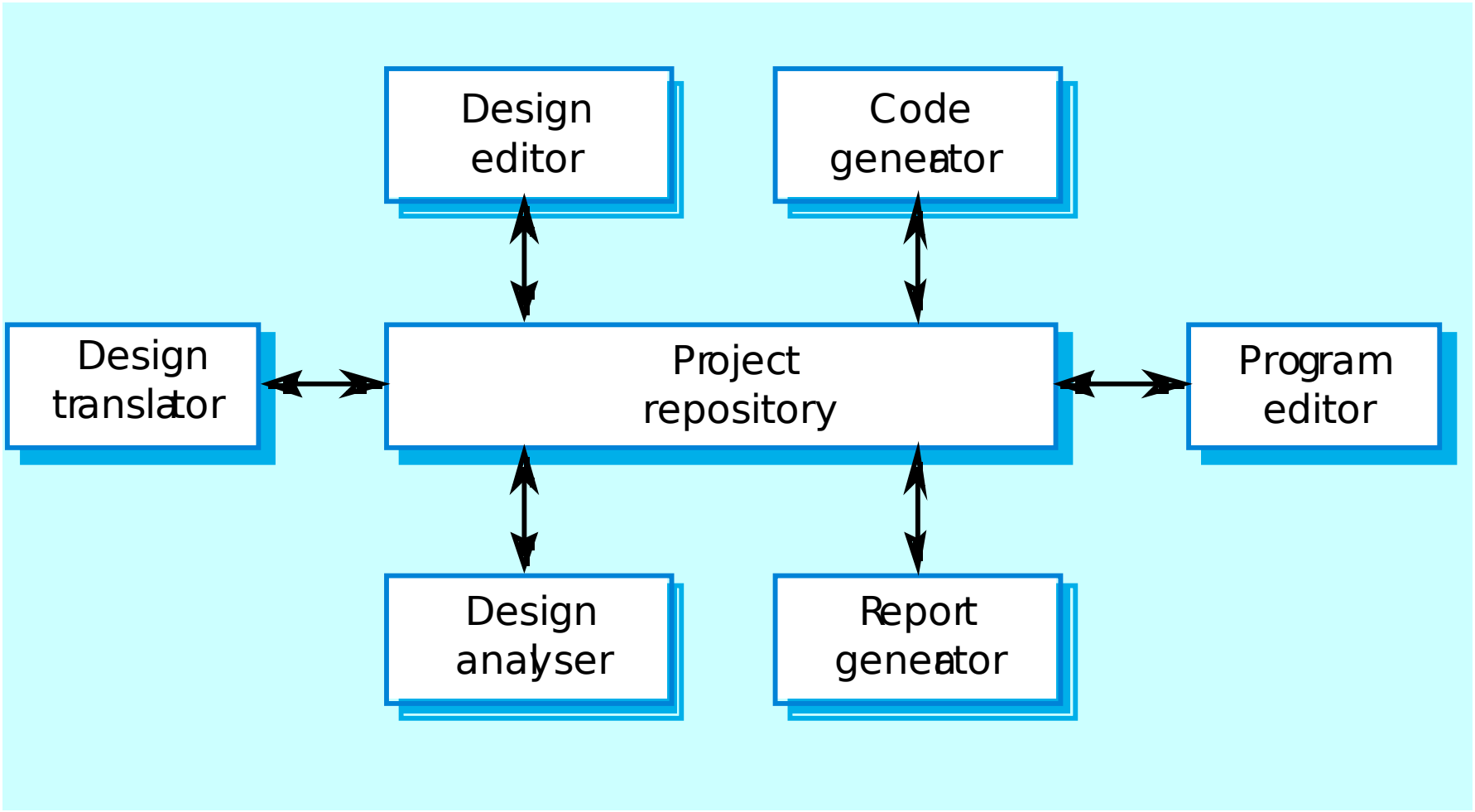
System organisation

- Reflects the basic strategy that is used to structure a system into its subsystems.
- Three organisational styles are widely used:
  - A ***shared data repository*** style;
  - A ***shared services and servers*** style;
  - An ***abstract machine or layered*** style.

The repository model

- Sub-systems exchange data in two ways:
  - ***Shared data is held in a central database or repository*** and may be accessed by all sub-systems;
  - ***Each sub-system maintains its own database and passes data explicitly to other sub-systems***.
- When ***large amounts of data*** are to be shared, the ***repository model*** of sharing is most commonly used.

CASE toolset architecture



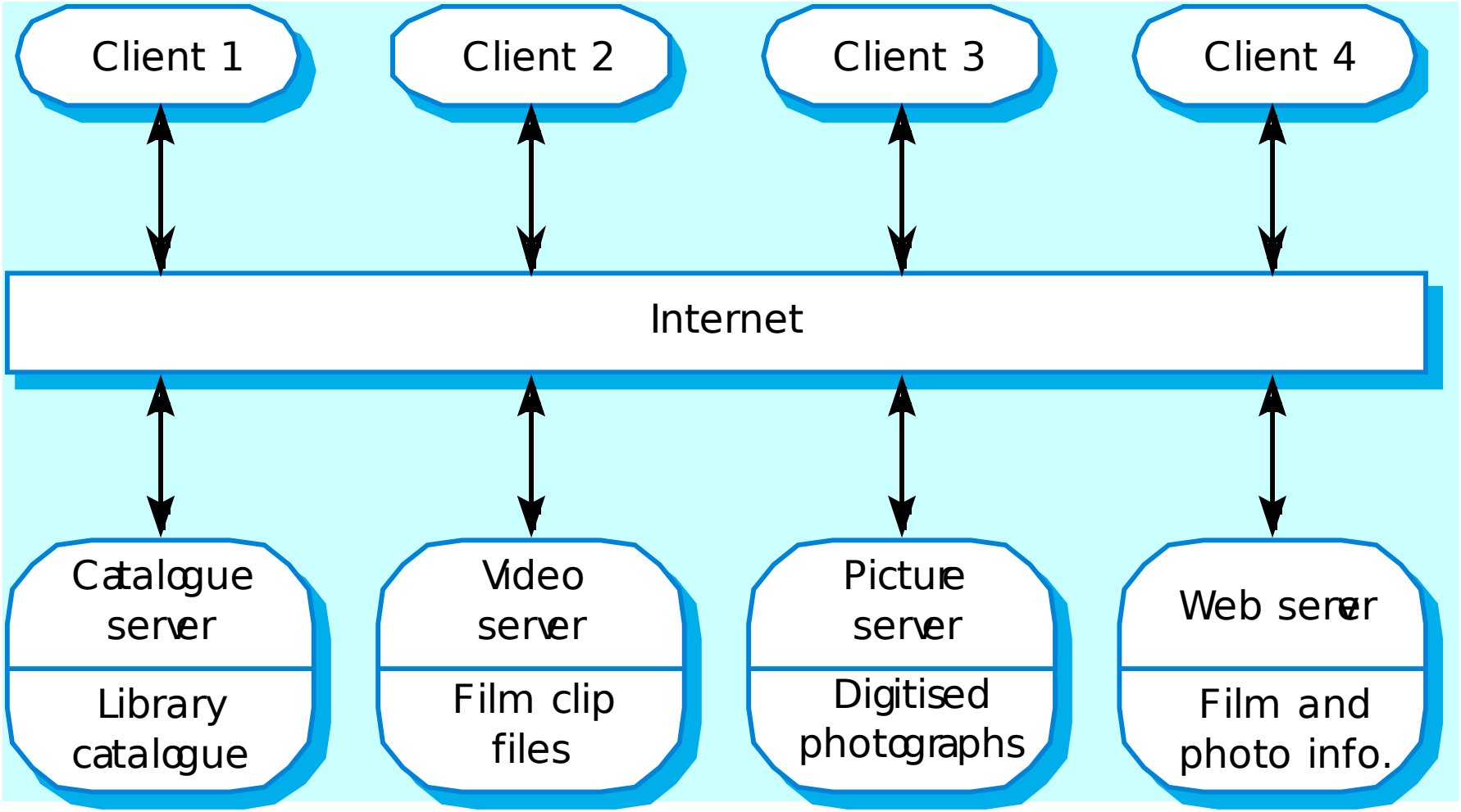
Repository model characteristics

- ***Advantages***
  - Efficient way to share large amounts of data;
  - Sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc.
- ***Disadvantages***
  - Sub-systems must agree on a repository data model. Inevitably a compromise;
  - Data evolution is difficult and expensive;
  - Difficult to distribute efficiently.

Client-server model

- Distributed system model which shows *how data and processing is distributed across a range of components*.
- *Set of stand-alone servers* which provide specific services such as printing, data management, etc.
- *Set of clients* which request for these services.
- Network which allows clients to access servers.

Film and picture library



Client-server characteristics

- ***Advantages***
  - Distribution of data is straightforward;
  - Makes effective use of networked systems. May require cheaper hardware;
  - Easy to add new servers or upgrade existing servers.
- ***Disadvantages***
  - No shared data model so sub-systems use different data organisation. Data interchange may be inefficient;
  - Redundant management in each server;
  - No central register of names and services - it may be hard to find out what servers and services are available.

