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 – as applied by nature – 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 inspirations; but restricted to the expessive power of the words she selects to communicate her 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

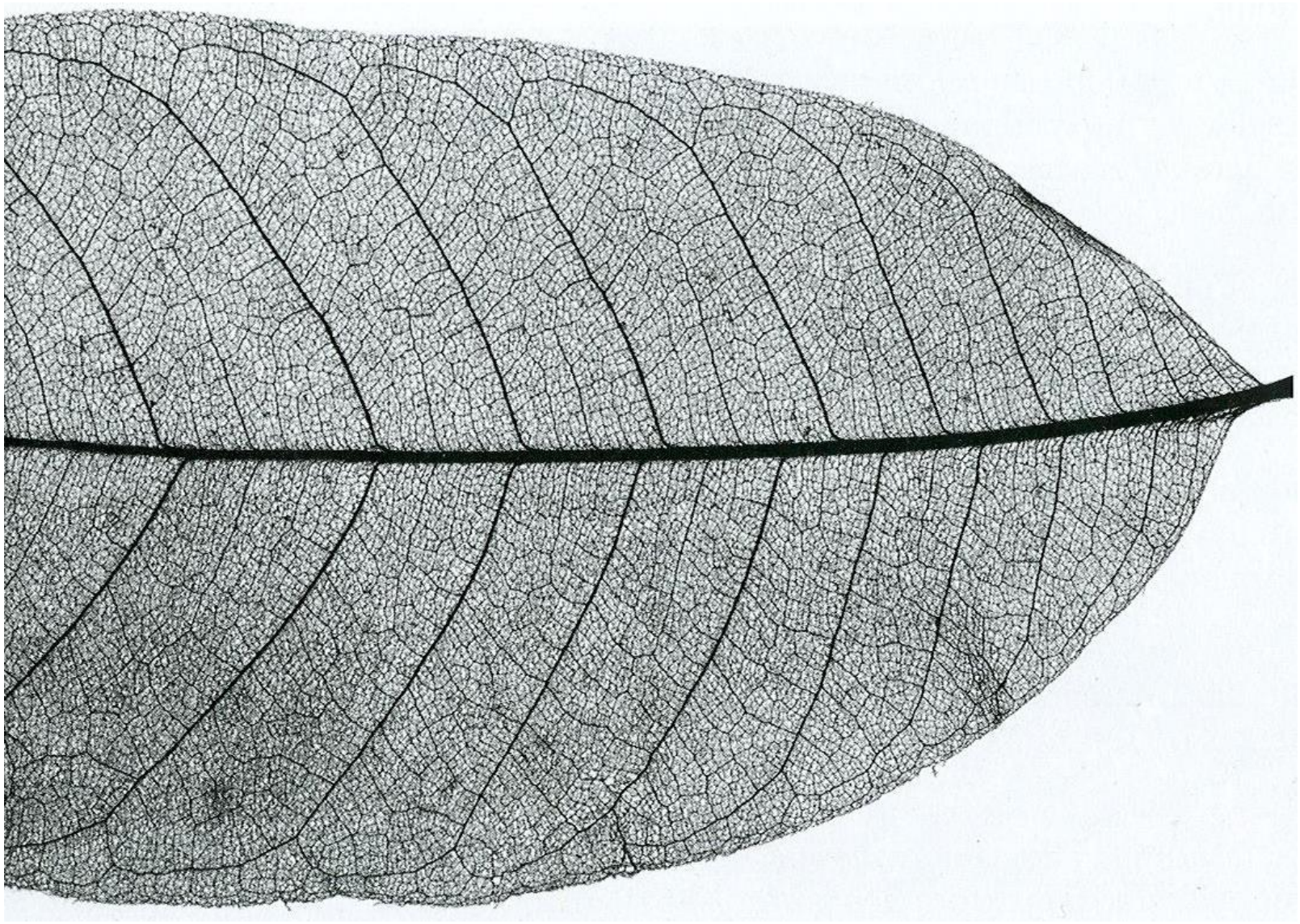
It is inevitable that the designer should have a well-established basis of knowledge on the set of communication tool(s) to use them to her best.

Nature's Tradition of Perfect Design

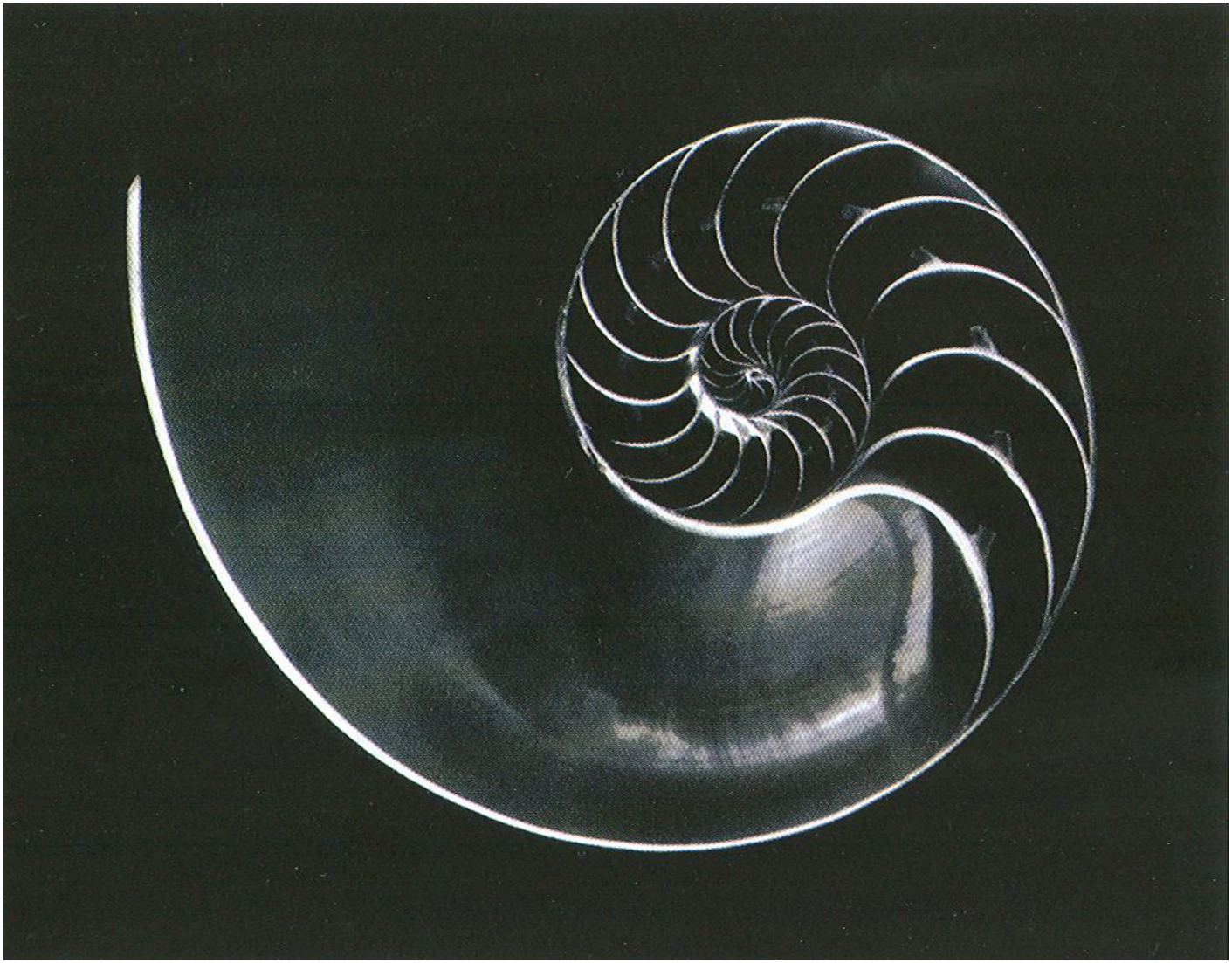
- *Nature designs perfectly since nature bases its designs upon the (to a significant extent unknown) internal reasons originating from an incredible level of balance in an enourmously diverse system of adaptable (evolving) organisms.*

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 is explained by 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 Top-Down & 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.*
- *Time that nature has is what designer does not. Hence, top-down approach is an option as a structuring approach to design in scarcity of time*

Key Points

- *Look for solutions as simple as possible;*
- *Nature's choice is also for as simple/optimal 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

- an *early stage* of the *system design*.
- a *link between specification and design*.
- ... involves *identifying major system components and their communications*.
- especially *cost-effective for large scale systems*.

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.

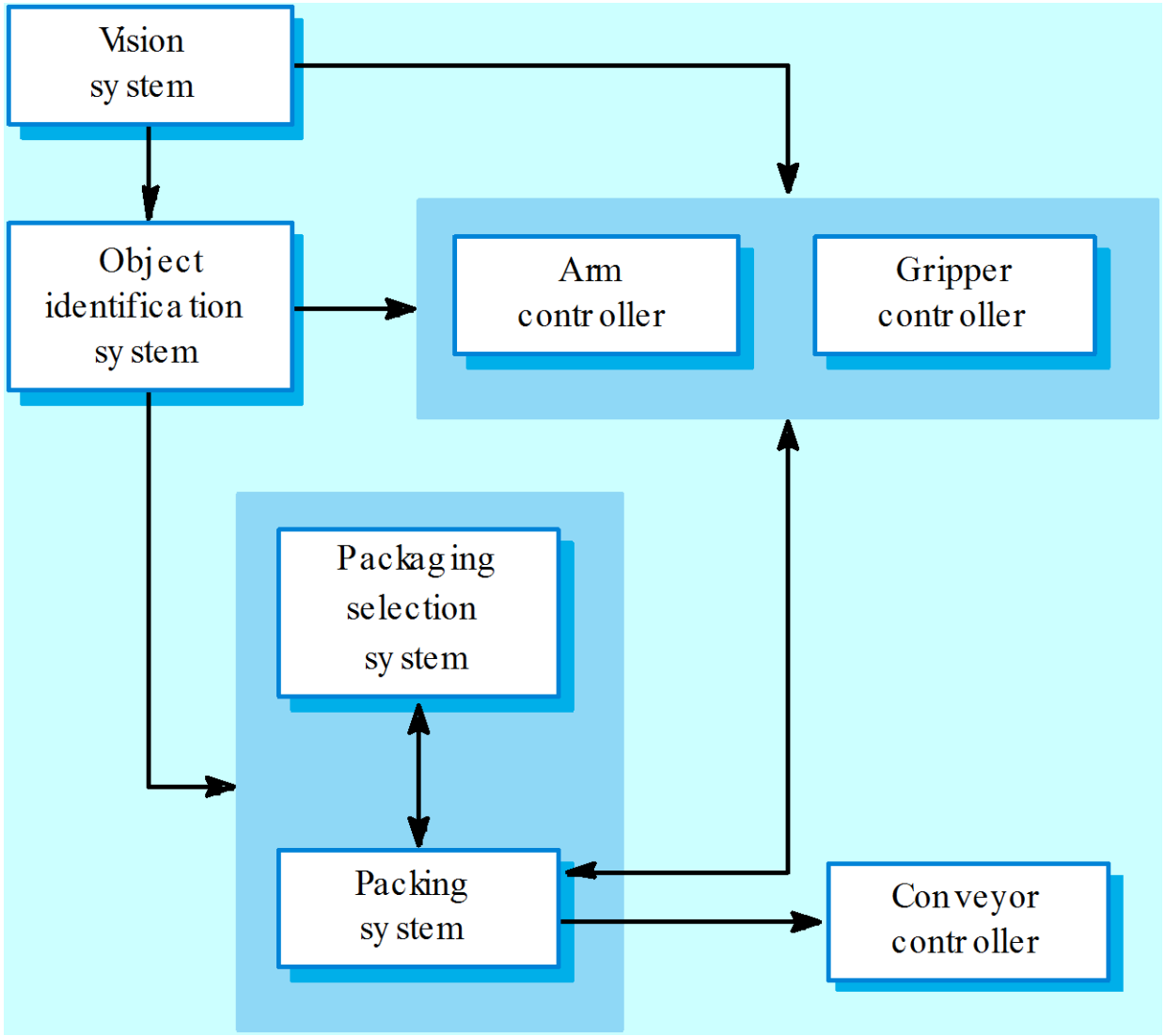
Architectural conflicts

- 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.

