Politecnico
di Milano

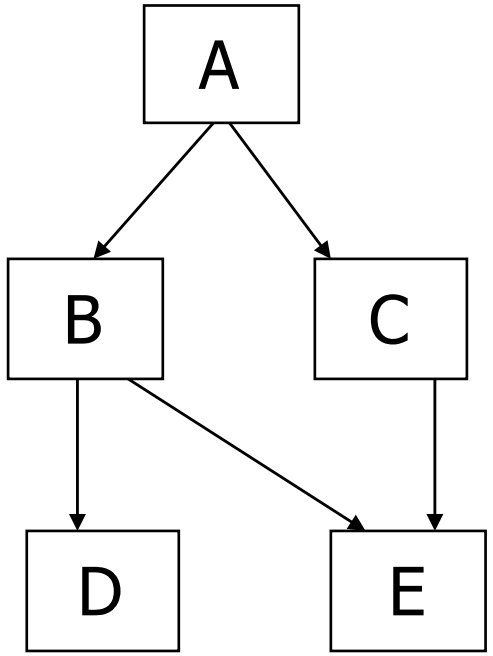
Architectural styles

Style

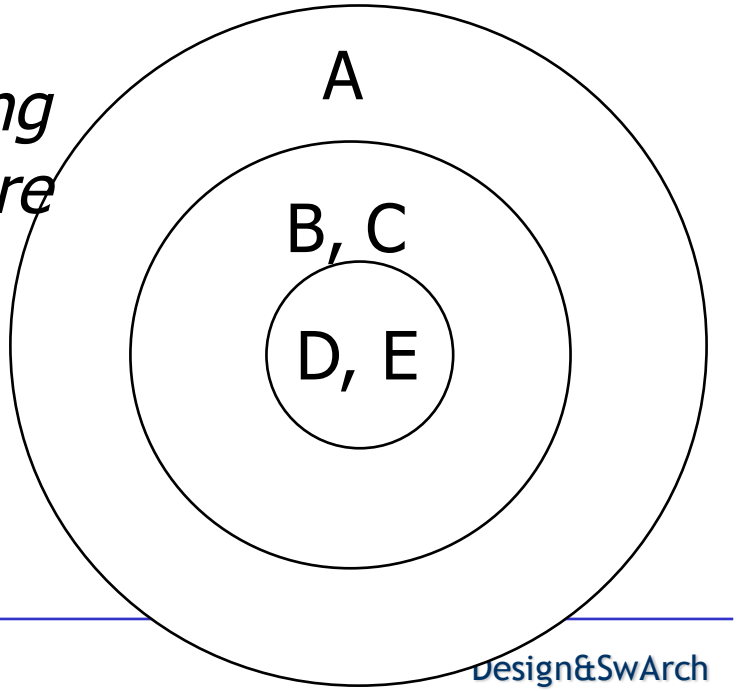
Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions among components can be simple and familiar, such as procedure call and shared variable access. But they can be complex and semantically rich, such as client-server protocols, asynchronous event multicast, and piped streams. (Garlan&Shaw 1996)

Layered style

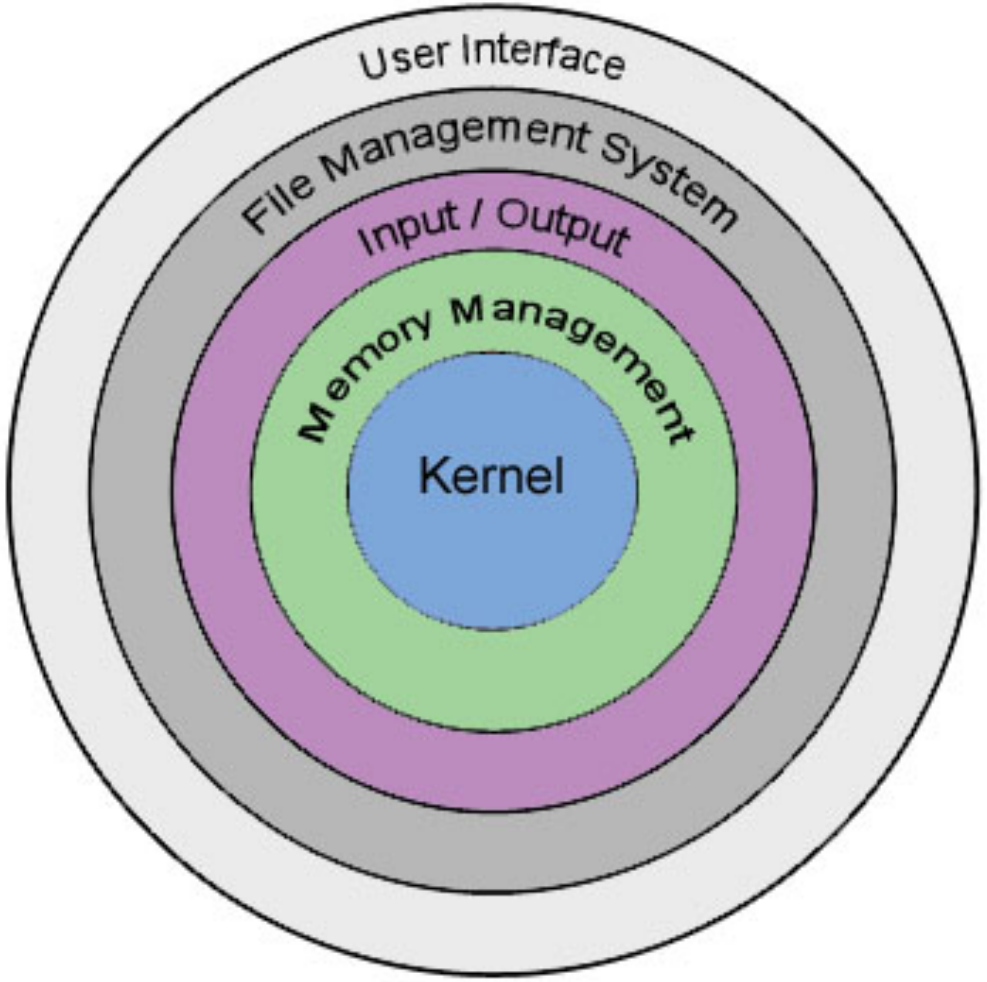
- The system is organized through abstraction levels, as a hierarchy of abstract machines
- Hierarchy is given by the USE relation