Abstract machine (layered) model

- Used to *model the interfacing of sub-systems*.
- Organises the *system into a set of layers* (or abstract machines) each of which provide *a set of services*.
- Supports the *incremental development* of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

Modular decomposition styles

- Styles of *decomposing sub-systems into modules*.
- No rigid distinction between system organisation and modular decomposition.

Sub-systems and modules

- A *sub-system* is a system in its own right whose operation is ***independent*** of the services provided by other sub-systems.
- A *module* is a system component that provides services to other components but would ***not*** normally be considered as a ***separate*** system.

Modular decomposition

- Two modular decomposition models covered
  - An ***object model*** where the system is decomposed into interacting object;
  - A pipeline or data-flow model where the system is decomposed into ***functional modules*** which transform inputs to outputs.
- If possible, decisions about concurrency should be delayed until modules are implemented.

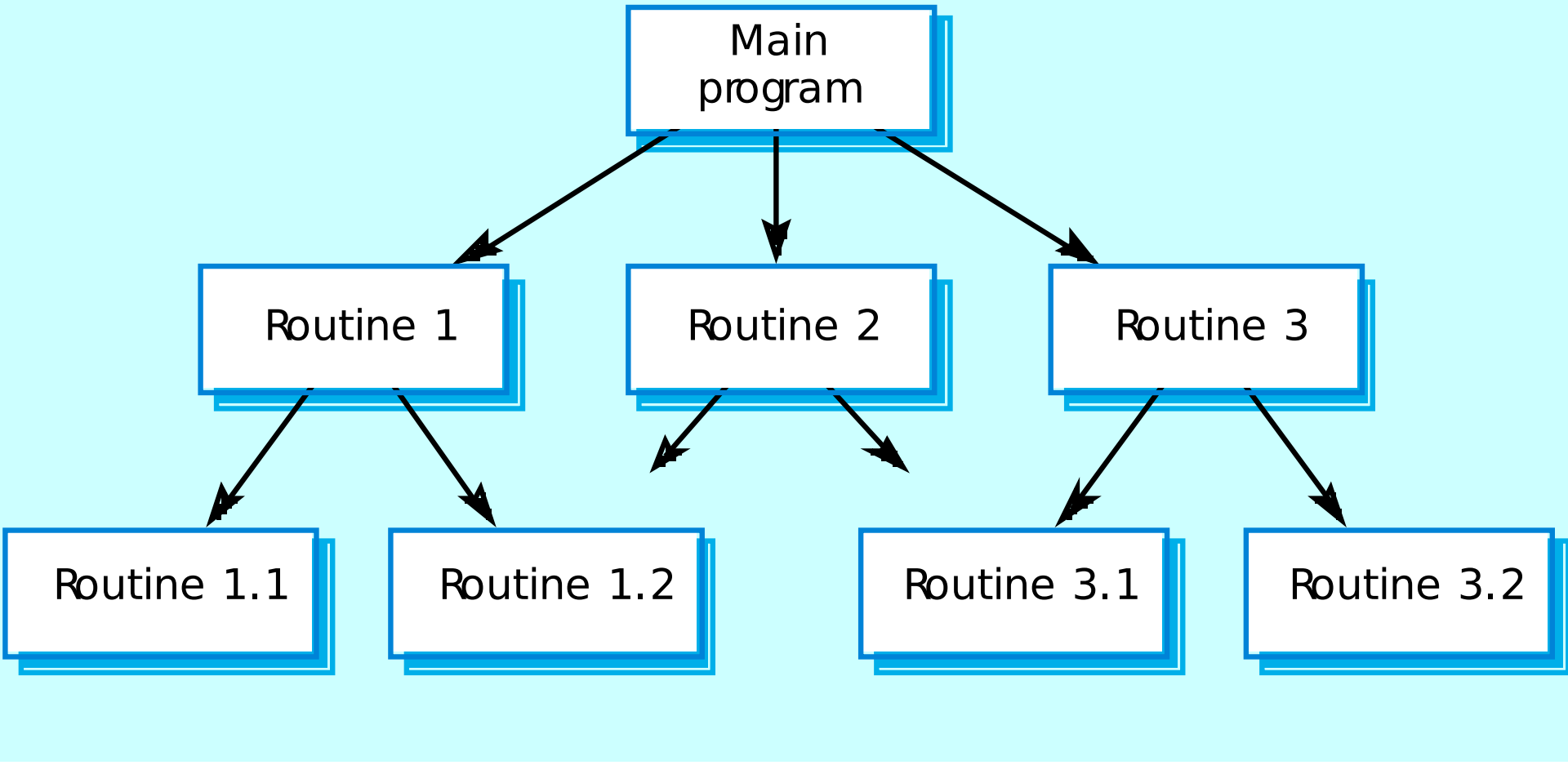
Control styles

- Ways of the control flow between sub-systems. Two ways:
  - ***Centralised control***
  - ***Event-based control***

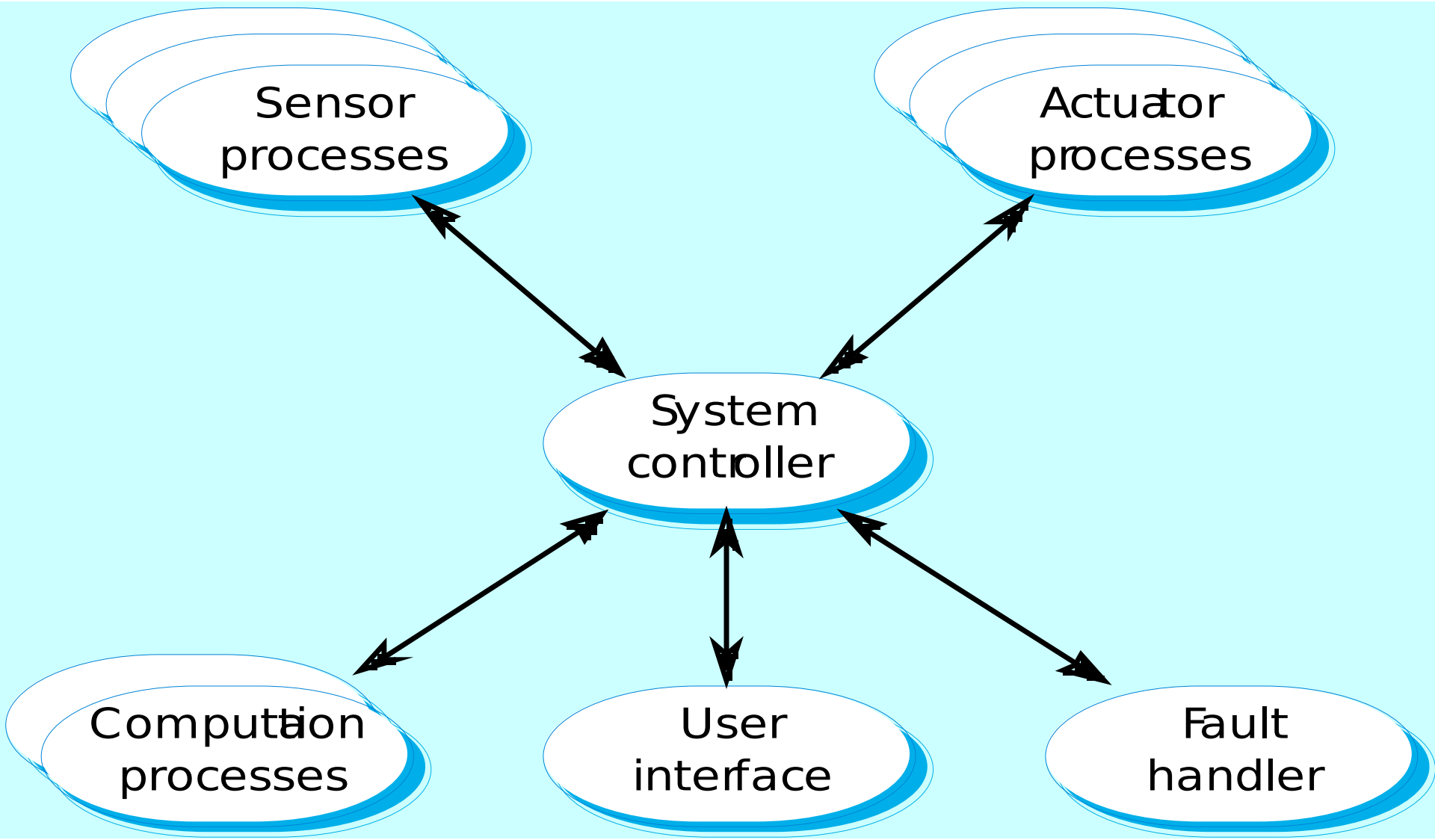
Centralised control

- A control sub-system takes responsibility for managing the execution of other sub-systems.
- Call-return model
  - Top-down subroutine model
  - Control starts at the root of a subroutine tree and moves downwards.
  - Applicable to sequential systems.
- Manager model
  - Applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. Can be implemented in sequential systems as a case statement.