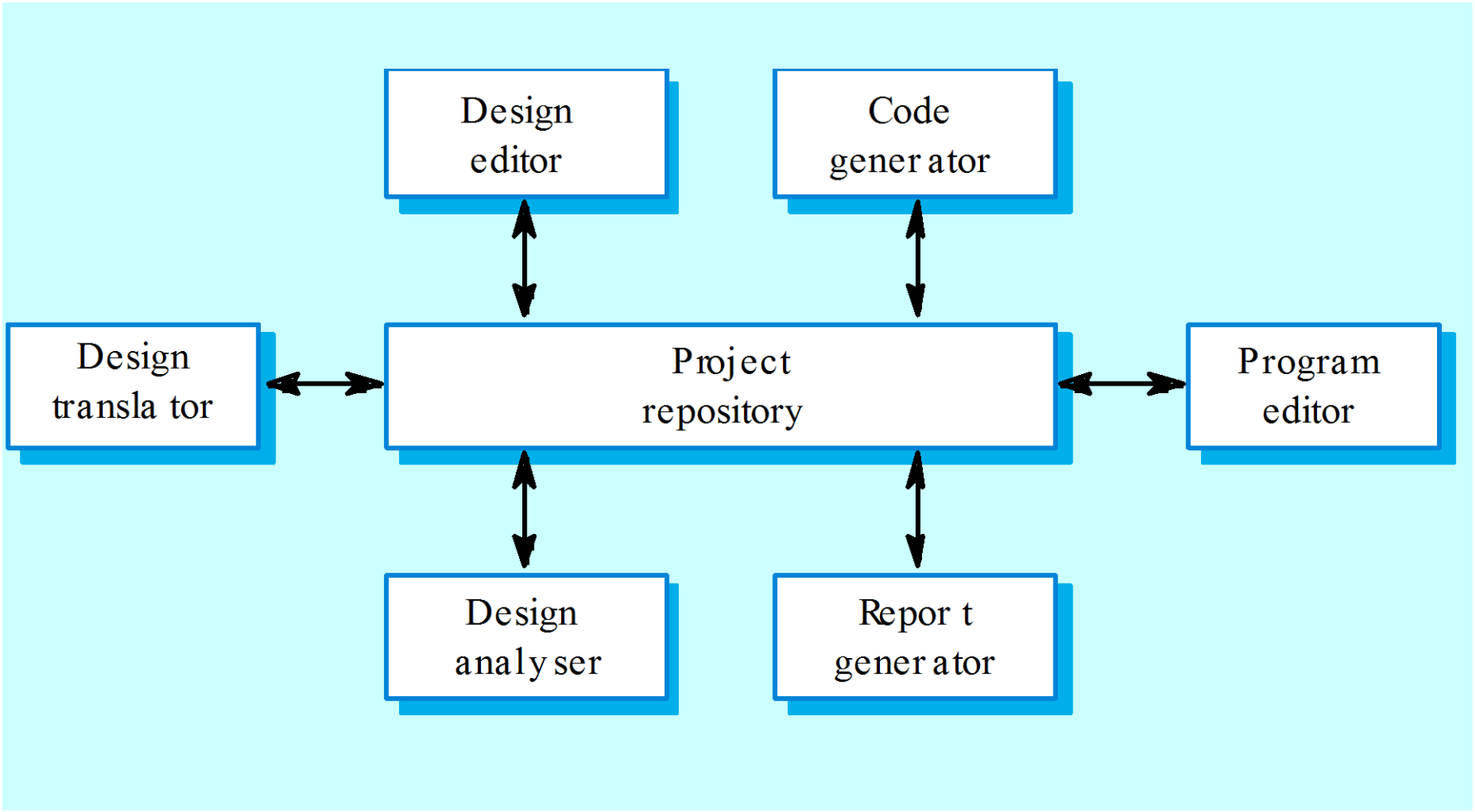
System organisation

- 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



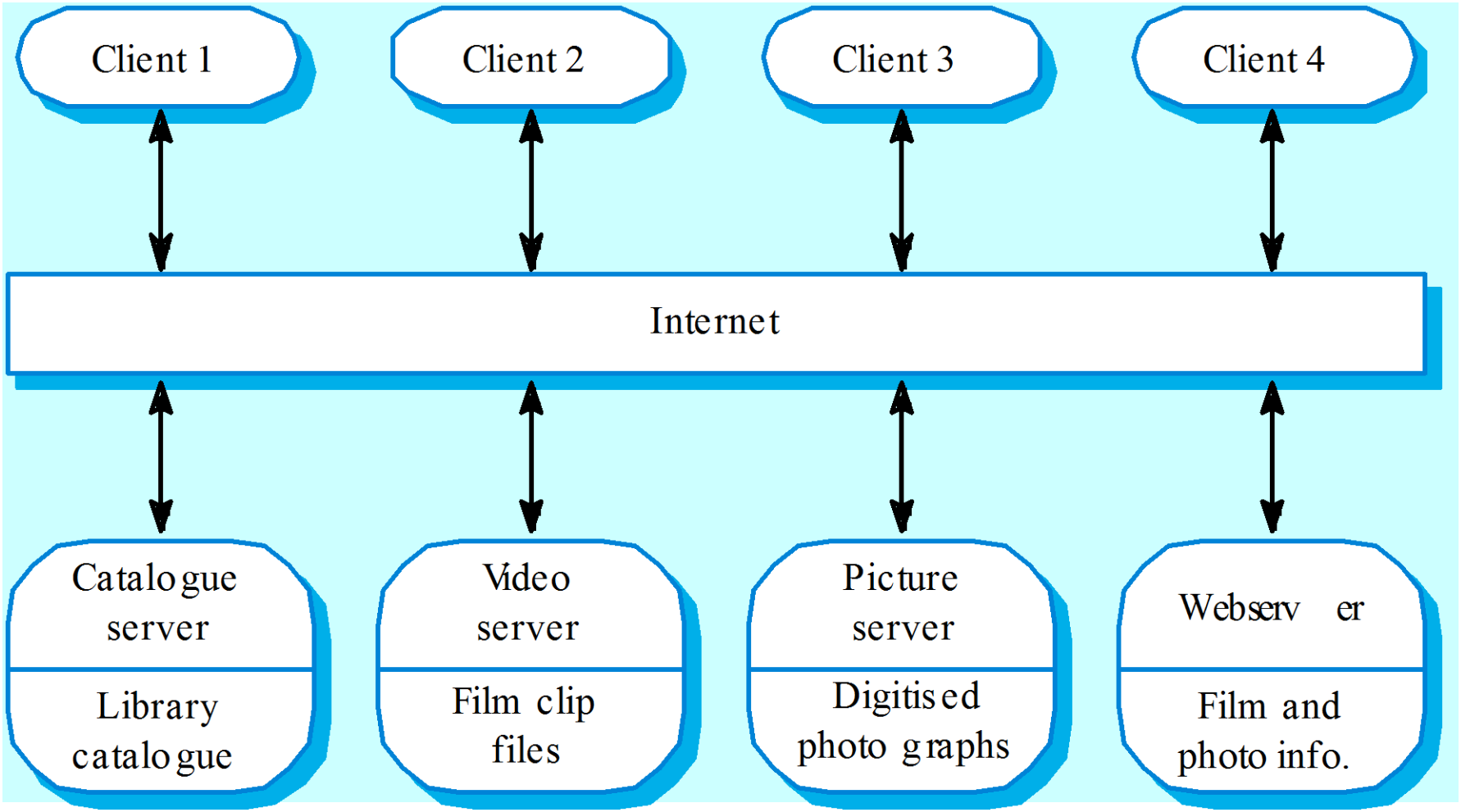
Repository model characteristics

- ***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.

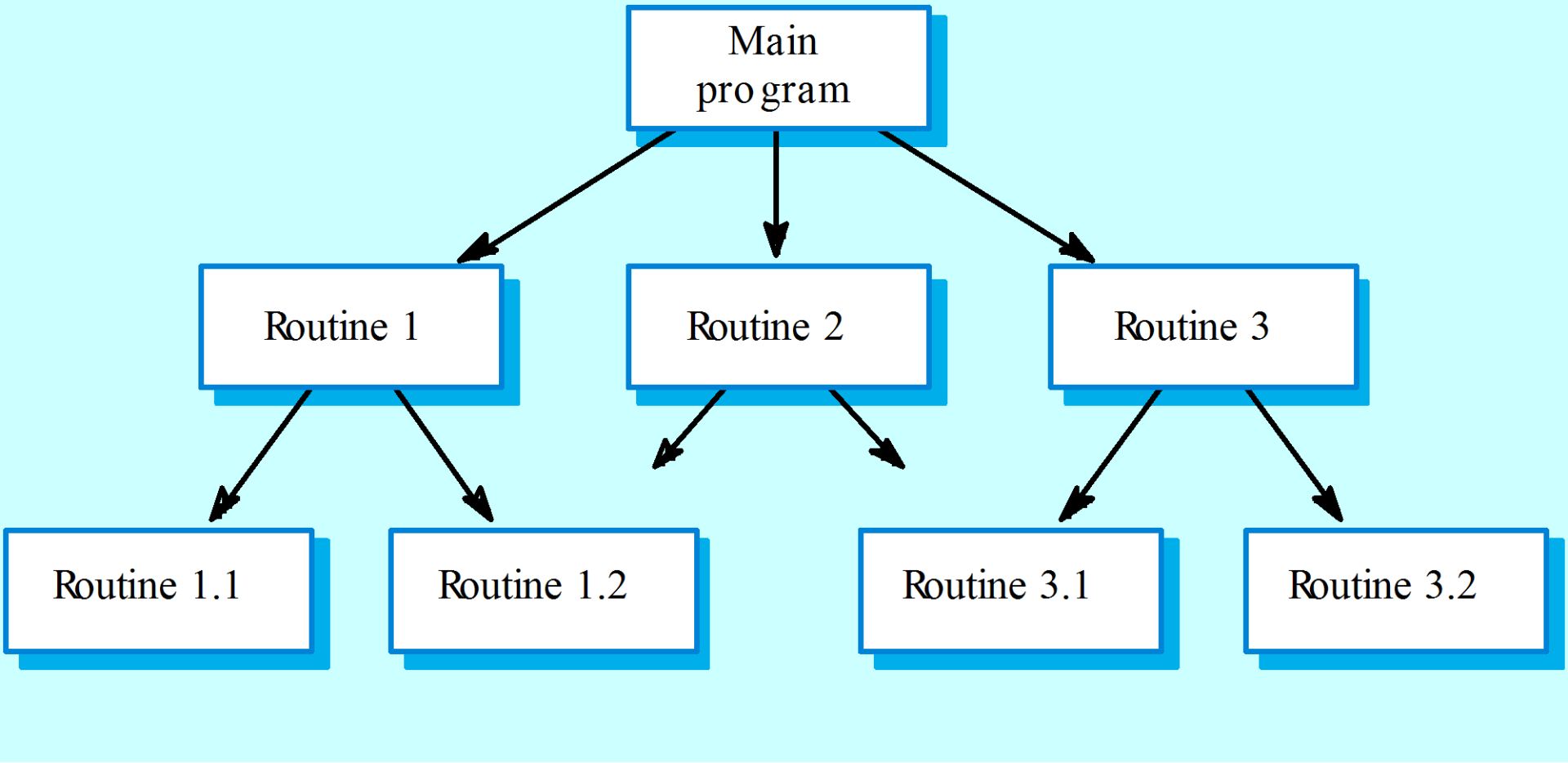
Control styles

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

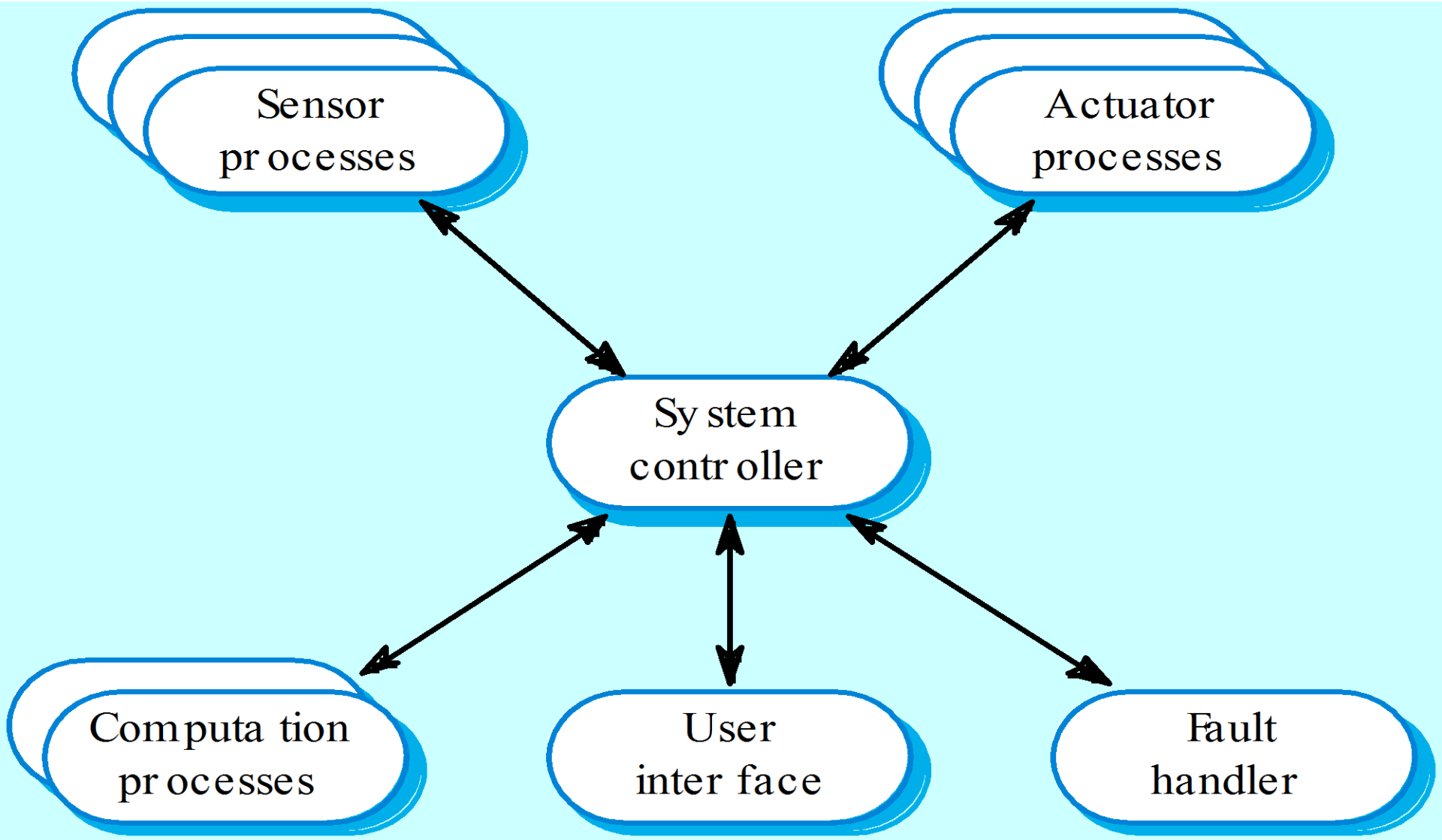
Centralised control

- 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 coordinates other system processes.