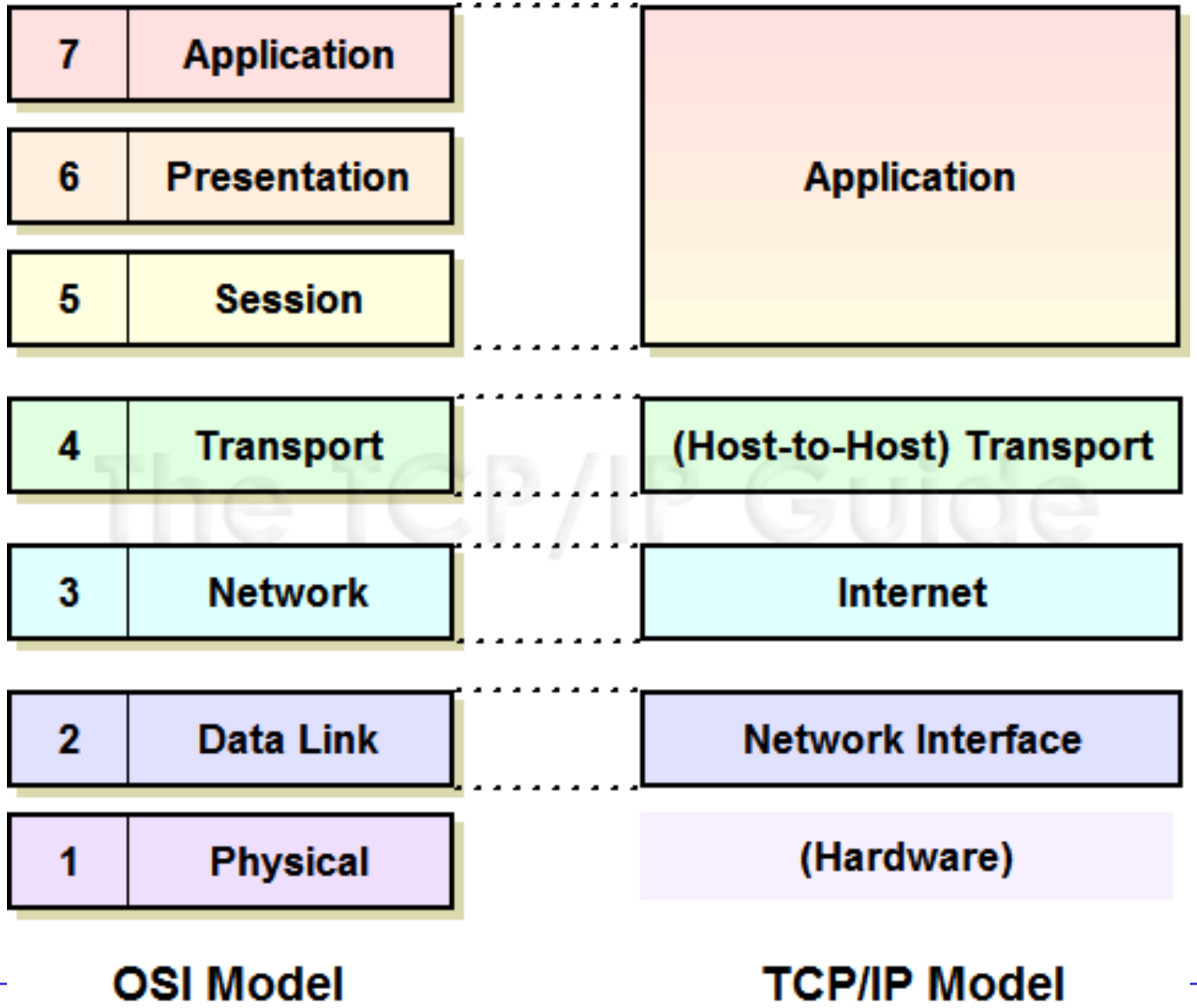
the onion-ring structure



Well-known layered systems: An Operating System



Well-known layered systems: The OSI and TCP/IP models



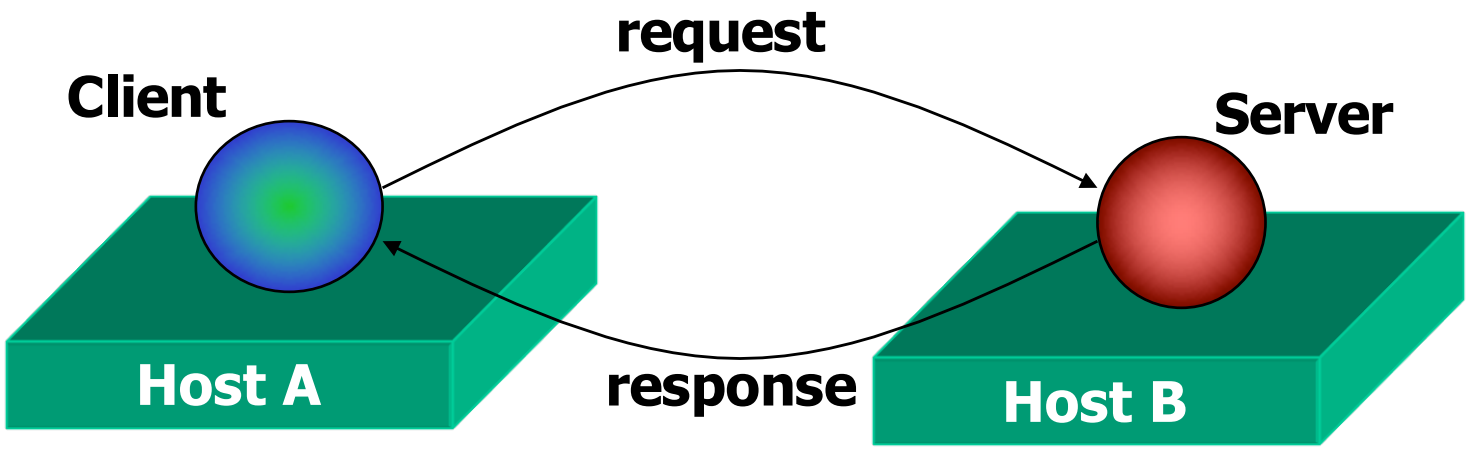
Client/Server

- The best known and most used architectural style for distributed applications

Client/Server

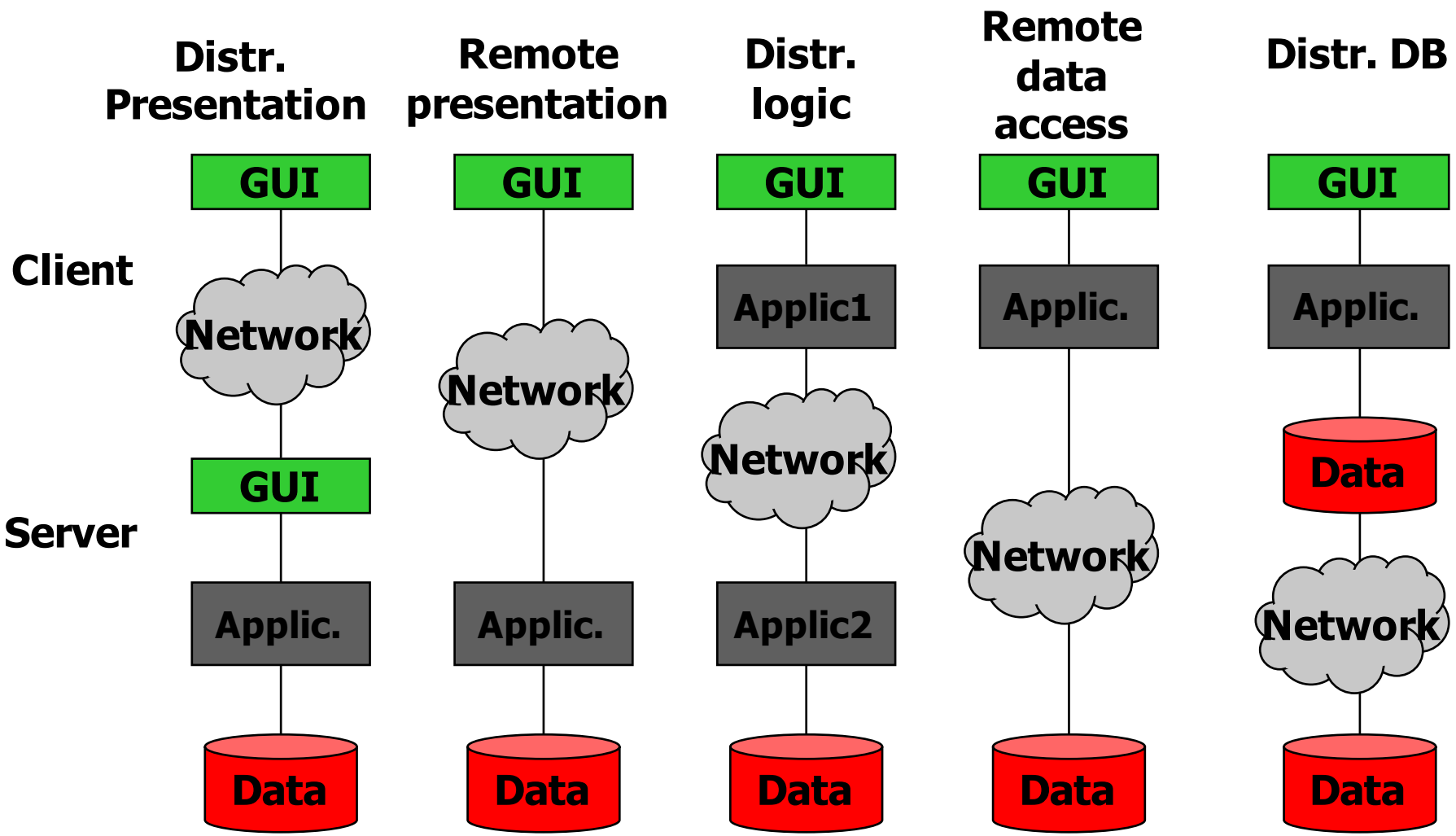
- Client & server are different processes with well-defined interface
  - Accessible only through interface
  - Client & server may be defined by a set of hw/sw modules
- Client & server play different roles
  - Server invoked to provide one of a set of services
  - Client uses the provided services and initiates communication
  - Comunication through messages or through remote invocations