Call-return model



Real-time system control



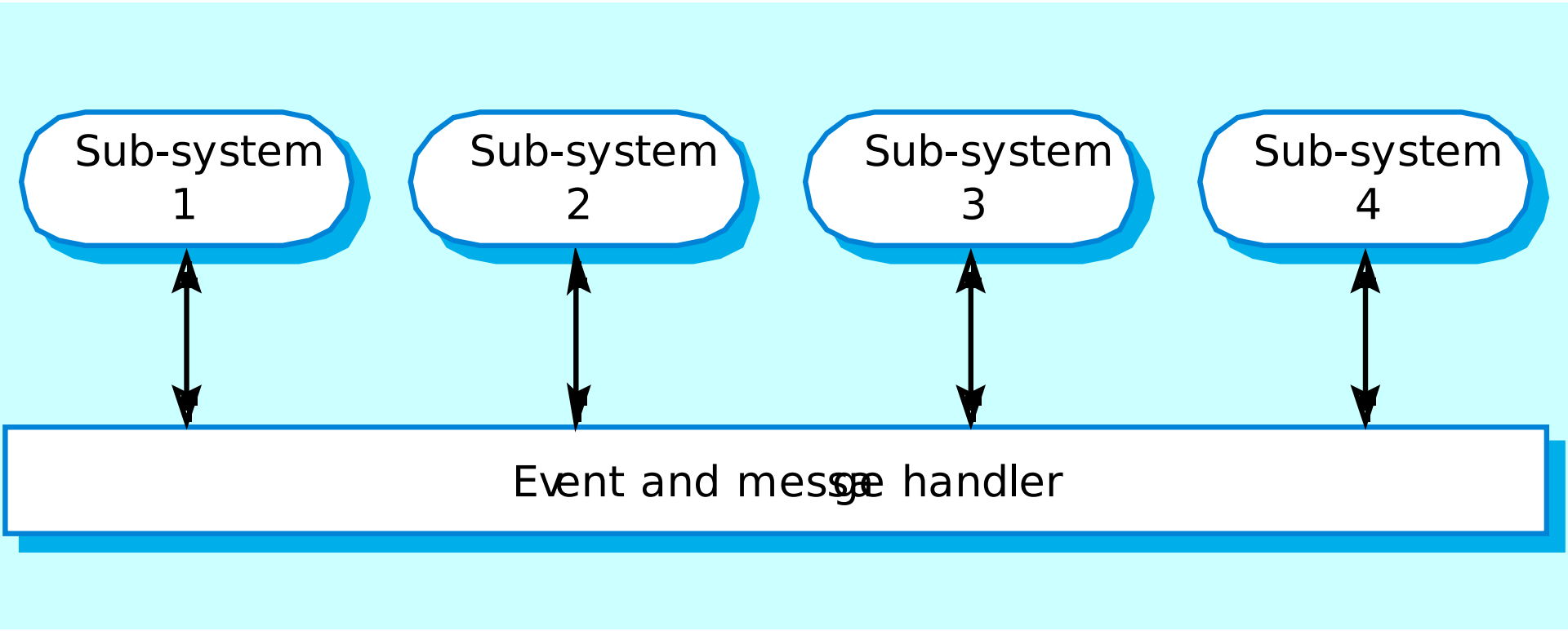
Event-driven systems

- The control of the sub-systems processing the event is driven by externally generated events
- Two primary event-driven models
  - ***Broadcast models***. An *event is broadcast to all sub-systems*. Any sub-system which can handle the event may do so;
  - ***Interrupt-driven models***. *Used in real-time systems* where interrupts are detected by an interrupt handler and passed to some other component for processing.
- Other event driven models include spreadsheets and production systems.

Broadcast model

- Effective in *integrating sub-systems on different computers* in a network.
- *Sub-systems register an interest in specific events*. When these occur, control is transferred to the sub-system which can handle the event.
- Control policy is not embedded in the event and message handler. Sub-systems decide on events of interest to them.
- However, sub-systems don’t know if or when an event will be handled.

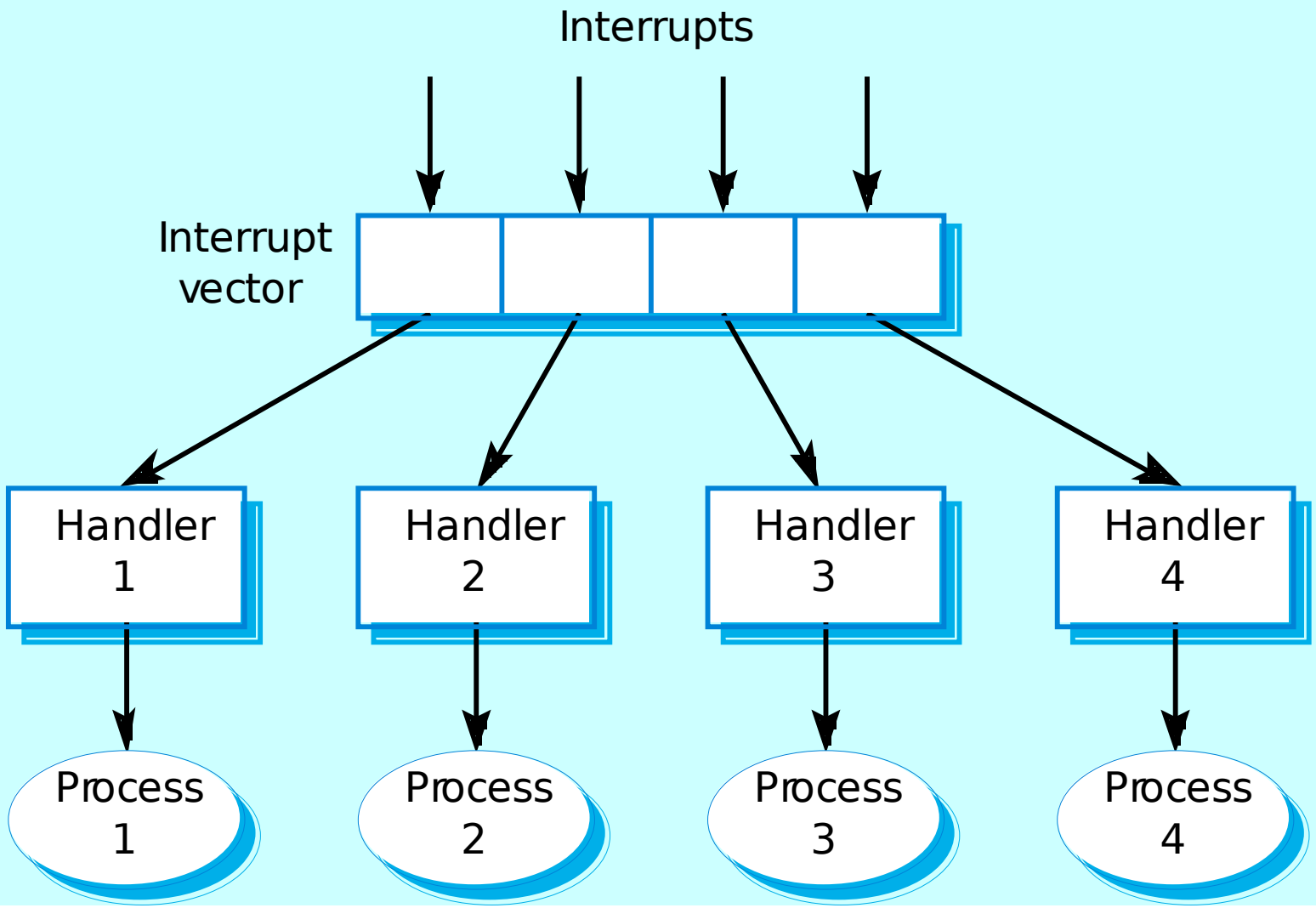
Selective broadcasting



Interrupt-driven systems

- *Used in real-time systems* where *fast response to an event is essential*.
- There are known *interrupt types with a handler* defined for each type.
- Each type is associated with *a memory location and a hardware switch causes transfer to its handler*.
- Allows *fast response but complex to program and difficult to validate*.