Call-return model



Real-time system control



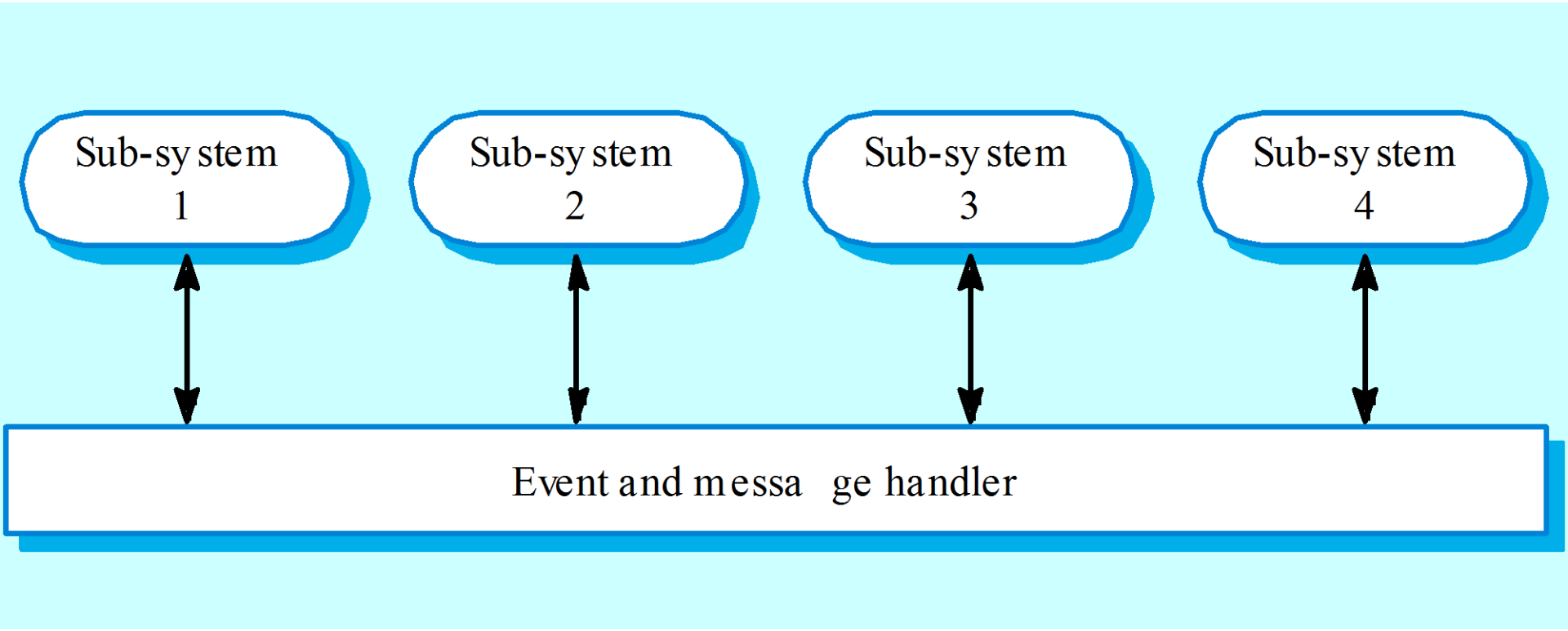
Event-driven systems

- 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.

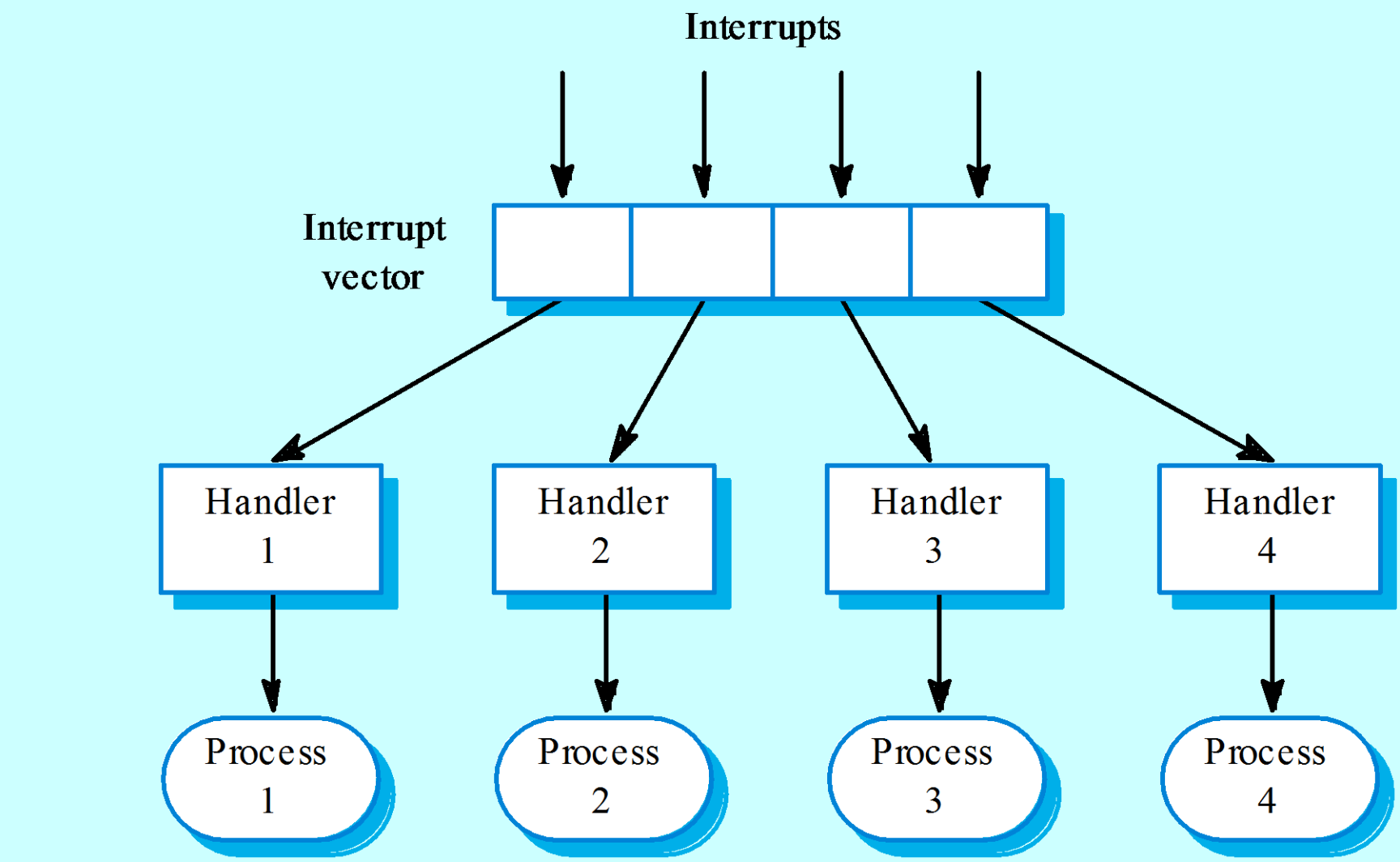
Selective broadcasting



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.*

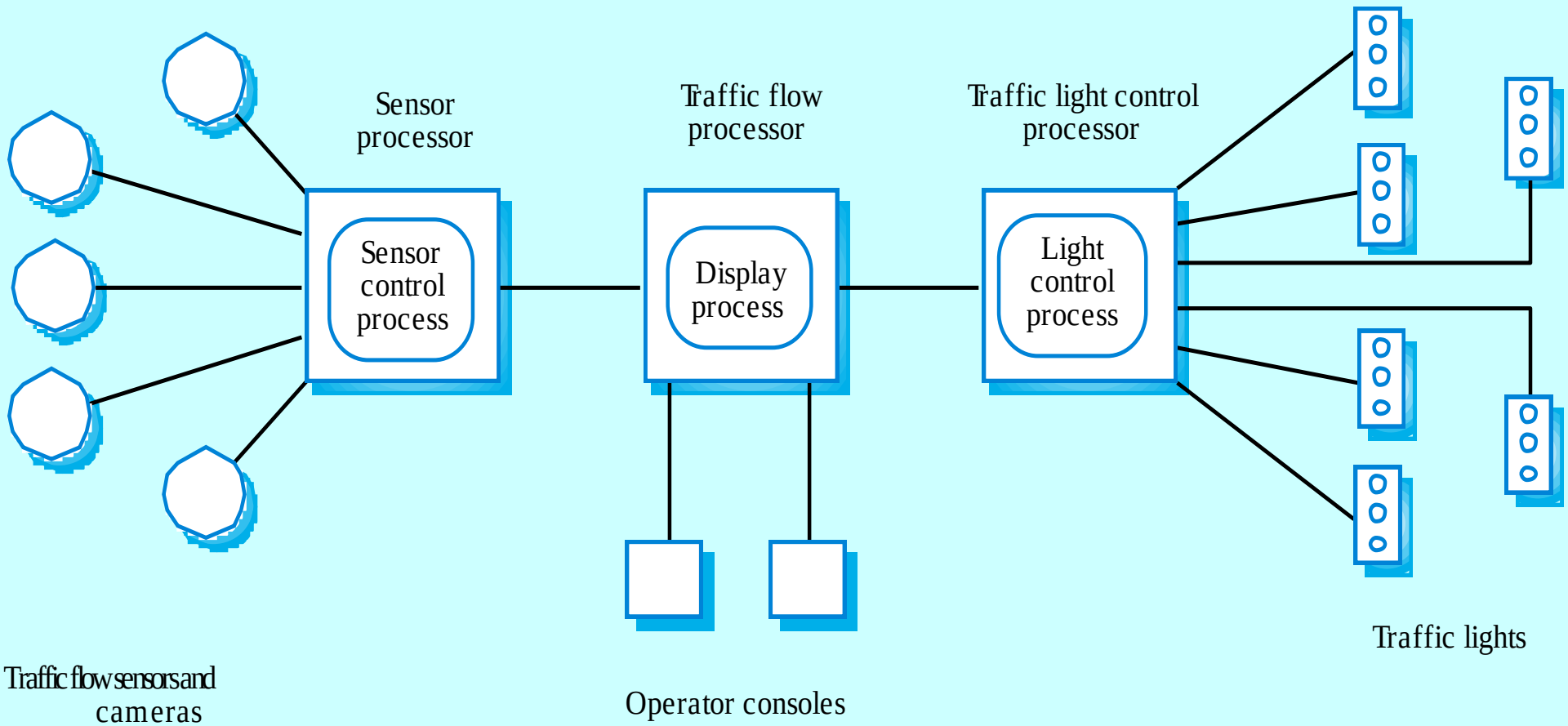
Middleware

- *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*
  - *either pre-ordered*
  - *or under the control of a dispatcher*.