C/S: a general scheme



Two-tier architecture

NOTE: also here we have layers!

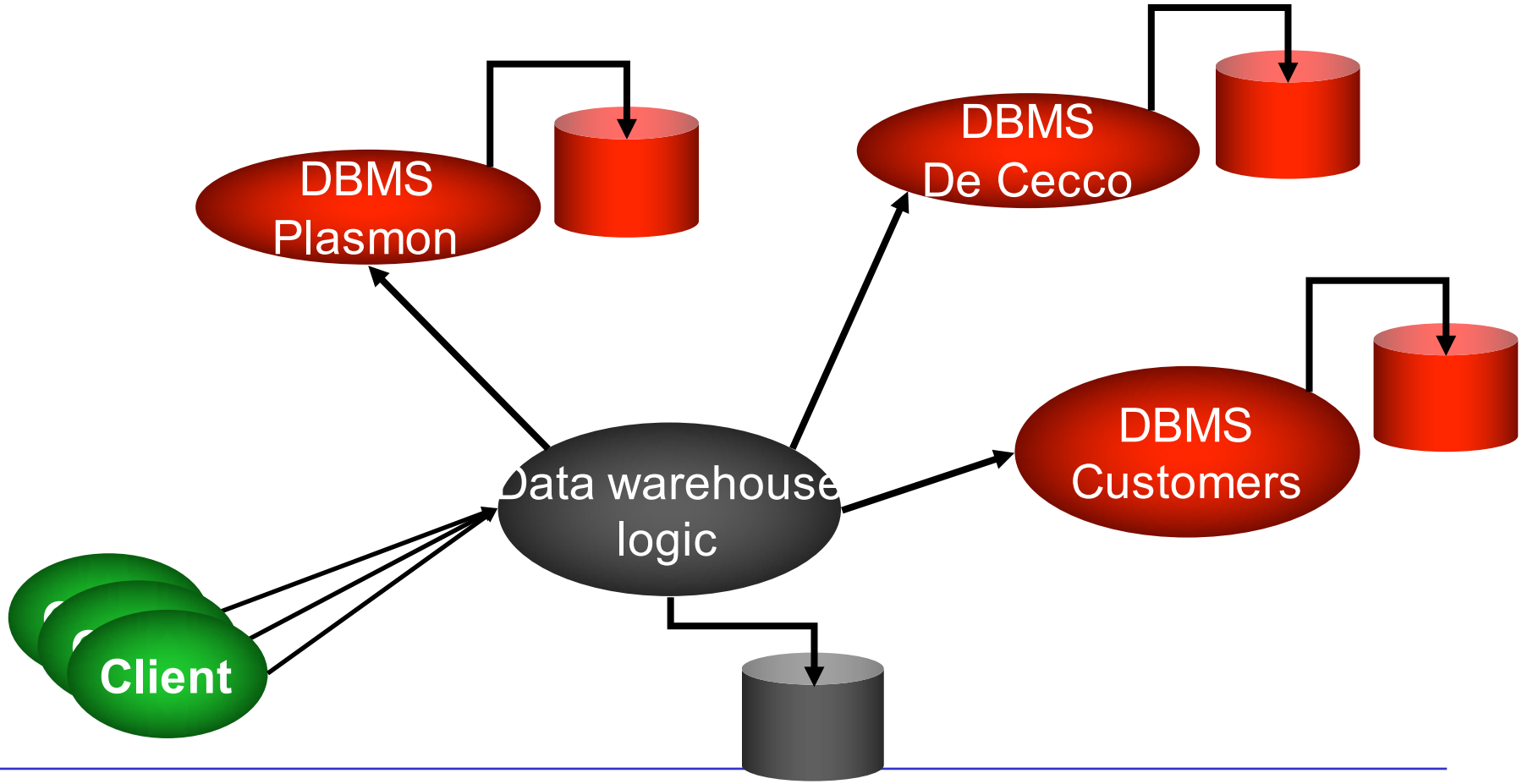


Three-tier architecture