Interrupt-driven control



Distributed System Design

Distributed systems

- Virtually all large computer-based systems are now distributed systems.
- 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 system disadvantages

- ***Complexity***
  - Typically, ***distributed systems are more complex*** than centralised systems.
- ***Security***
  - ***More susceptible*** to external attack.
- ***Manageability***
  - ***More effort required*** for system management.
- ***Unpredictability***
  - Unpredictable responses depending on the system organisation and ***network load***

Distributed systems architectures

- ***Client-server architectures***
  - *Distributed services*
    - provided by *servers*
    - requested and used by *clients*.
- ***Distributed object architectures***
  - *No distinction between clients and servers.*

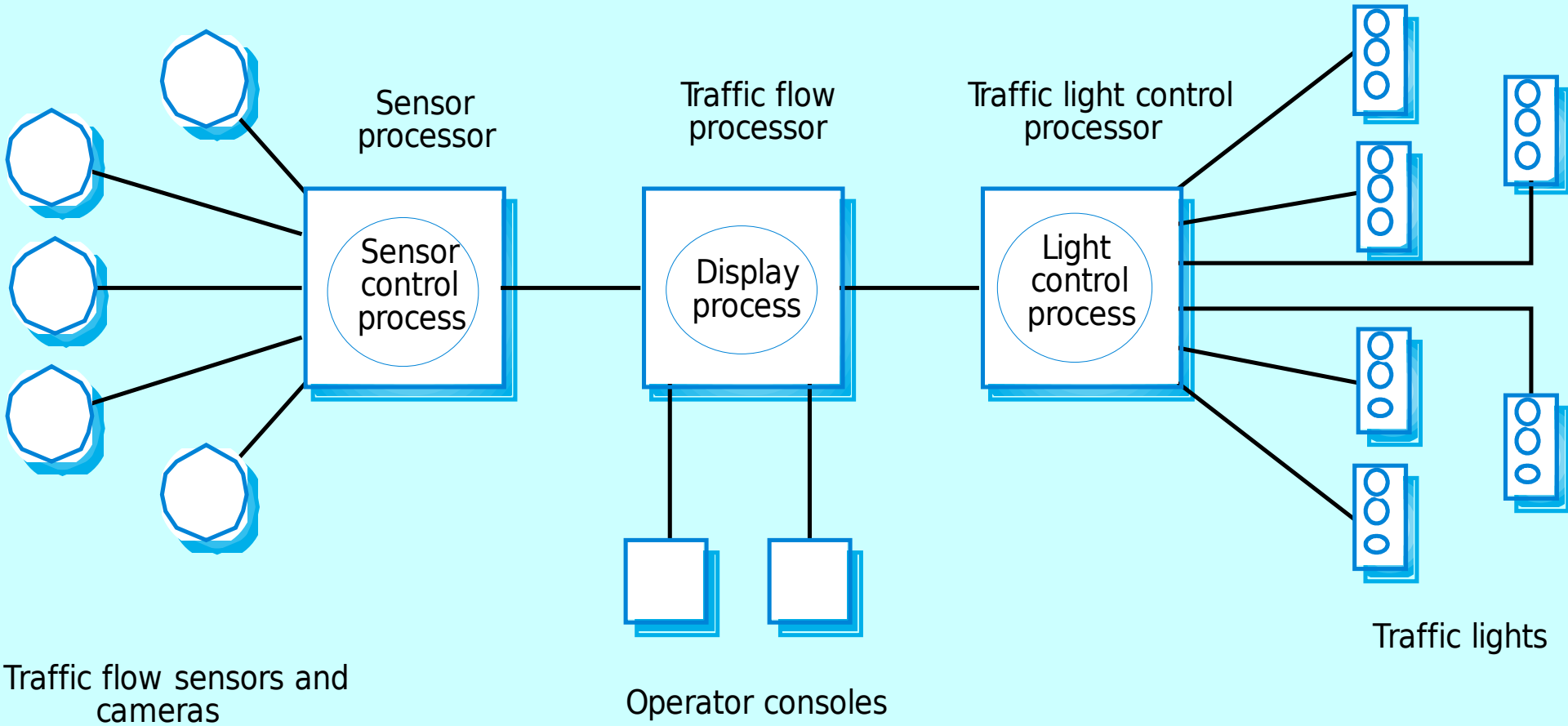
Middleware

- *Software that manages and supports the different components of a distributed system*. In essence, it sits in the ***middle*** of the system.
- Middleware is usually ***off-the-shelf*** rather than specially written software.
- Examples
  - Transaction processing monitors;
  - Data converters;
  - Communication controllers.

Multiprocessor architectures

- *Simplest distributed system* model.
- System composed of *multiple processes* which may (but need not) *execute on different processors*.
- Architectural model of many *large real-time systems*.
- *Process to processor distribution*
  - *pre-ordered*
  - *under the control of a dispatcher*.