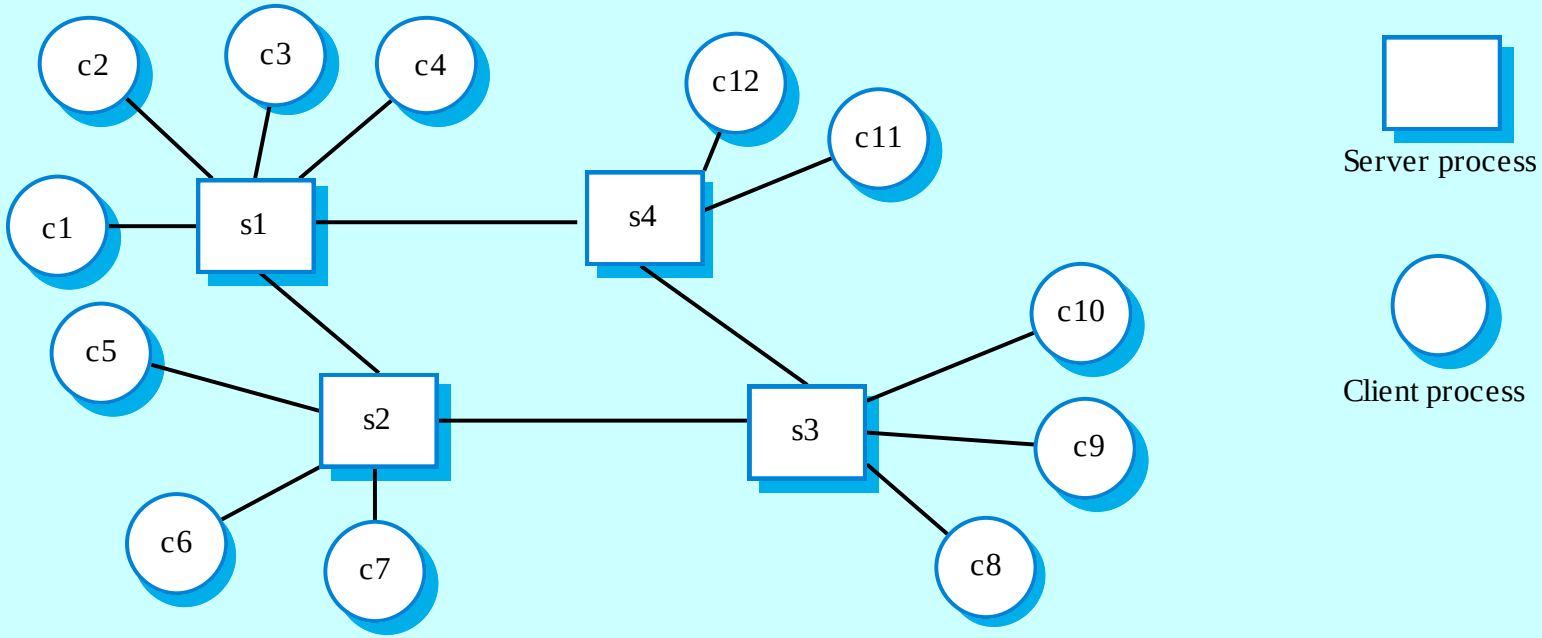
A multiprocessor traffic control system



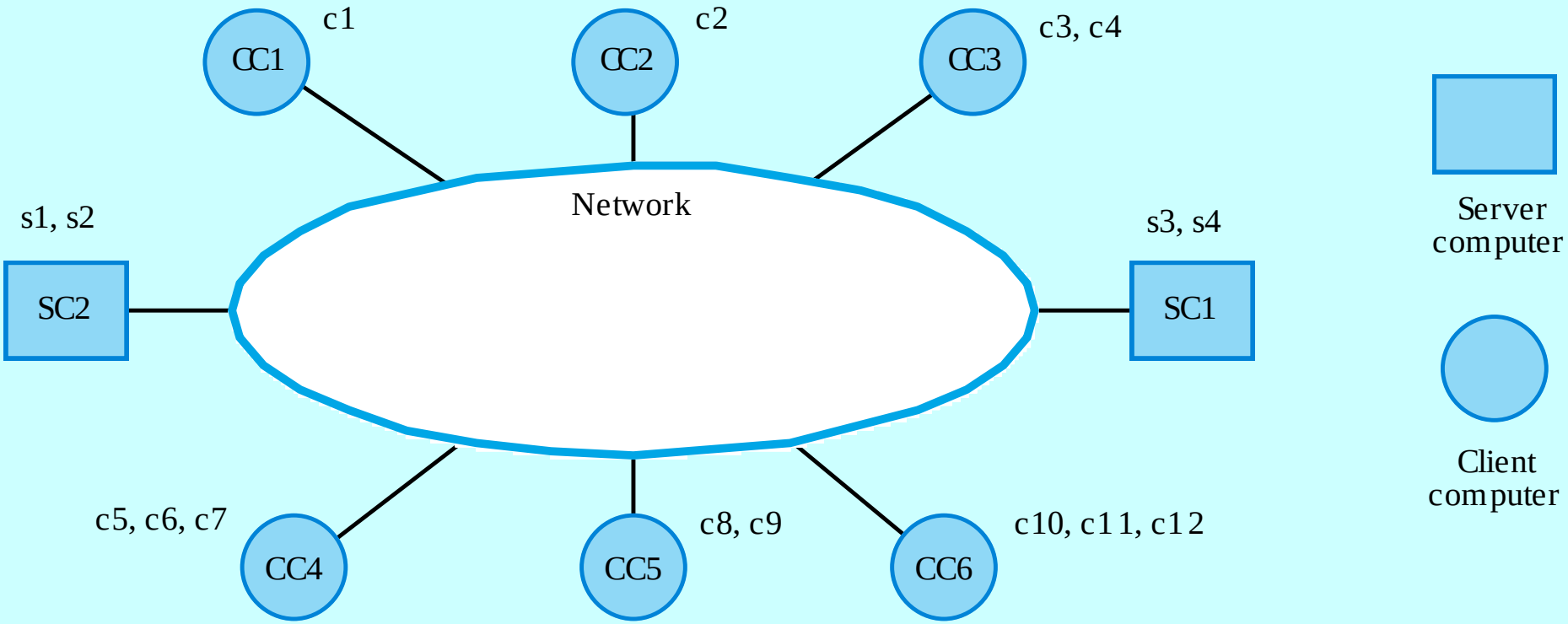
Client-server architectures

- 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



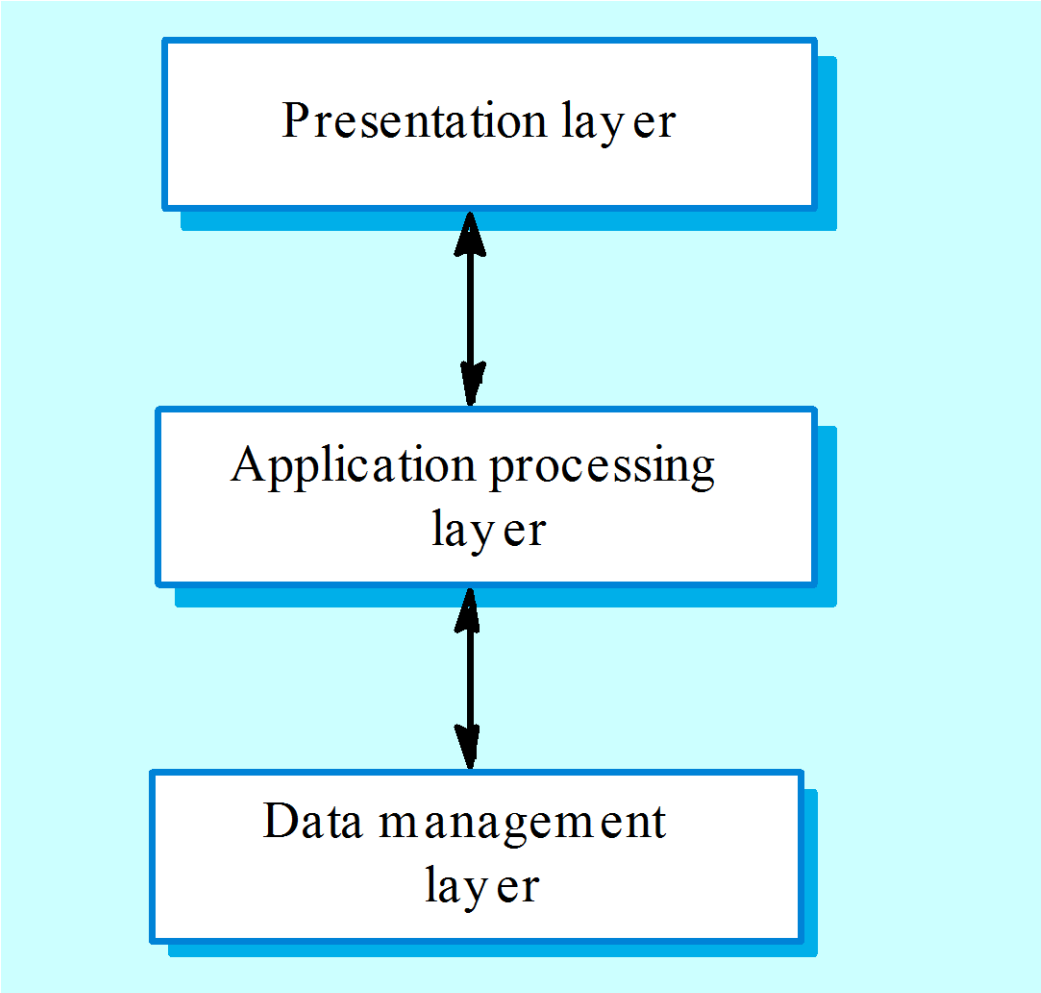
Computers in a C/S network



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*.

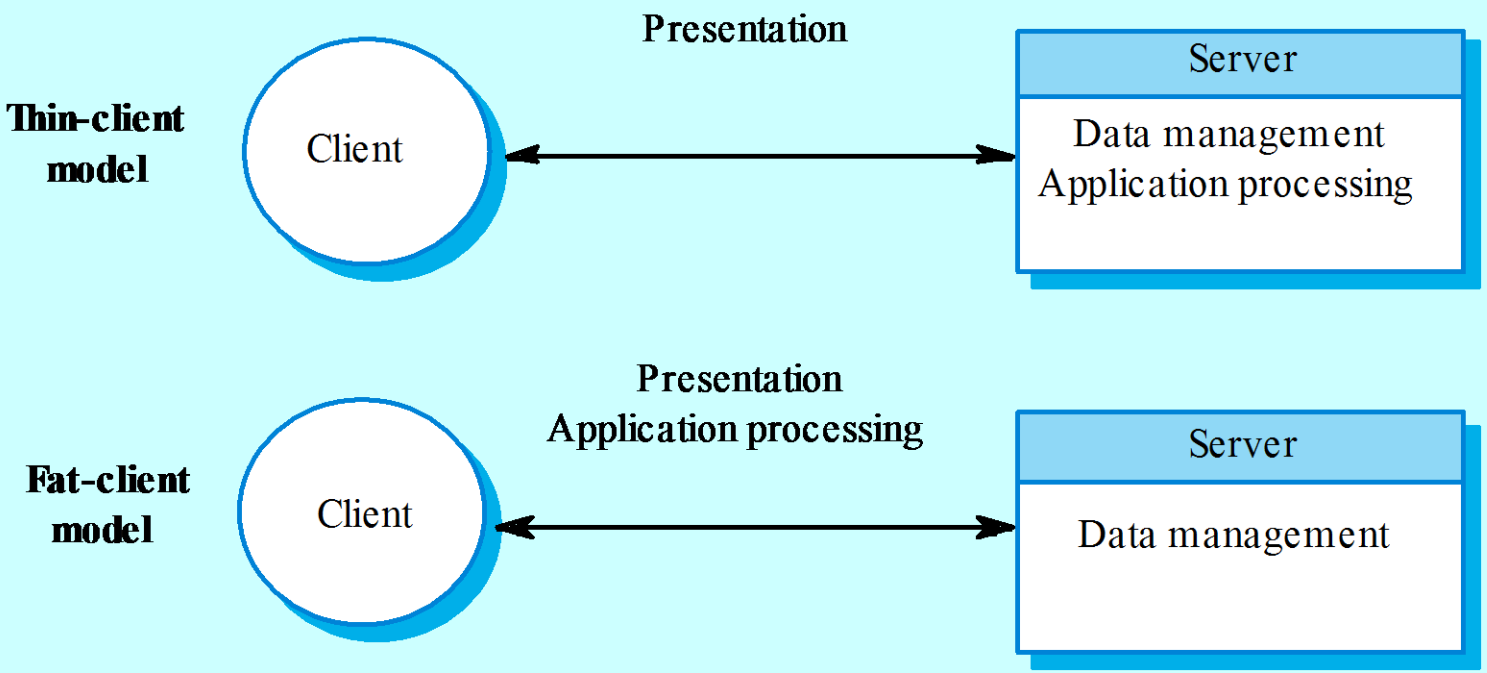
Application layers



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



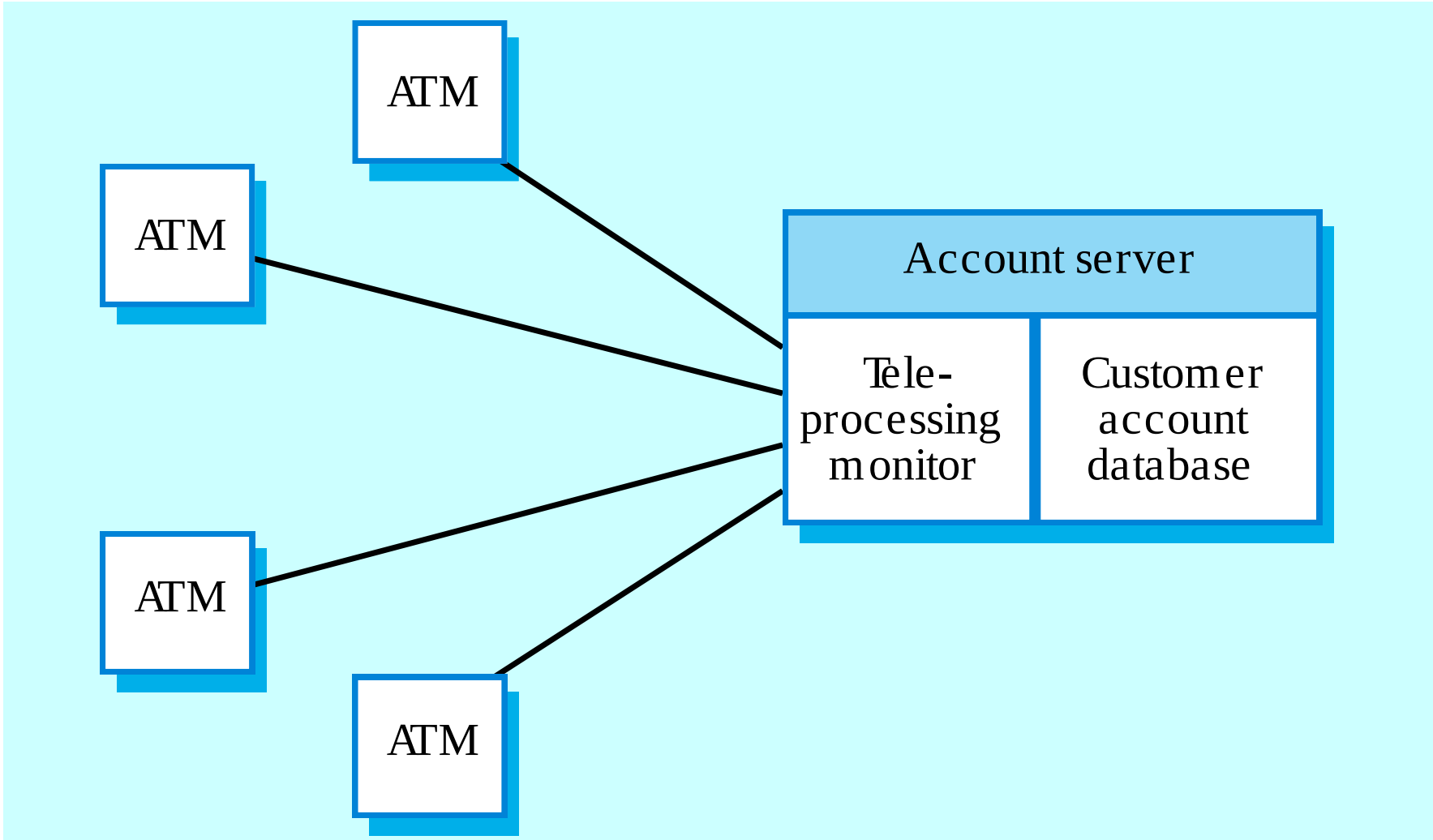
Thin client model

- 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 (performance & scalability issue)*.

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 (Maintainability issue)*.