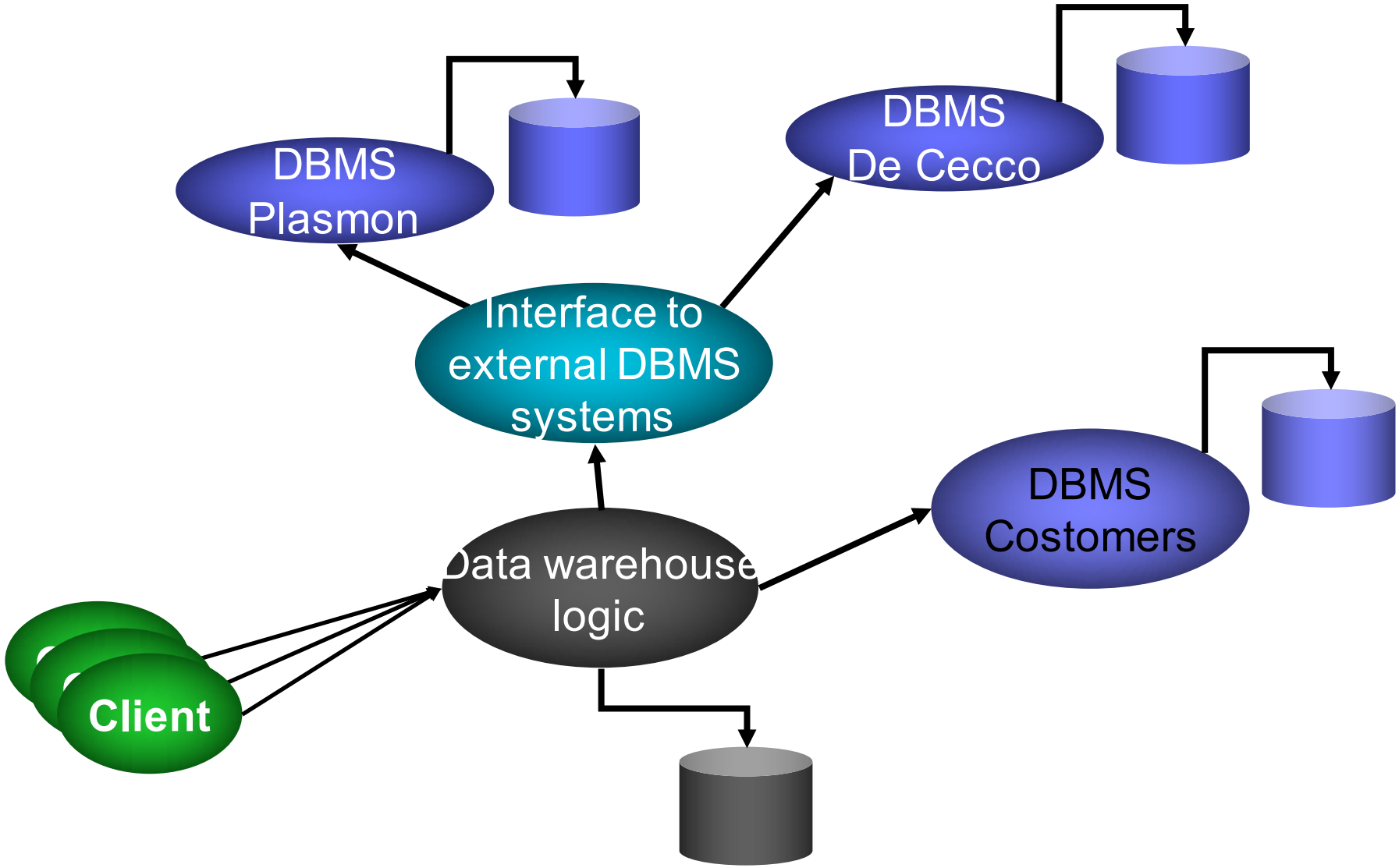
- In the typical scheme, the mid level has all the application logic
- Advantage
  - decoupling of logic and data, logic and presentation