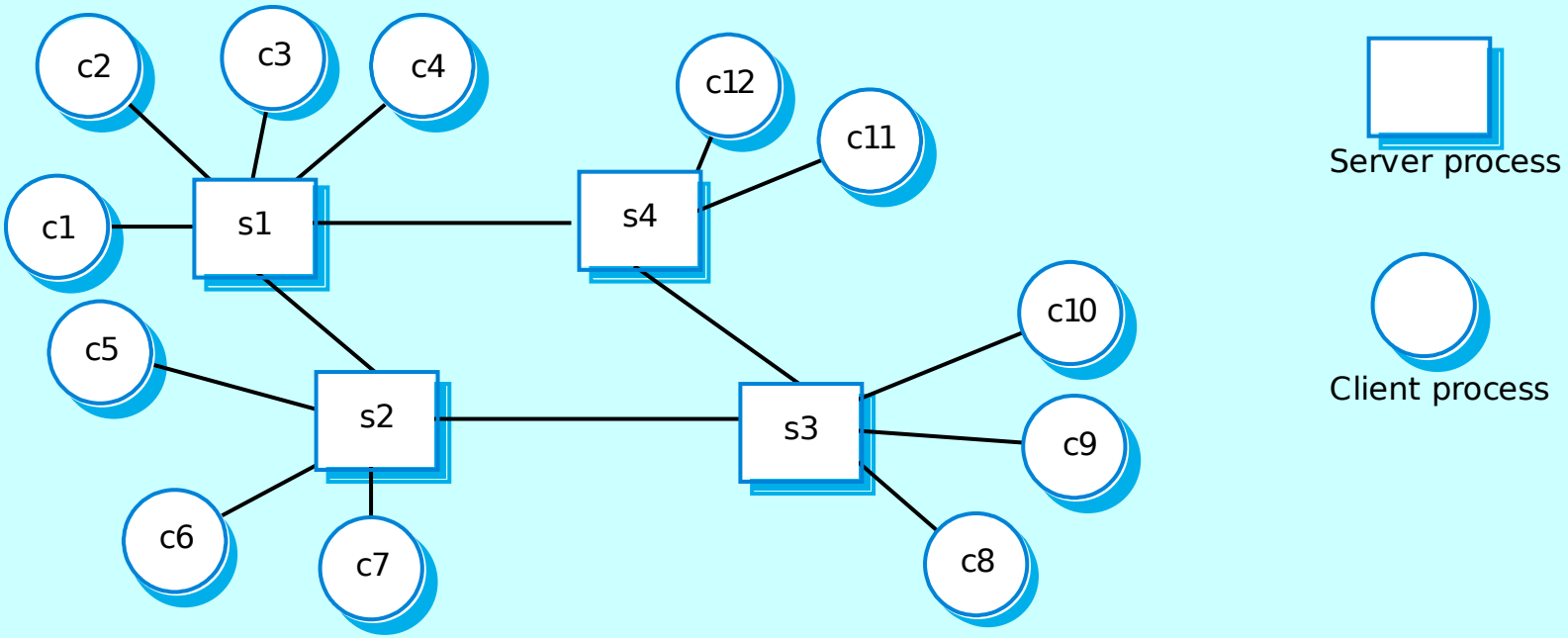
A multiprocessor traffic control system



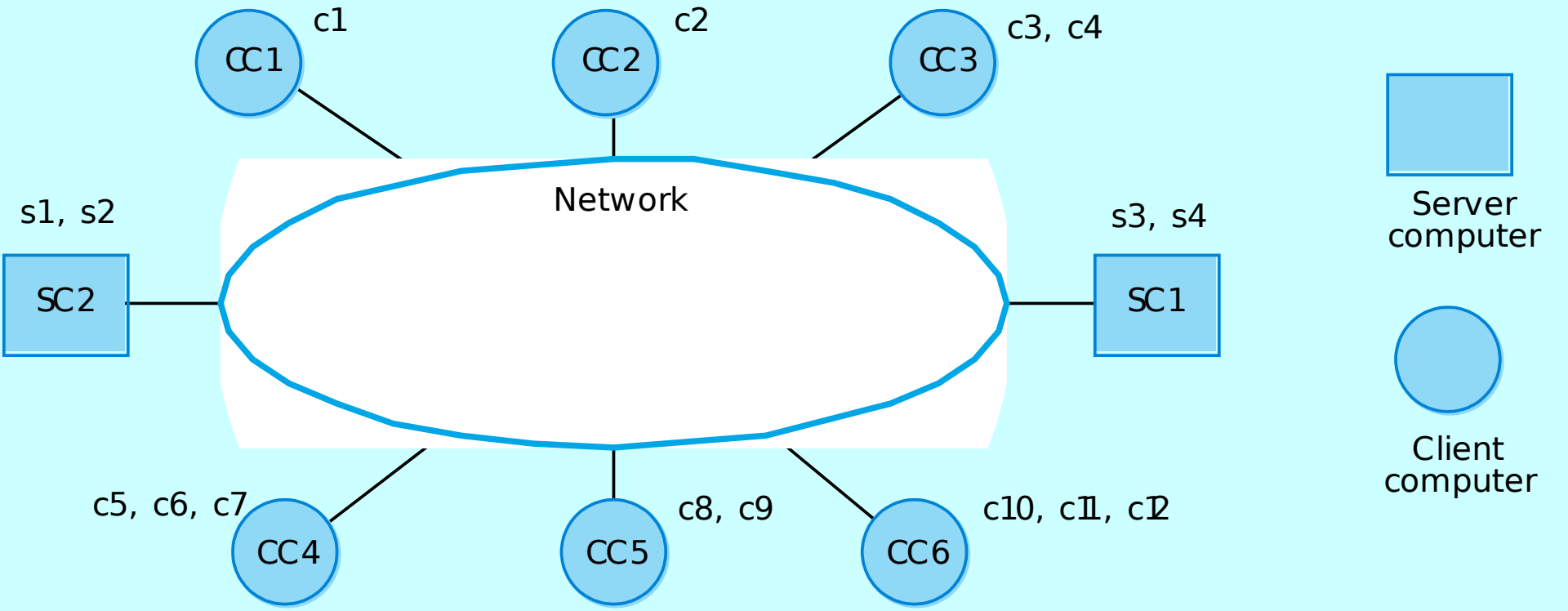
Client-server architectures

- The application is modelled as a set of services that are provided by servers and a set of clients that use these services.
- Clients know of servers but servers need not know of clients.
- Clients and servers are logical processes
- The mapping of processors to processes is not necessarily 1 : 1.

A client-server system



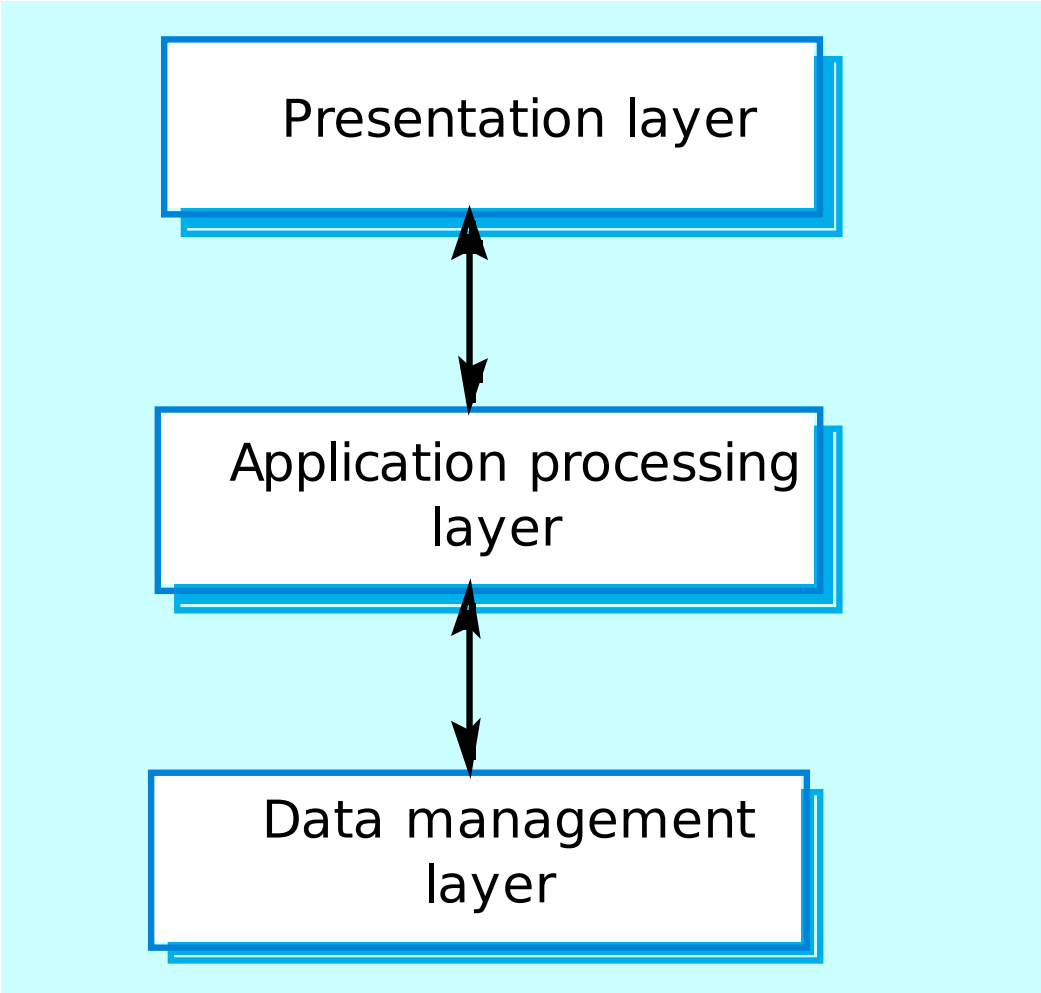
Computers in a C/S network



Layered application architecture

- ***Presentation layer***
  - *presenting the results of a computation* to users and
  - *collecting user inputs*.
- ***Application processing layer***
  - *providing application specific functionality* e.g., in a banking system, banking functions such as open account, close account, etc.
- ***Data management layer***
  - *managing the system databases*.

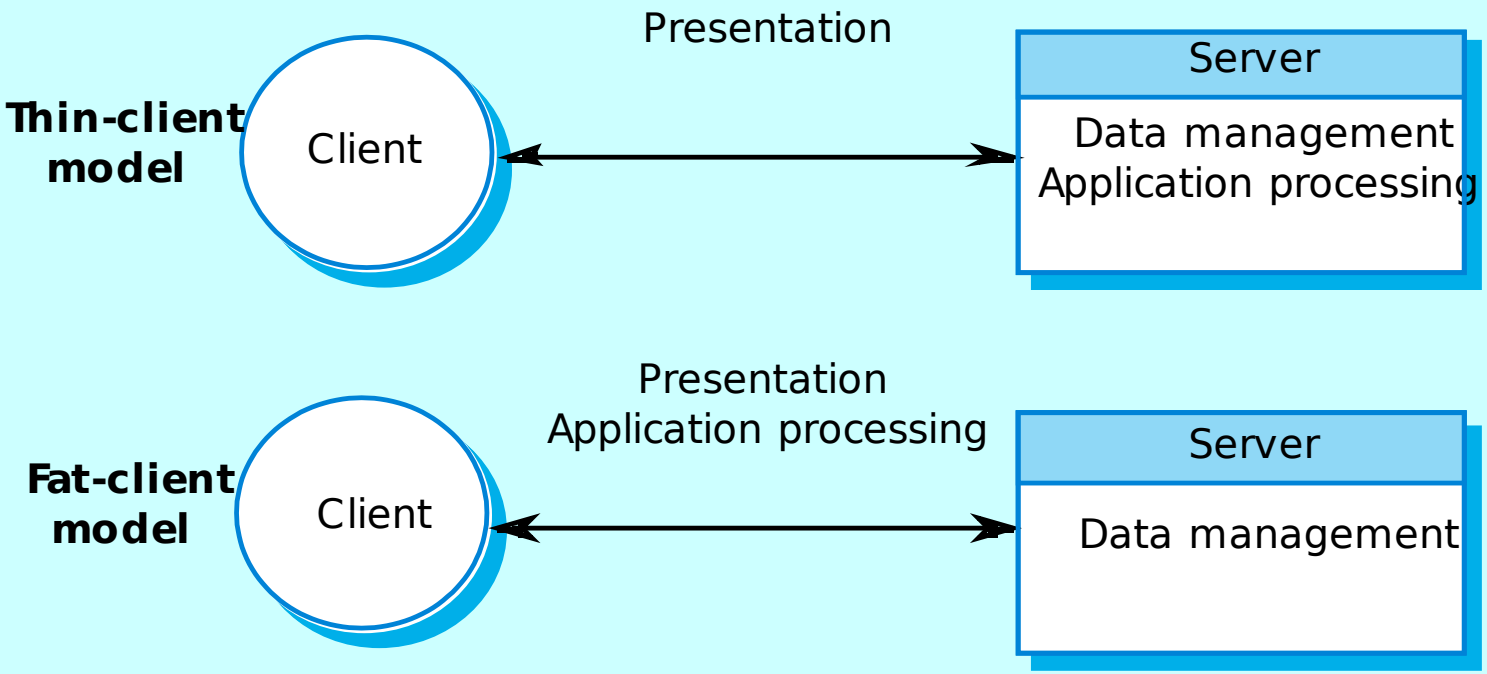
Application layers



Thin and fat clients

- ***Thin-client model***
  - *Application processing and data management is on the server.*
  - *Client simply responsible for running presentation software.*
- ***Fat-client model***
  - *Server only responsible for data management.*
  - *Client realizes application logic and interactions with the system user.*