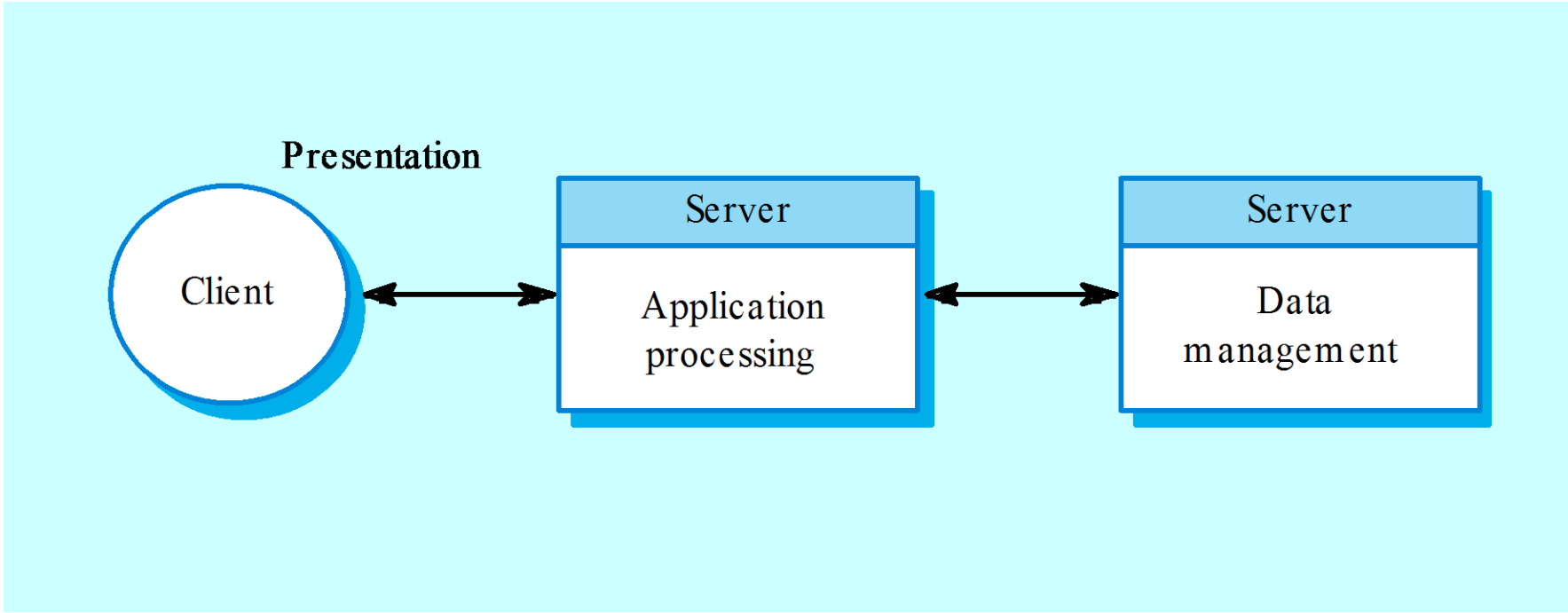
A client-server ATM system



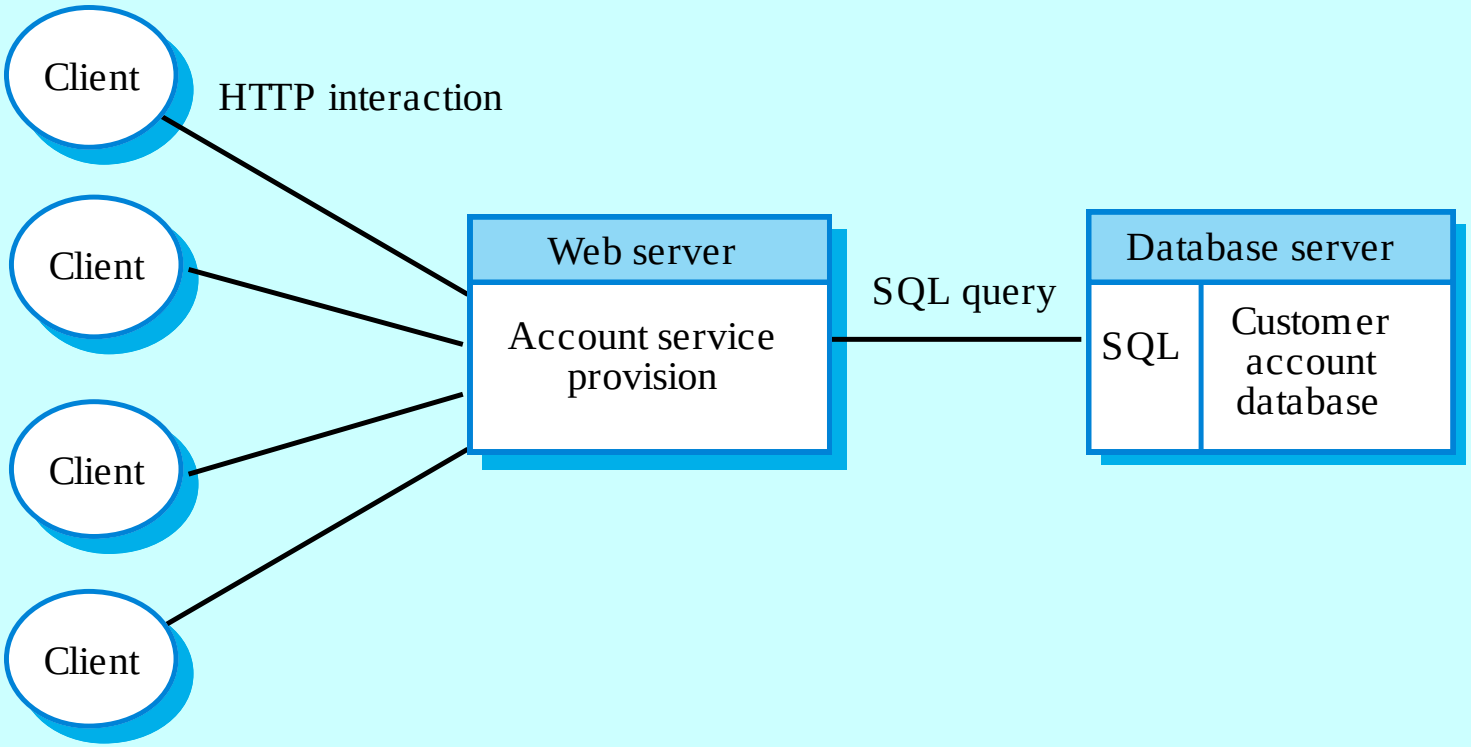
Three-tier architectures

- In a three-tier architecture, each of the application architecture *layers may execute on a separate processor*.
- Allows for *better performance and scalability 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



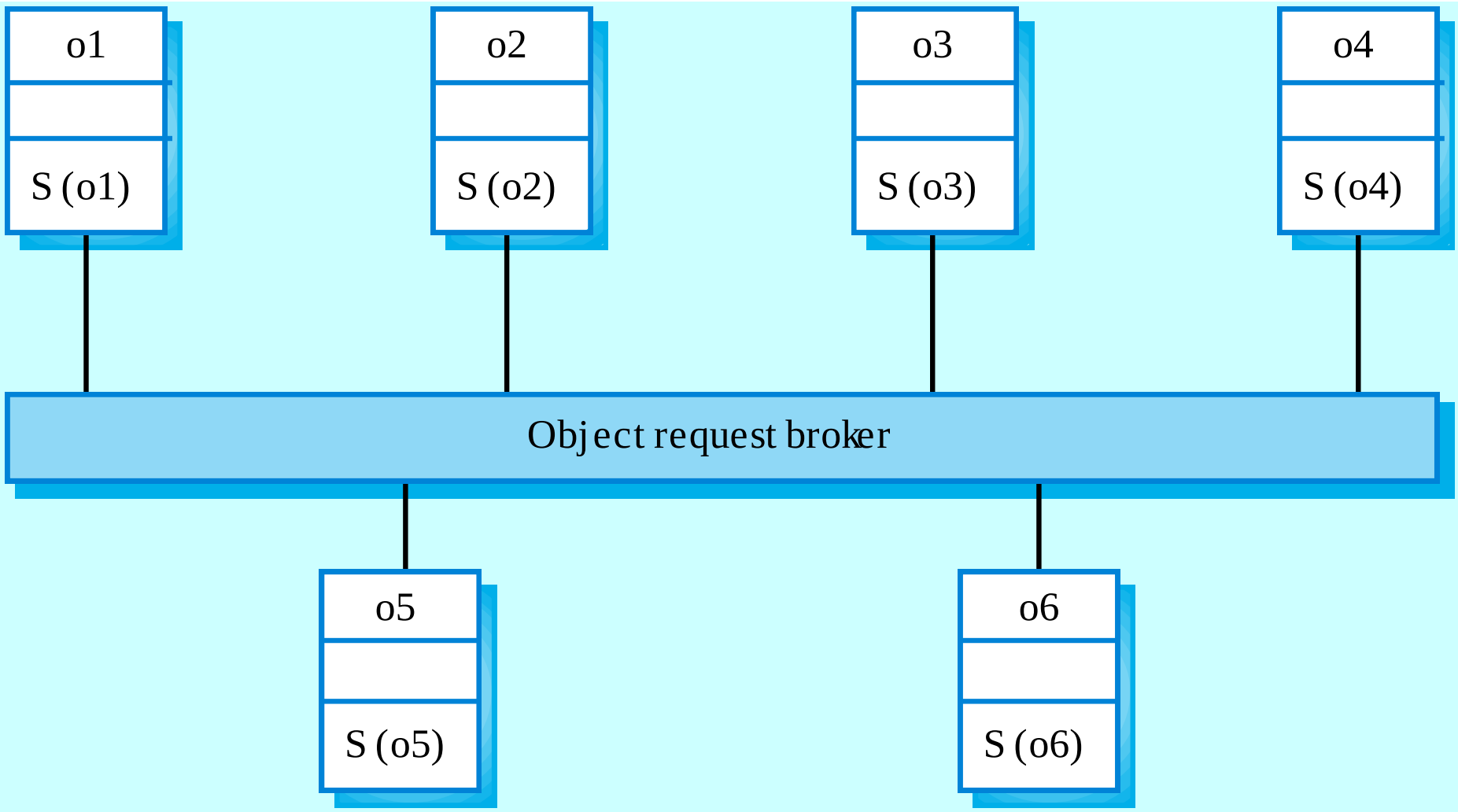
Use of C/S architectures

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



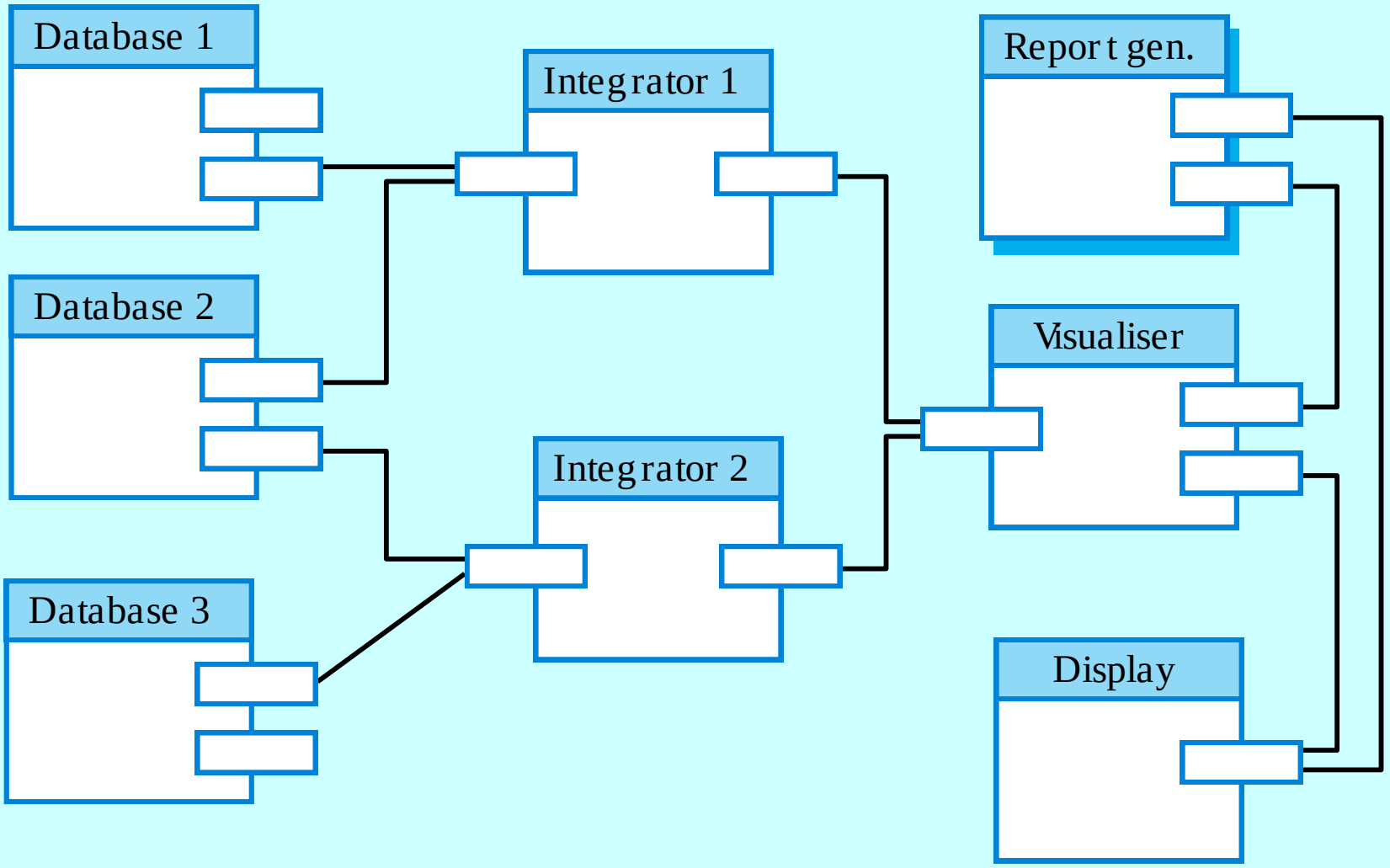
Advantages of distributed object architecture

- 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



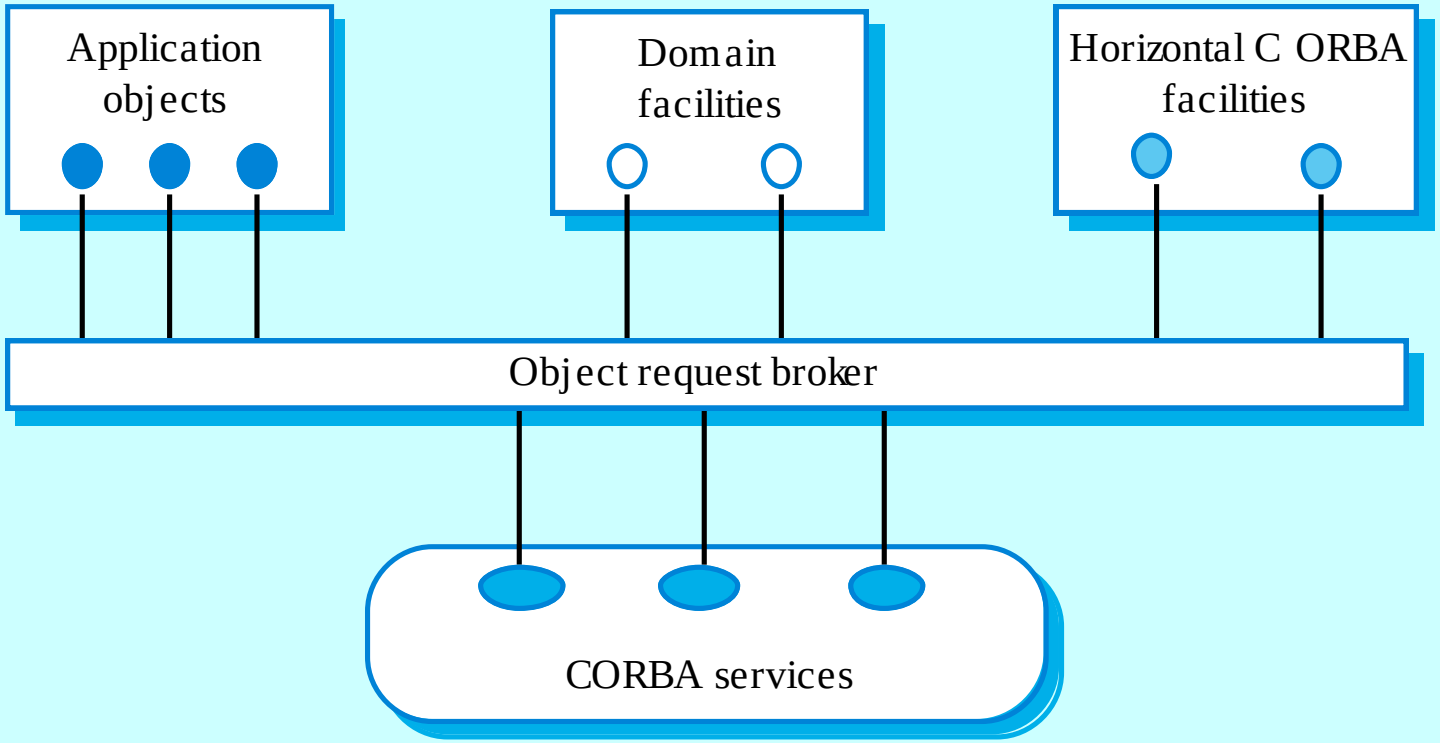
Data mining system

- 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



CORBA Services

- Domain-independent interfaces used by many distributed object programs.
- Example: a service providing for the discovery of other available services; almost always necessary regardless of the application domain.
- The Naming Service -- which allows clients to find objects based on names; ***white pages***
- The Trading Service -- which allows clients to find objects based on their properties; ***yellow pages***

CORBA Common Facilities

- Application Objects: Objects developed for the specific application.
- Domain Facilities: *Standard objects defined for a specific domain*. The domain object standards cover *finance/insurance*, *e-commerce*, *health care*, ... etc.
- Fundamental Corba Services: provide basic *distributed computing services such as directories and security management.*

CORBA Common Facilities

- Horizontally-oriented (i.e., *common to many application domain*), but unlike object services they are oriented towards end-user applications.
- Example: *Distributed Document Component Facility* (DDCF), a compound document Common Facility based on OpenDoc.
- DDCF allows for presentation and interchange of objects based on a document model, for example, facilitating the linking of a spreadsheet object into a report document.