An example

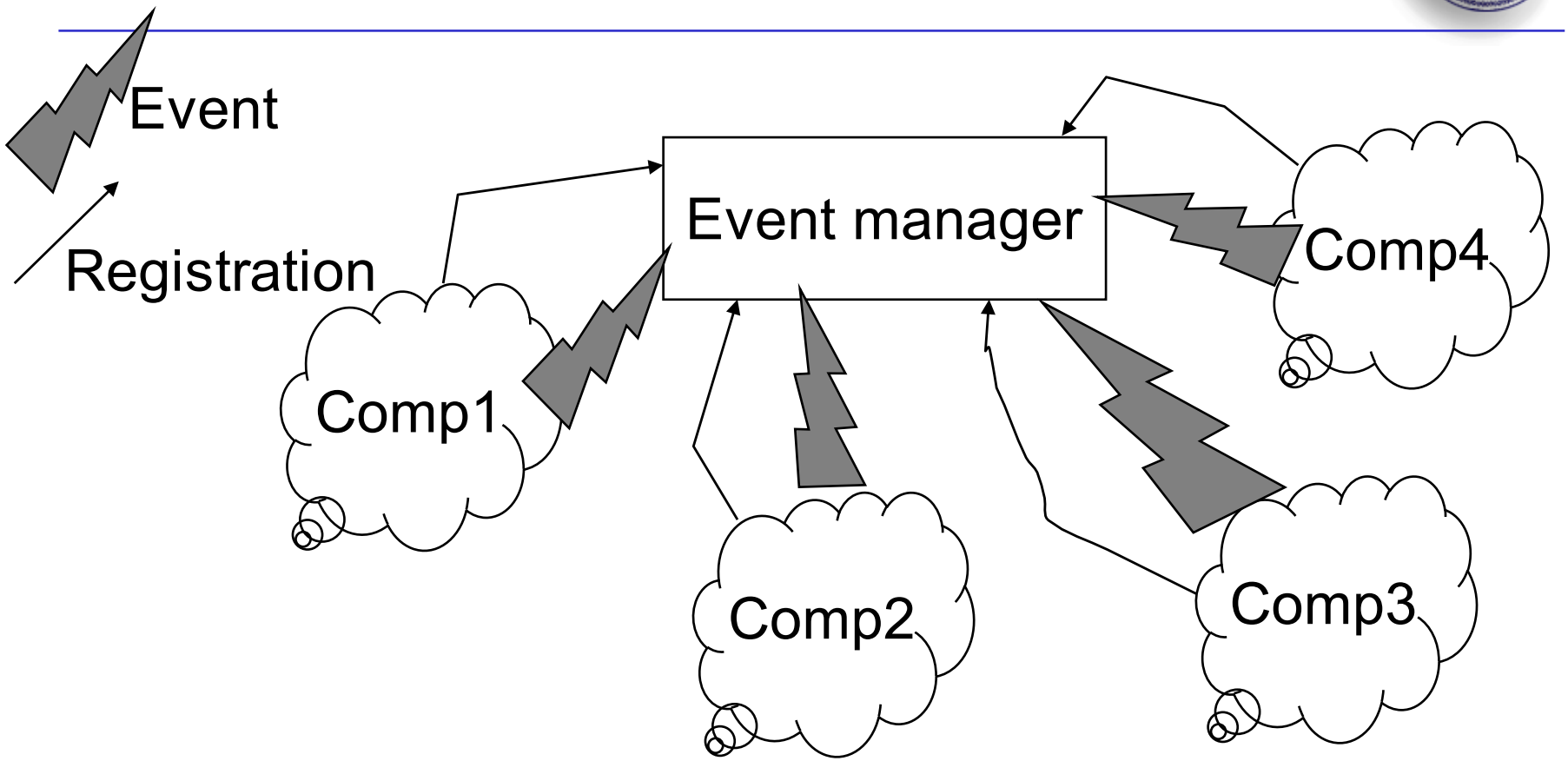
Data warehouse for a supermarket



From 3 to n levels



Event-based systems



- Events are broadcasted to all registered components
- No explicit naming of target component

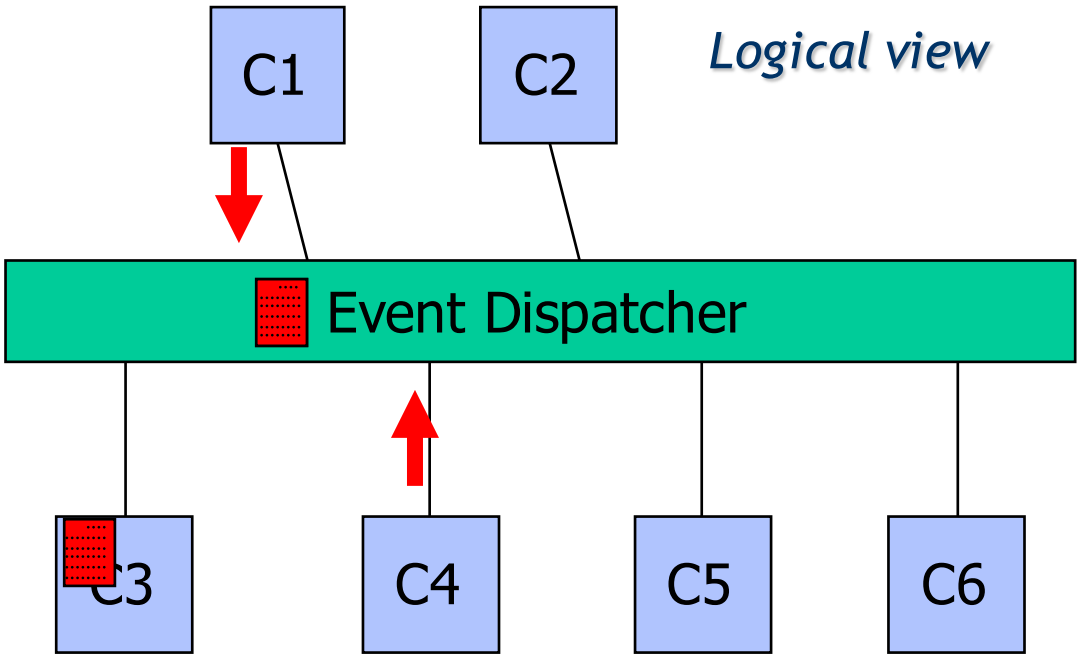
Event-based paradigm

- Often called *publish-subscribe*
- Publish
  - event generation
- Subscribe
  - declaration of interest
- Different kinds of event languages possible

Event-based systems

- \+ Increasingly used for modern integration strategies
- \+ Easy addition/deletion of components
- – Potential scalability problems
- – Ordering of events