Thin and fat clients



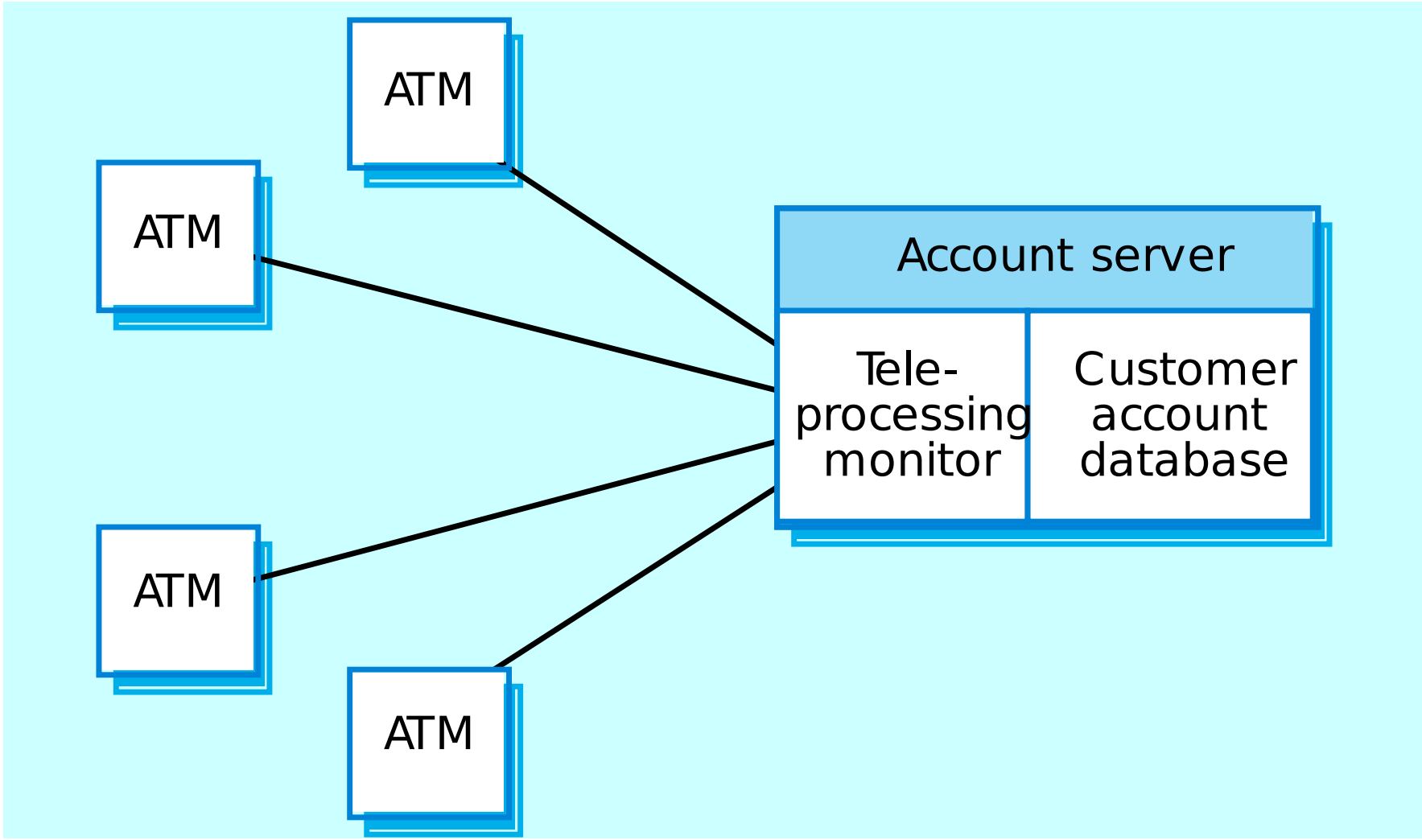
Thin client model

- 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.

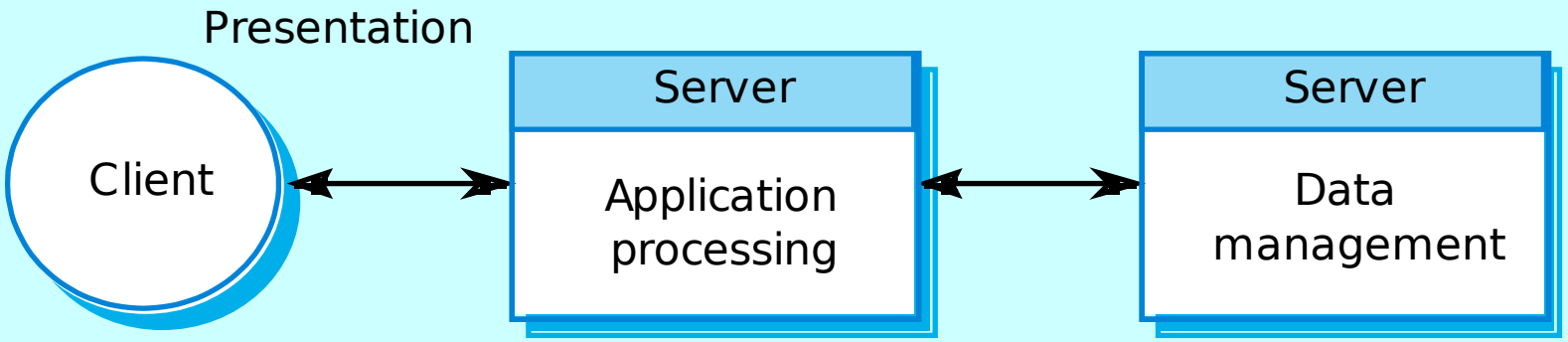
A client-server ATM system



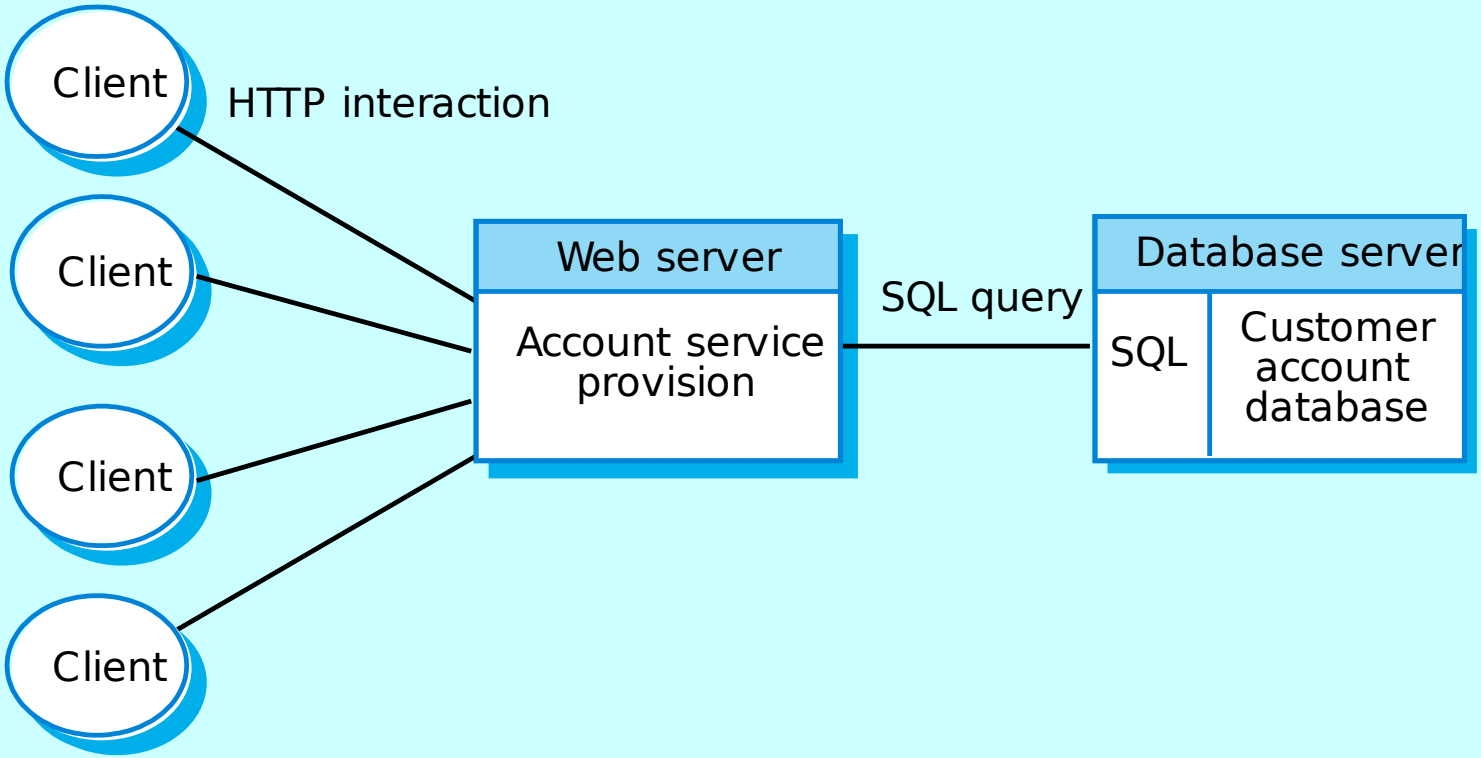
Three-tier architectures

- In a three-tier architecture, each of the application architecture *layers may execute on a separate processor*.
- Allows for *better performance than a thin-client approach and is simpler to manage than a fat-client approach.*
- A *more scalable architecture - as demands increase. extra servers can be added.*