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.

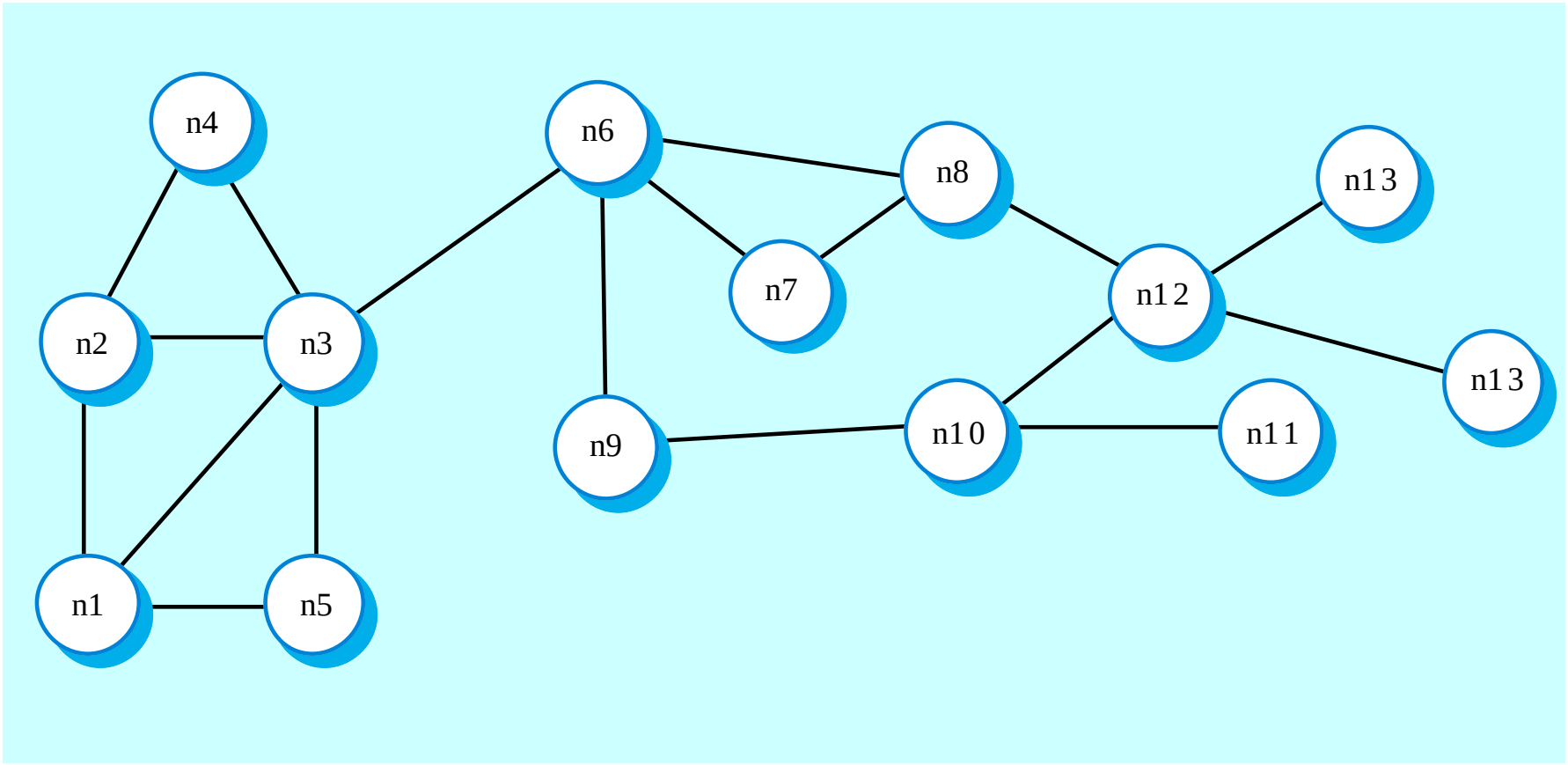
Examples to P2P architectures

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

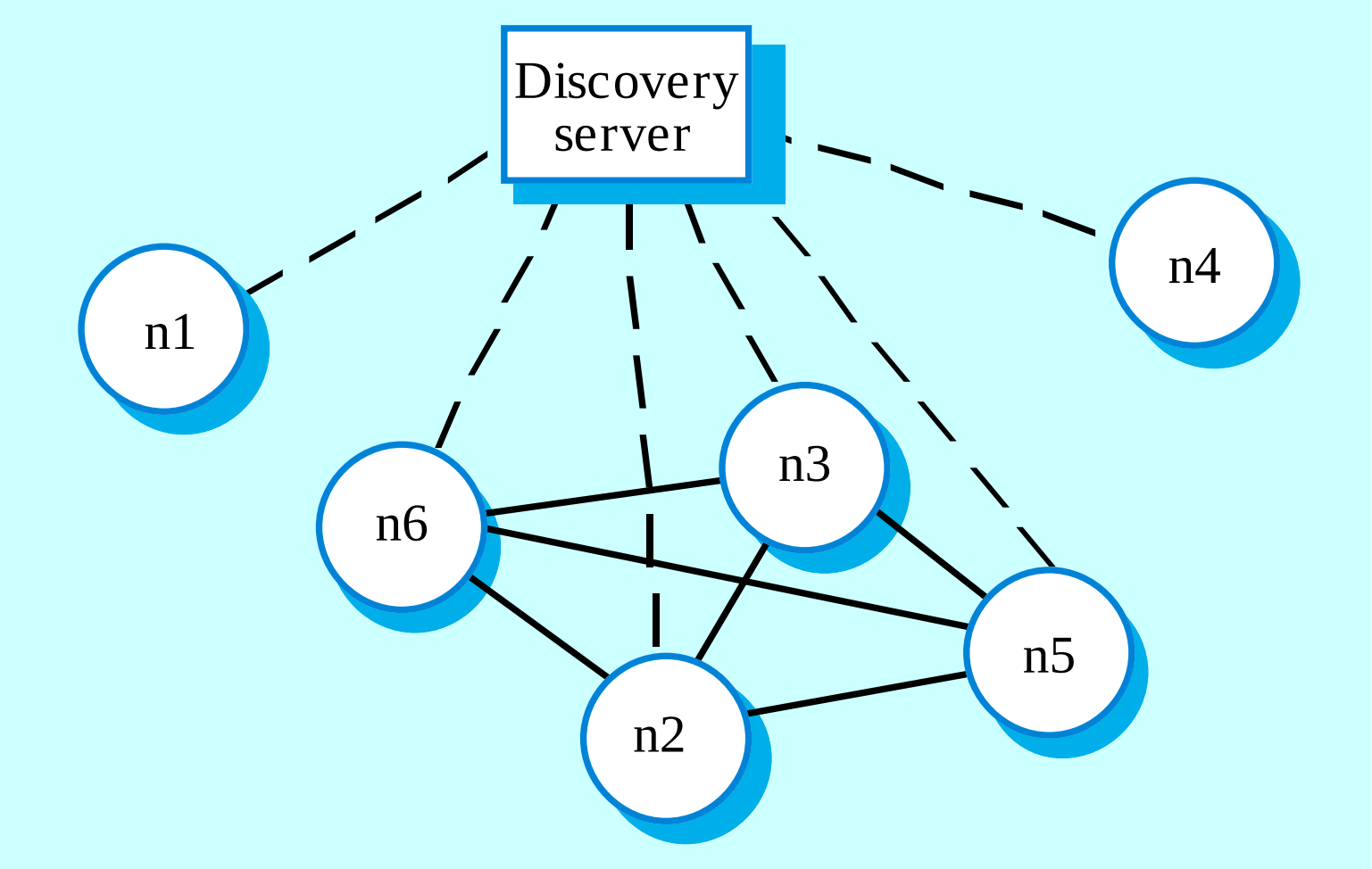
P2p architectural models

- 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



Real Time (RT) or Embedded System Design

Embedded SW - 1

- Computers are used to *control* a wide range of systems including
  - simple domestic (in-house) machines,
  - games controllers, or
  - *entire* manufacturing plants.

Embedded SW - 2

- Their software must
  - *react to events* generated by the hardware and, often,
  - *issue control signals* in response to these events.

Embedded SW - 3

- SW in these systems is
  - *embedded in system hardware (mostly in ROM)*, and
  - usually *responds, in real time, to events from the system's environment*.

Responsiveness - 1

- *the critical difference* between RT or embedded and other software systems, such as
  - information systems,
  - web-based systems or
  - personal software systems

  is this real time reaction to these events.

Responsiveness - 2

- For *non-RT* systems, *correctness can be defined by specifying:*
  - how system inputs *map* to corresponding outputs that should be produced by the system.

Responsiveness - 3

- *For a RT system, correctness depends both on:*
  - *the response to an input (as for non-RT systems),* ***and***
  - *the time taken to generate that response.*
- *A response taking too late may be ineffective* and *the system incorrect !!!*

Definition - 1

- A *real-time system (RTS)* is some SW where the correct functionality depends on the extent to which it generates
  1. *expected outputs from the inputs presented,* ***and***
  2. *the time at which these results are produced.*

Definition - 2

- A *soft real-time system (soft RTS)* is a system
  - *whose operation is* ***degraded***
  - if results are not produced according to the specified timing requirements.

Definition - 3

- A *hard real-time system (hard RTS)* is a system
  - whose *operation is* ***incorrect***
  - if results are not produced according to the timing specification.

Characteristics of RT Systems

- generally *run continuously* and *do not terminate*.
- *Unpredictably interact* with the system's environment.
- *physical limitations may exist* that affect RTS design (e.g *military specifications*).
- *Direct hardware interaction* may be necessary (e.g. Hard reset or other calibration switches).
- *Issues of safety and reliability may dominate the system design* (*i.e., RTSs are critical systems !!!*)

RT Systems Design

- The design process for RTSs must *consider*, *in detail*, *the design and performance of the system hardware (a systems engineering process)*.
- Part of the design process may involve *deciding which system capabilities to implement in software and which in hardware*.
- *Low-level decisions on hardware, support software and system timing must be considered early in the process*.
- These may mean that *additional software functionality, such as battery and power management, has to be included in the system*.

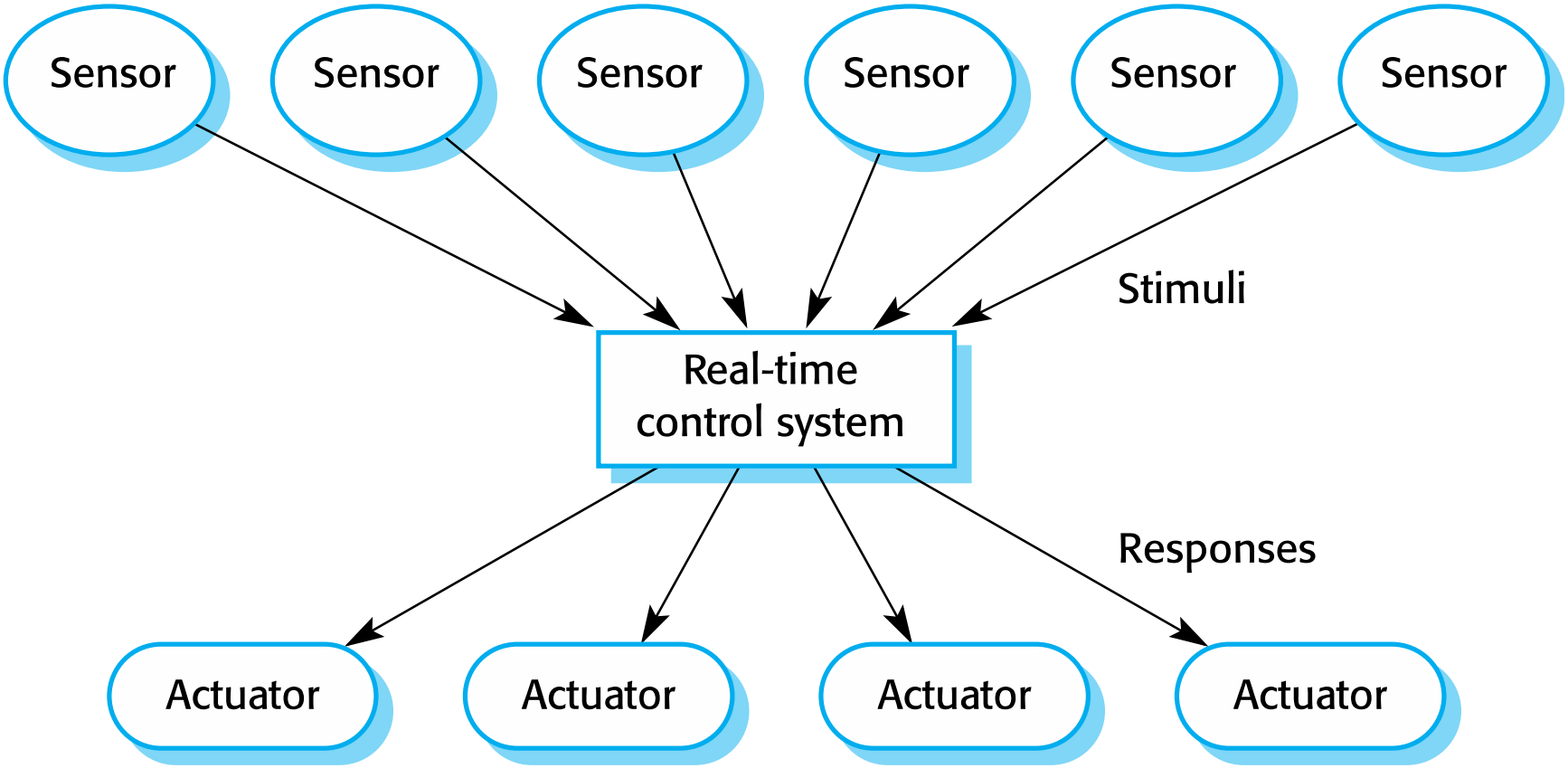
Reactive Systems

- *RTSs* are often considered to be *reactive systems*. Given a stimulus, the system must produce a reaction or response within a specified time.
- *Periodic stimuli*. Stimuli which *occur at predictable time intervals* (*e.g.*, a temperature sensor may be polled 10 times per second.)
- *Aperiodic stimuli*. Stimuli which *occur at unpredictable times* (*e.g.*, a system *power failure* may *trigger an interrupt which must be processed by the system*.