Event-based systems



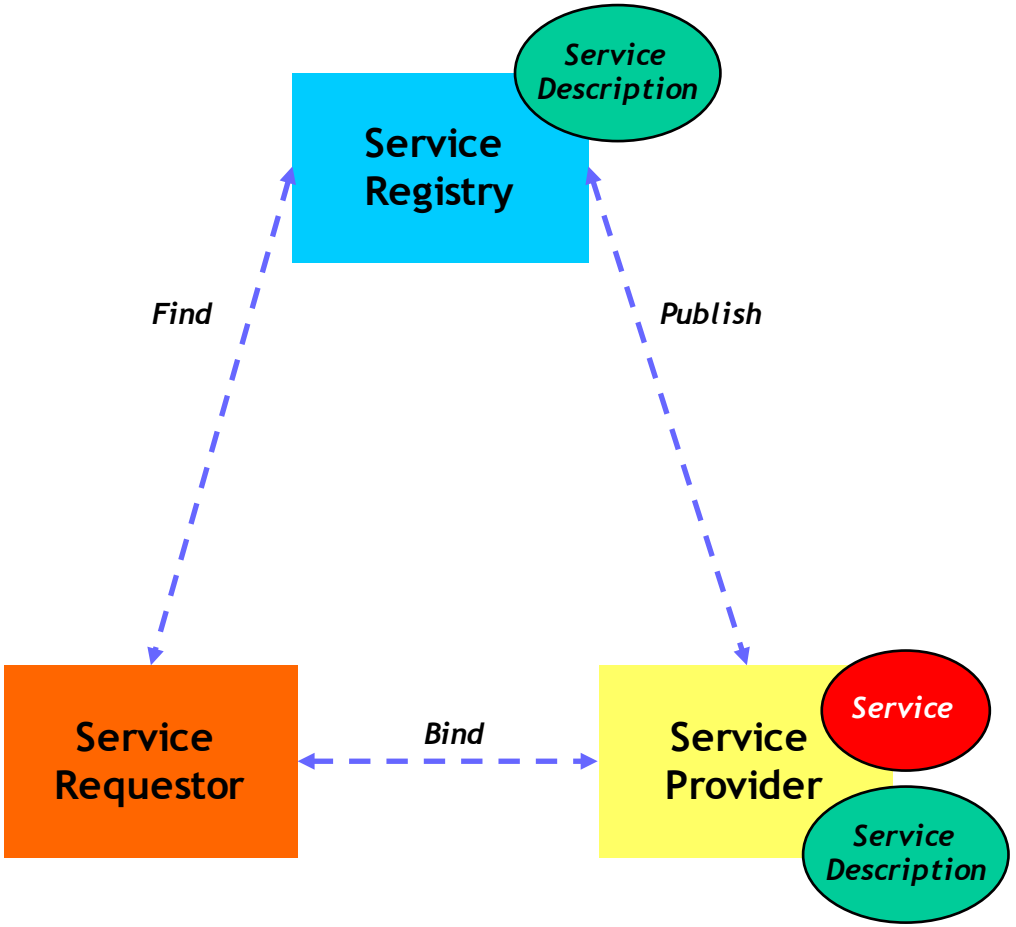
Common characteristics

- Asynchrony
  - send and forget
- Reactive
  - computation driven by receipt of message
- Location/identity abstraction
  - destination determined by receiver, not sender
- Loose coupling
  - senders/receivers added without reconfiguration
  - one-to-many, many-to-one, many-to-many

Service-oriented architecture (SOA)

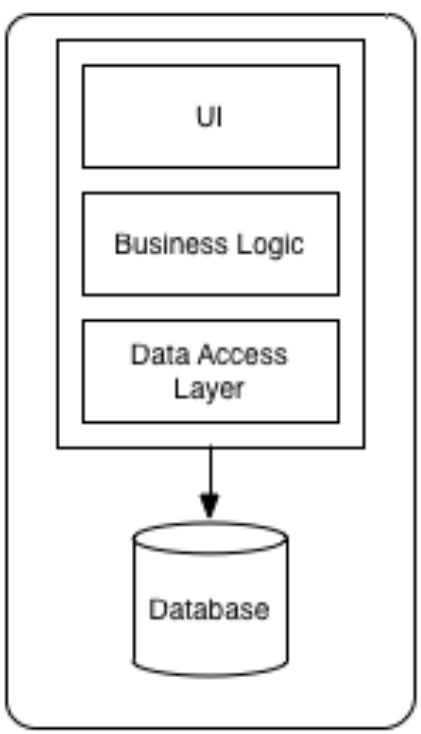
- The SOA defines three agents
  - Service requestors: they request the execution of a service
  - Service providers: they offer services
  - Intermediaries (service brokers, meta information providers, service registries): they provide information (metadata) about other information/services
- The interaction among these roles results in the execution of the following operations
  - Service description publication
  - Service discovery
  - Service binding