A 3-tier C/S architecture



An internet banking system



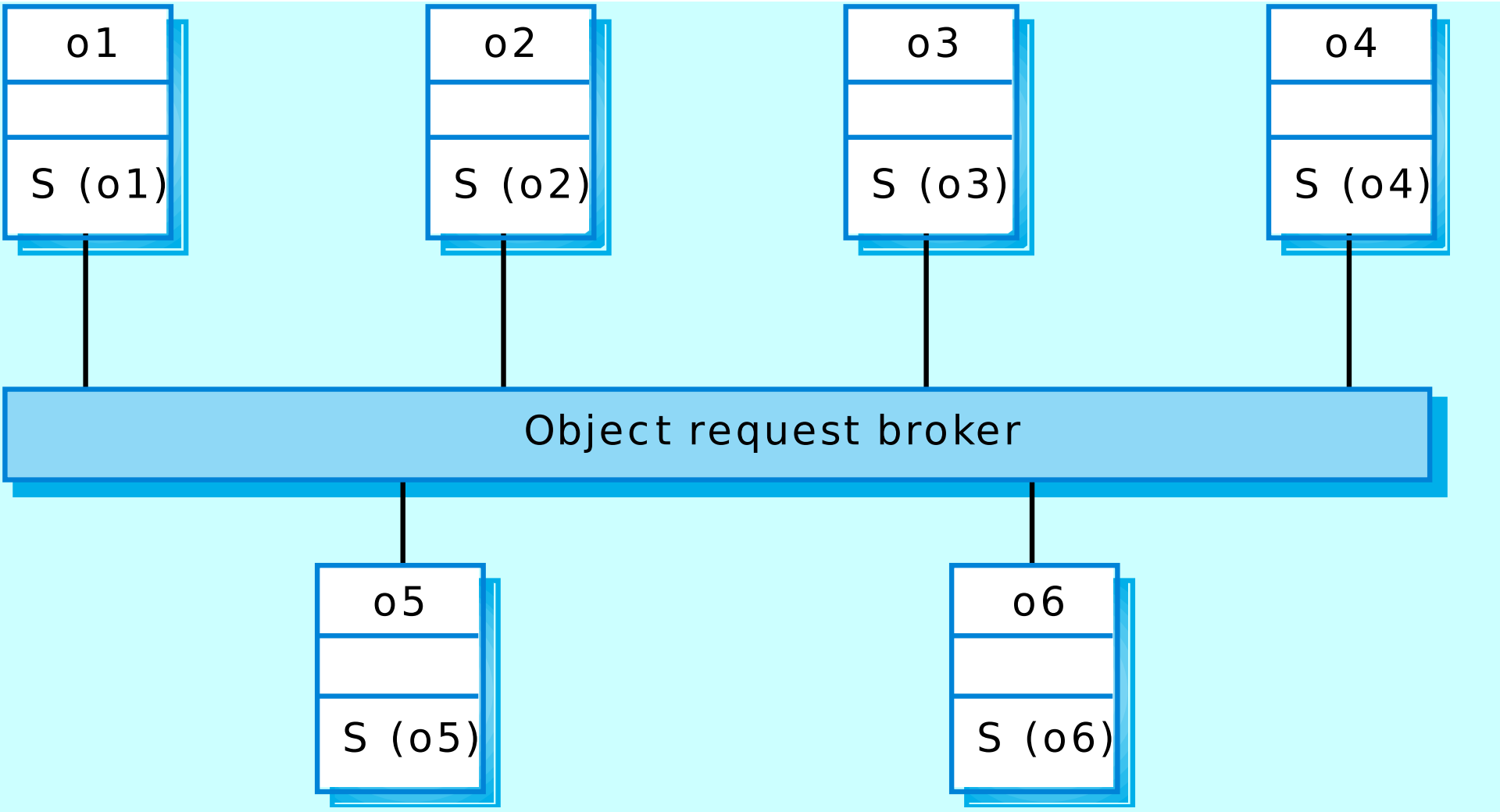
Use of C/S architectures

Architecture	Applications
Two-tier C/S architecture with thin clients	Legacy system applications where separating application processing and data management is impractical. Computationally-intensive applications such as compilers with little or no data management. Data-intensive applications (browsing and querying) with little or no application processing.
Two-tier C/S 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 visualisation) is required. Applications with relatively stable end-user functionality used in an environment with well-established system management.
Three-tier or multi-tier C/S 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 object architectures

- There is ***no distinction*** in a distributed object architectures between clients and servers.
- Each distributable entity is *an object* that provides *services to other objects and receives services from other objects*.
- Object *communication is through a middleware* system called an *object request broker*.
- However, *distributed object architectures are more complex to design than C/S systems*.

Distributed object architecture



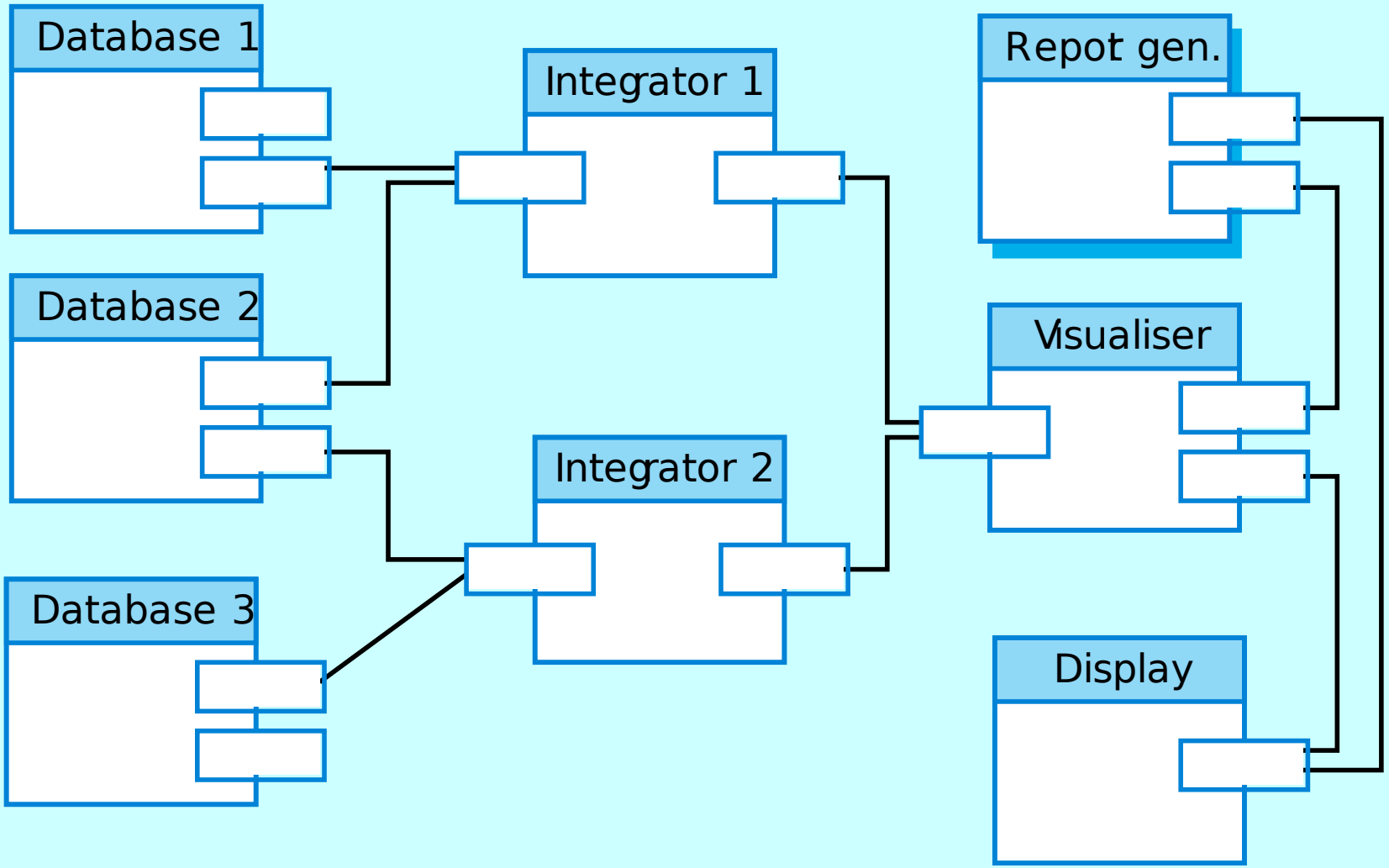
Advantages of distributed object architecture

- 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 to it as required.
- The system is *flexible and scaleable*.
- It is *possible to reconfigure the system dynamically* with objects migrating across the network as required.