Stimuli and Responses for a Burglar Alarm System

Stimulus	Response
Clear alarms	*Switch off all active alarms; switch off all lights that have been switched on.*
Console panic button positive	*Initiate alarm; turn on lights around console; call police.*
Power supply failure	*Call service technician.*
Sensor failure	*Call service technician.*
Single sensor positive	*Initiate alarm; turn on lights around site of positive sensor.*
Two or more sensors positive	*Initiate alarm; turn on lights around sites of positive sensors; call police with location of suspected break-in.*
Voltage drop of between 10% and 20%	*Switch to battery backup; run power supply test.*
Voltage drop of more than 20%	*Switch to battery backup; initiate alarm; call police; run power supply test.*

A General Model for an Embedded RTS



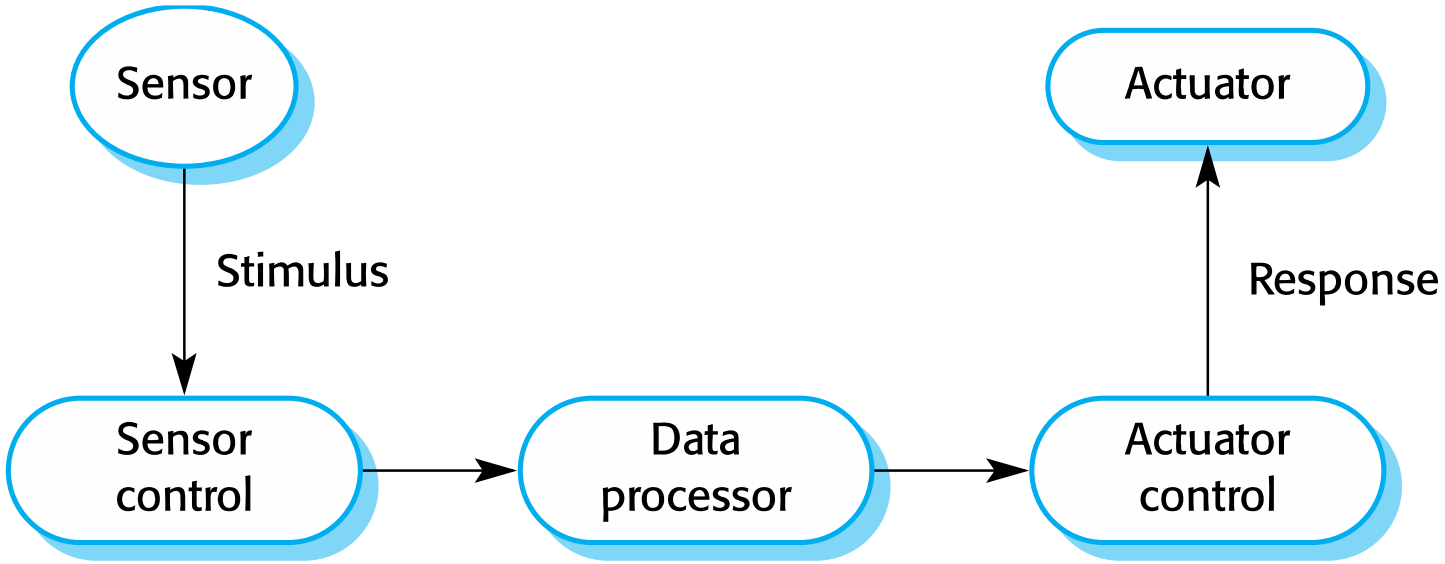
Architectural Considerations

- To *meet the timing demands* made by different stimuli/responses, the system *architecture must allow for fast switching between stimulus handlers*.

- *Timing demands of different stimuli are different* so a simple *sequential loop* is *not usually adequate*.

- *RTSs* are therefore usually *designed as cooperating processes* with *a real-time executive controlling* these processes.

Sensor and Actuator Processes



Architectural Considerations

- *Sensor control processes*
  - *Collect information* from sensors. *May buffer information collected* in response to a sensor stimulus.
- *Data processor*
  - *Carries out processing of collected information and computes the system response.*
- *Actuator control processes*
  - *Generate control signals for the actuators.*

Design process activities

- Platform selection
- Stimuli/response identification
- Timing analysis
- Process design
- Algorithm design
- Data design
- Process scheduling

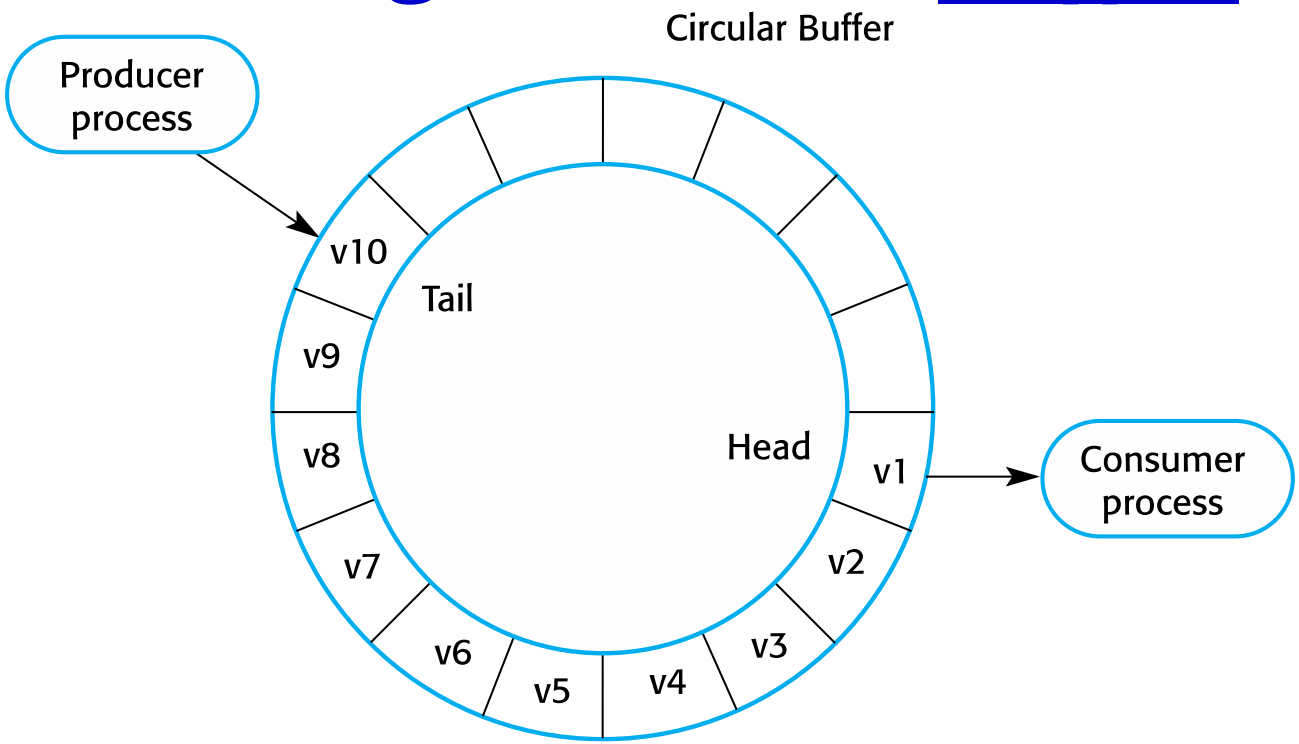
Process coordination

- *Processes* in a RTS have *to be coordinated and share information*.
- *Process coordination mechanisms ensure mutual exclusion to shared resources*.
- *Only one process is allowed to modify a shared resource at a given time*, other processes should wait.
- *When designing the information exchange between processes*, you have to *take into account* the fact that these *processes may be running at different speeds*.

Mutual exclusion

- *Producer (sensor control) processes collect data and add it to the **buffer**. Consumer (actuator control) processes take data from the **buffer** and make elements available.*
- *Producer and consumer processes must be mutually excluded from accessing the same element.*
- *The buffer must stop producer processes adding information to a full buffer and consumer processes trying to take information from an empty buffer.*

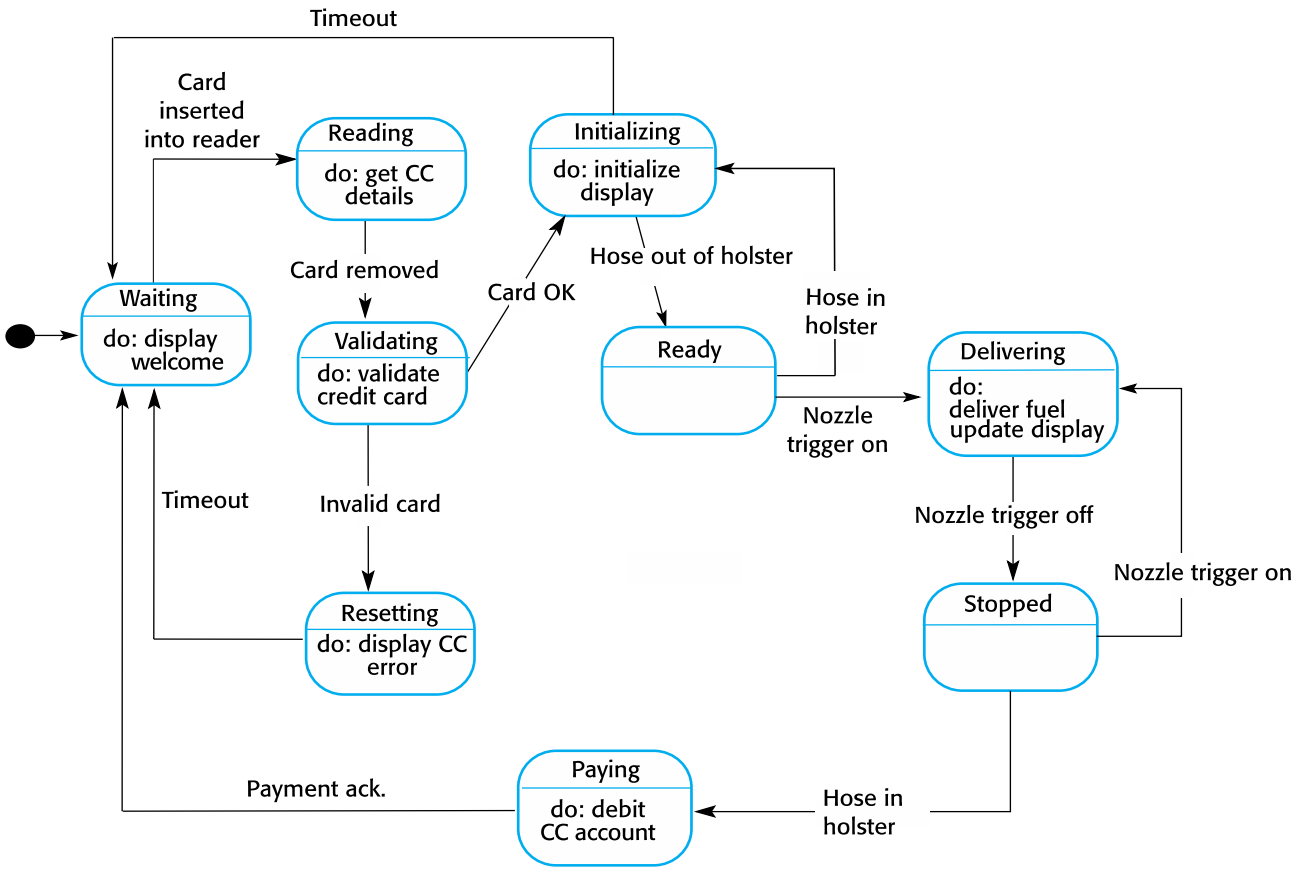
Producer/consumer processes sharing a circular ***buffer***



Real-time system modelling

- The *effect of a stimulus* in a RTS may trigger *a transition from one state to another*.
- *State models* are *therefore often used to describe embedded RTSs*.
- *UML state diagrams* may be used to *show the states and state transitions in a RTS*.

State machine model of a petrol (gas) pump



Sequence of actions in real-time pump control system

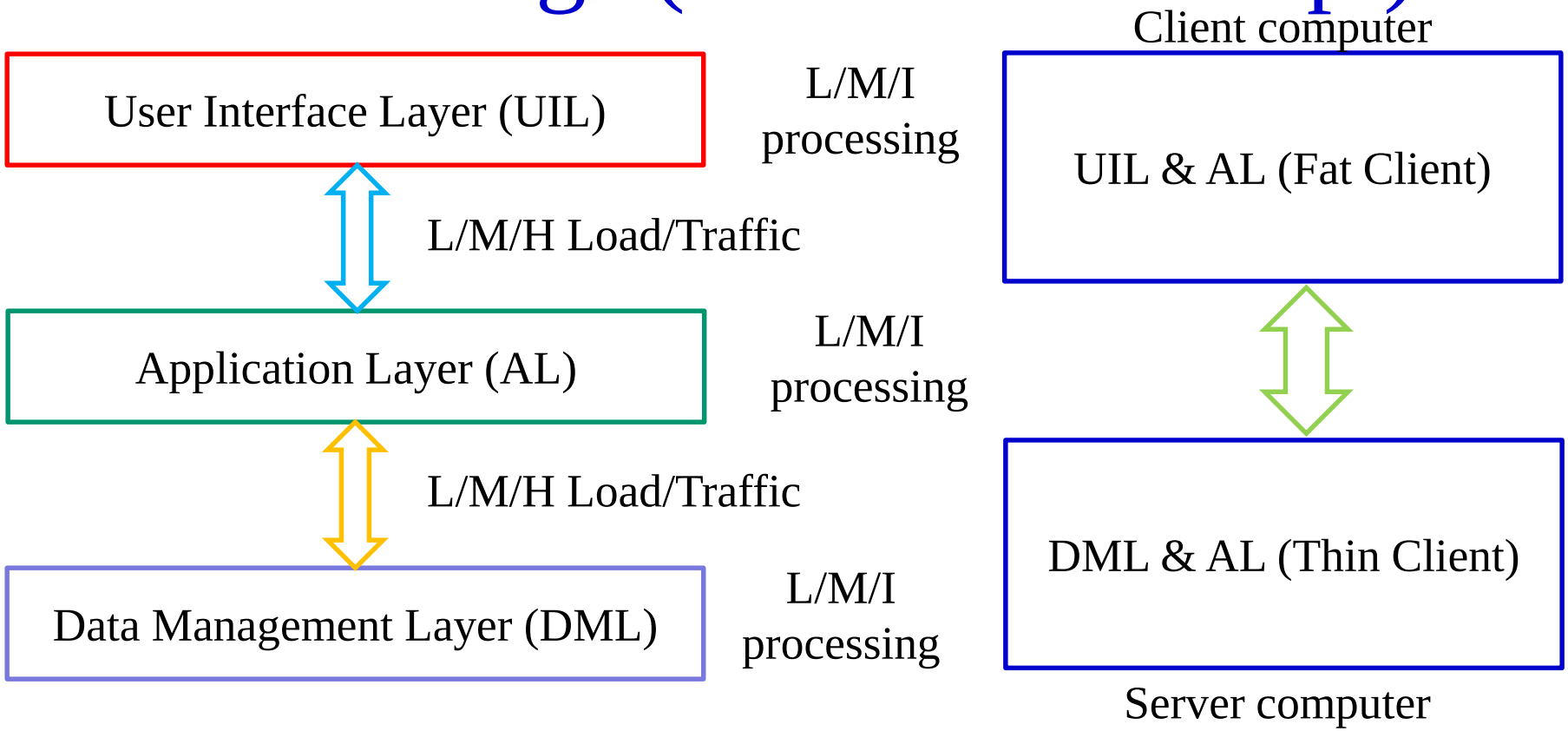
- The buyer *inserts a credit card* into a card reader built into the pump.
- *Removal of the card* triggers a *transition to a **validating** state where the card is validated*.
- *If the card is valid, the system initializes the pump* and, *when the fuel hose is removed from its holster, transitions to the **delivering** state*.
- After the fuel *delivery is complete* and the *hose replaced in its holster*, the *system moves to a **paying** state*.
- *After payment, the pump software returns to the **waiting** state*

Real-time programming

- *Programming languages for RTS development* have to include *facilities to access system hardware*. This *makes the prediction of the timing of particular operations possible* in these languages.
- *Systems-level languages*, such as ***C*** *allowing efficient code generation* are widely *used in preference to* languages like *Java*.
- There is a *performance overhead in object-oriented systems because extra code is required to mediate access to attributes and handle calls to operations*. The *loss of performance may make it impossible to meet real-time deadlines*.

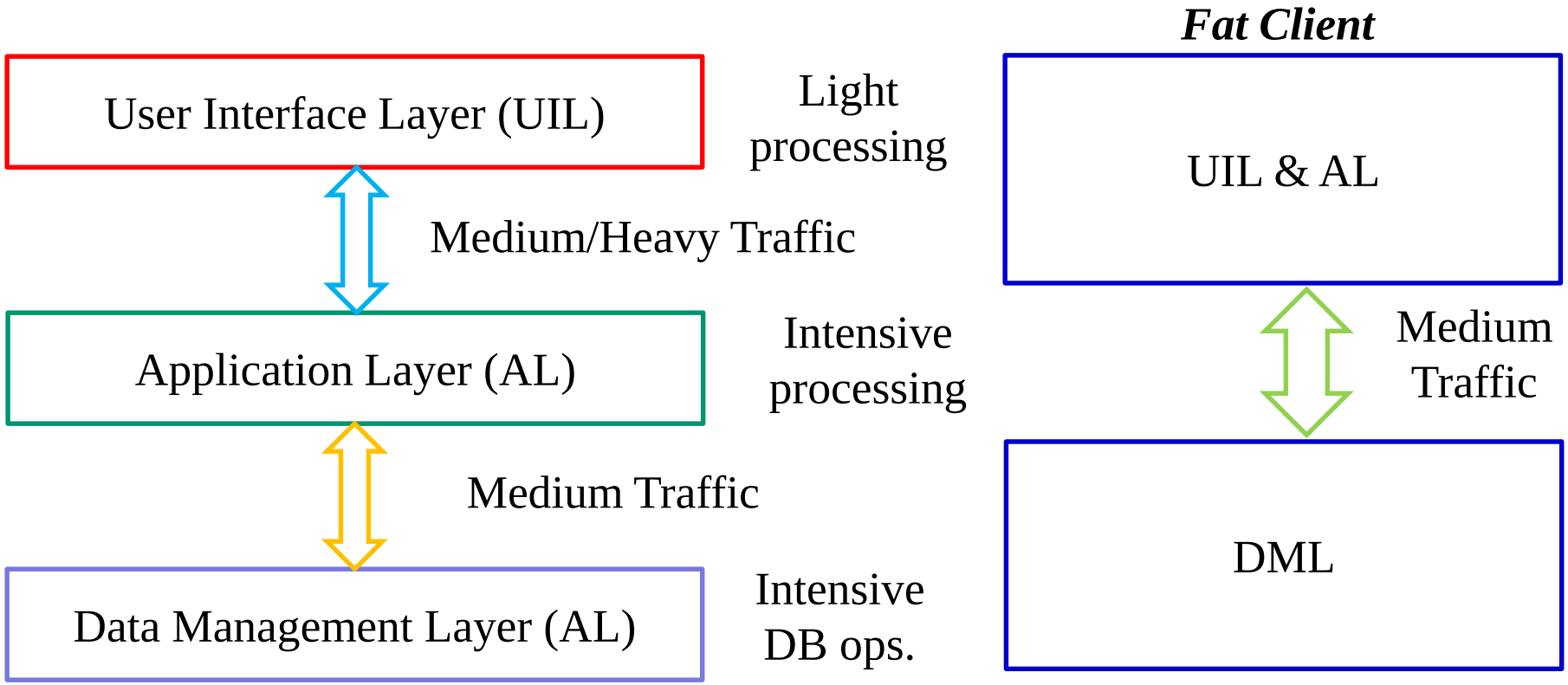
Supplementary Slides

Load/Processing Patterns versus C/S Settings (Performance v.p.)



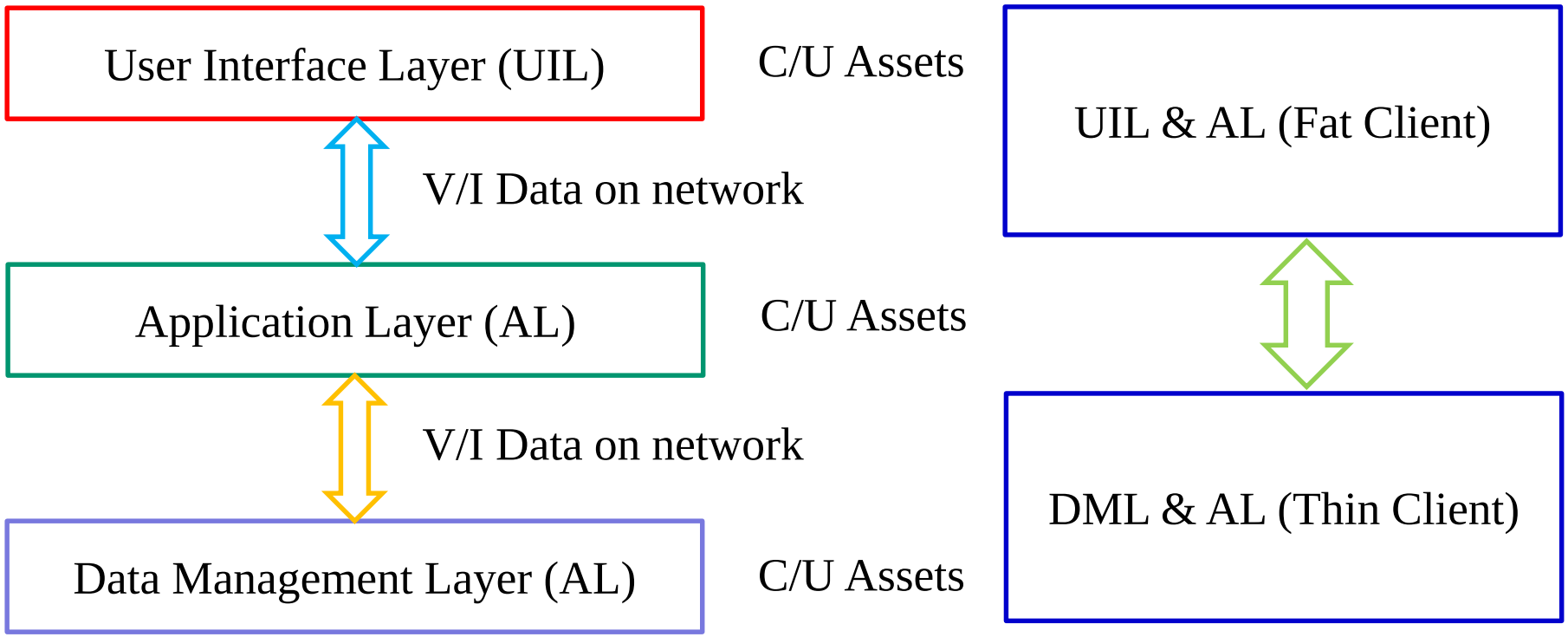
L: Light; M: Medium; H: Heavy; I: Intensive

Example Load/Processing Patterns for Fat Client Arch.



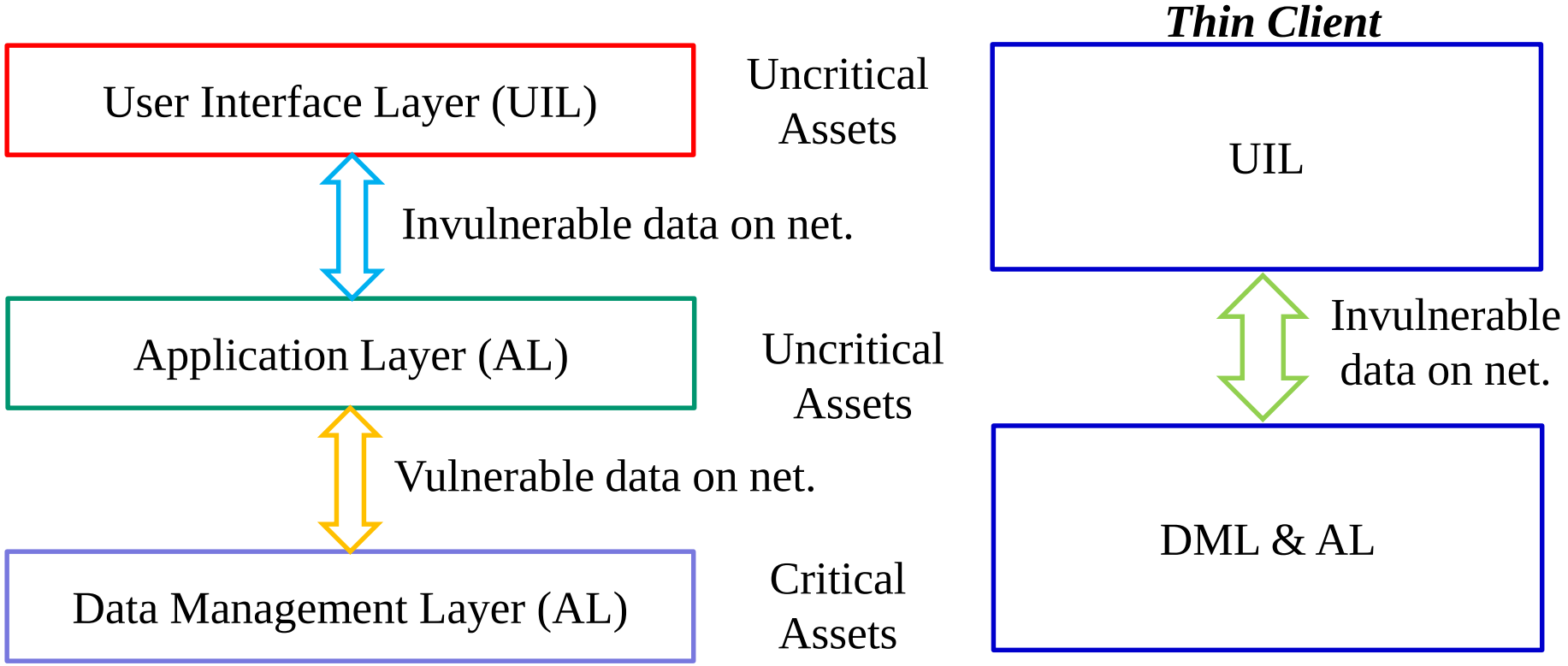
L: Light; M: Medium; H: Heavy; I: Intensive

Criticality/Vulnerability Patterns versus C/S Settings (Security vp).



C: Critical; U: Uncritical; V: Vulnerable; I: Invulnerable

Example Criticality/Vulnerability Patterns vs. for Thin Client Arch.



C: Critical; U: Uncritical; V: Vulnerable; I: Invulnerable