Uses of distributed object architecture

- As a logical model allowing to *structure and organise the system*. In this case, you think about how to *provide application functionality solely in terms of services and combinations of services*.
- As a *flexible approach to the implementation of client-server systems*. The logical model of the system is a client-server model but both clients and servers are realised as distributed objects communicating through a common communication framework.

A data mining system



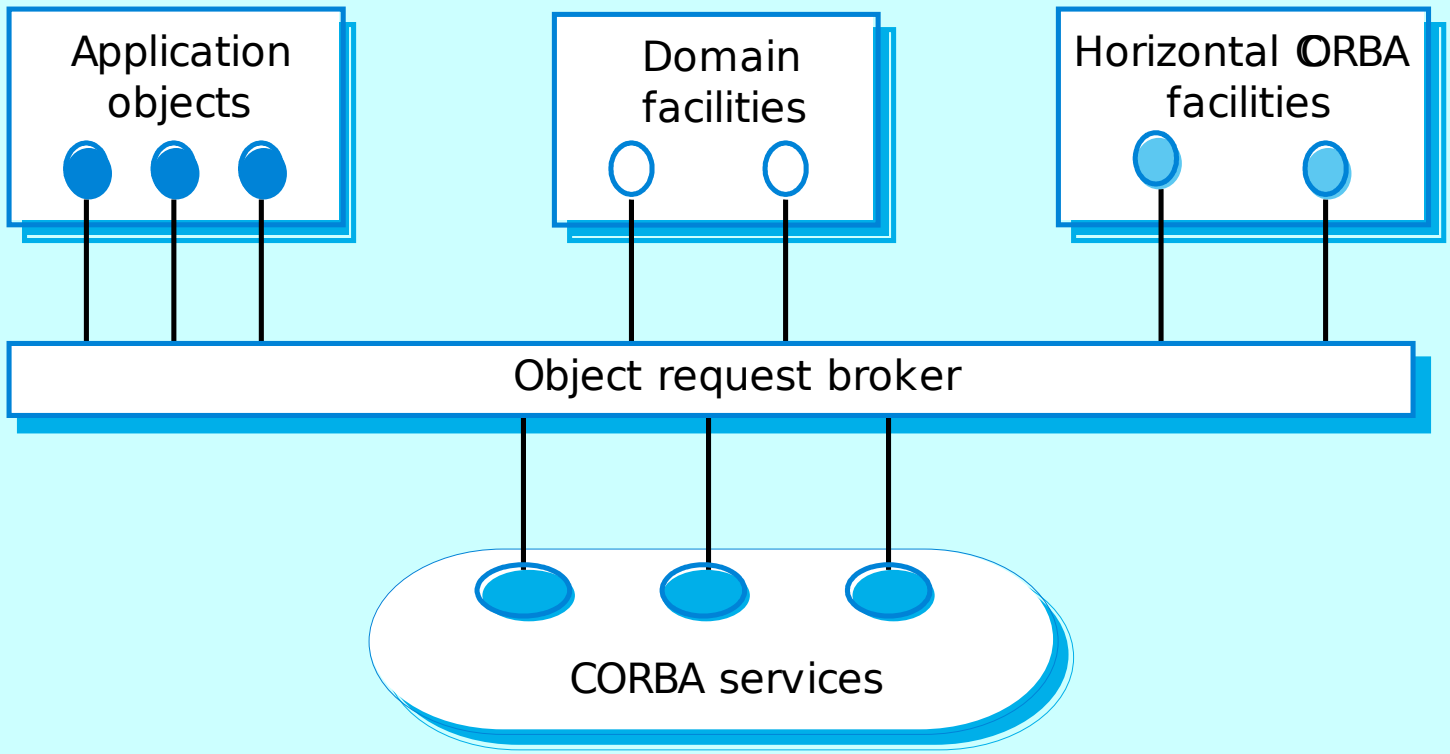
Data mining system

- The logical model of the system is not one of service provision where there are distinguished data management services.
- It allows the number of databases that are accessed to be increased without disrupting the system.
- It allows *new types of relationship* to be mined *by adding new integrator objects*.

CORBA

- CORBA: a *standard for an Object Request Broker* - *middleware to manage communications between distributed objects*.
- Middleware for distributed computing is required at 2 levels:
  - At the logical communication level, the middleware allows objects on different computers to exchange data and control information;
  - At the component level, the middleware provides a basis for developing compatible components. CORBA component standards have been defined

CORBA application structure



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.

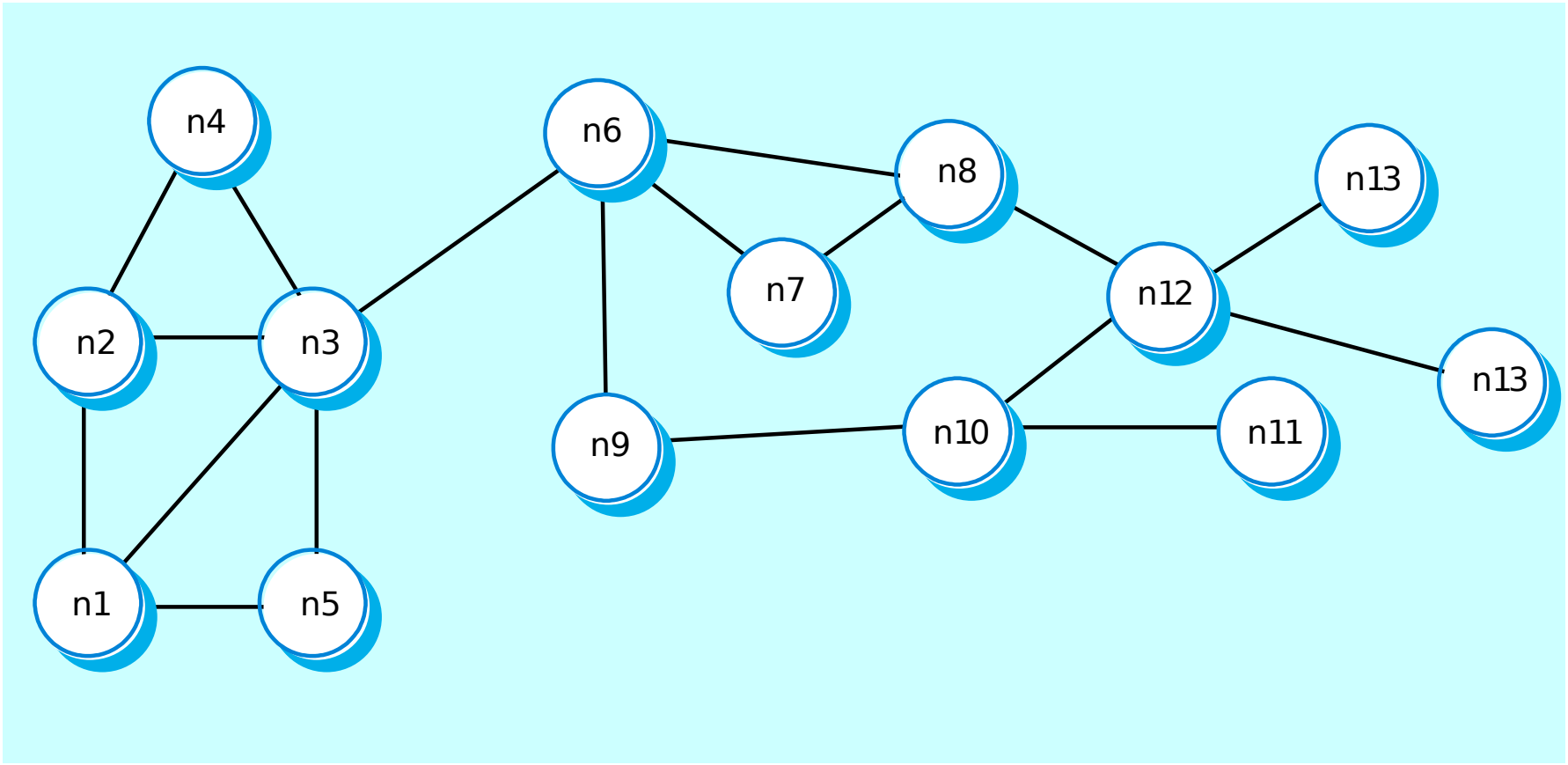
Examples to P2P architectures

- File sharing systems on PCs.
- Instant messaging systems such as ICQ to establish a direct communication between users.

P2p architectural models

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

Decentralised p2p architecture



Semi-centralised p2p architecture