Roles and operations

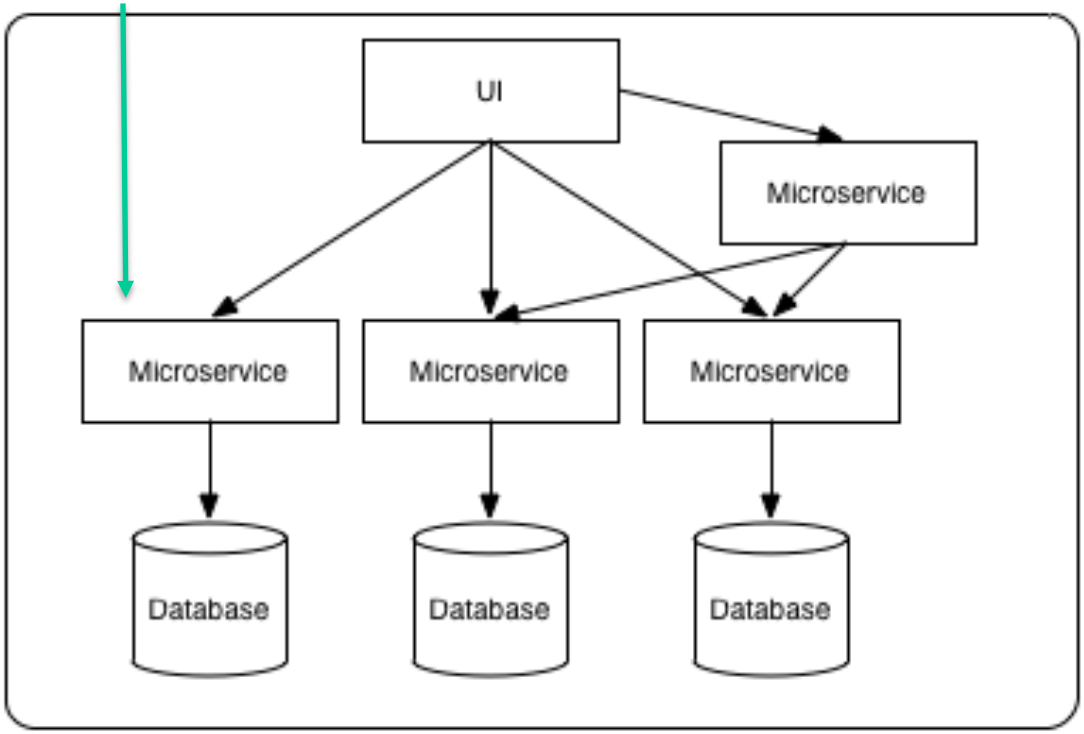


Microservices

Stateless
Available in multiple
No data sharing among microservices
Each microservice uses a specific database



Monolithic Architecture



Microservices Architecture

Politecnico
di Milano

… More Architectural Styles: Cloud Patterns

http://www.cloudcomputingpatterns.org

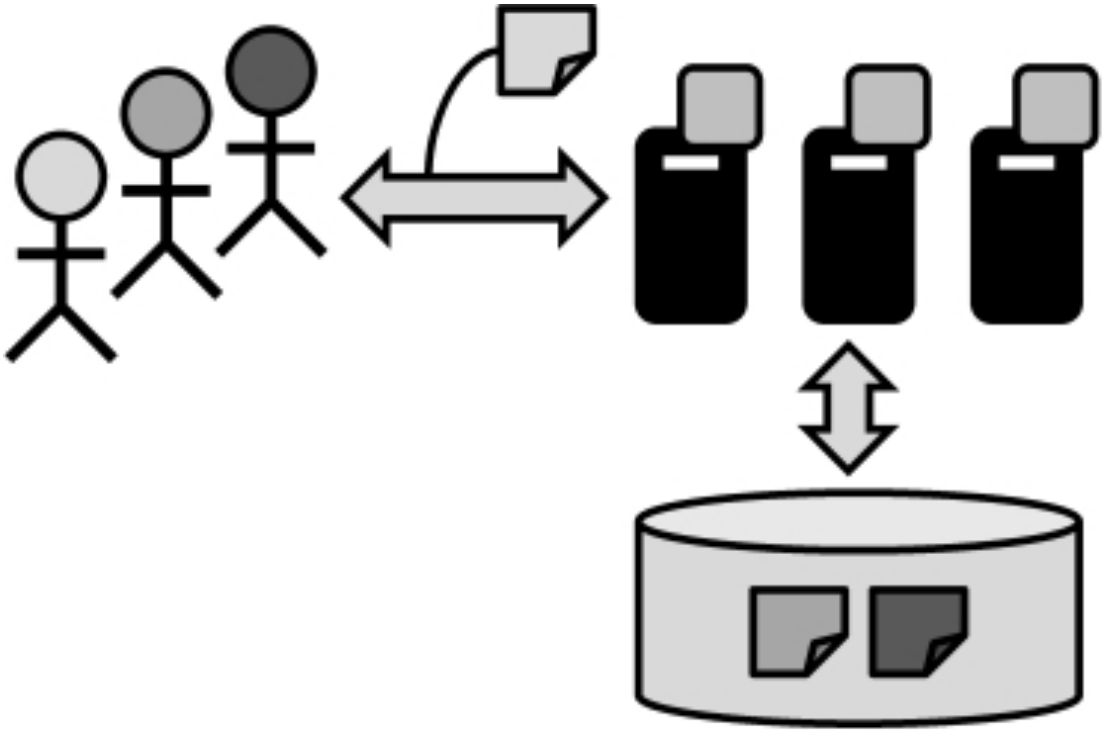
Classification of patterns

- Basic Architectural Patterns
  - Composite Application
  - Loose Coupling
  - Stateless Component
  - Idempotent Component
- Elasticity Patterns
  - Map Reduce
  - Elastic Component
  - Elastic Load Balancer
  - Elastic Queue
- Availability Patterns
  - Watchdog
  - Update Transition
- Multi-Tenancy Patterns
  - Single Instance Component
  - Single Configurable Instance Component
  - Multiple Instance Component

Stateless components

- Context
  - A componentized application subsumes components that can fail.
- Challenges
  - Cloud resources can display low availability.
  - Component instances can be added and removed regularly when the demand changes.
- Solution
  - Implement the components in a way that they do not contain any internal state, but completely rely on external persistent storage.

Stateless components

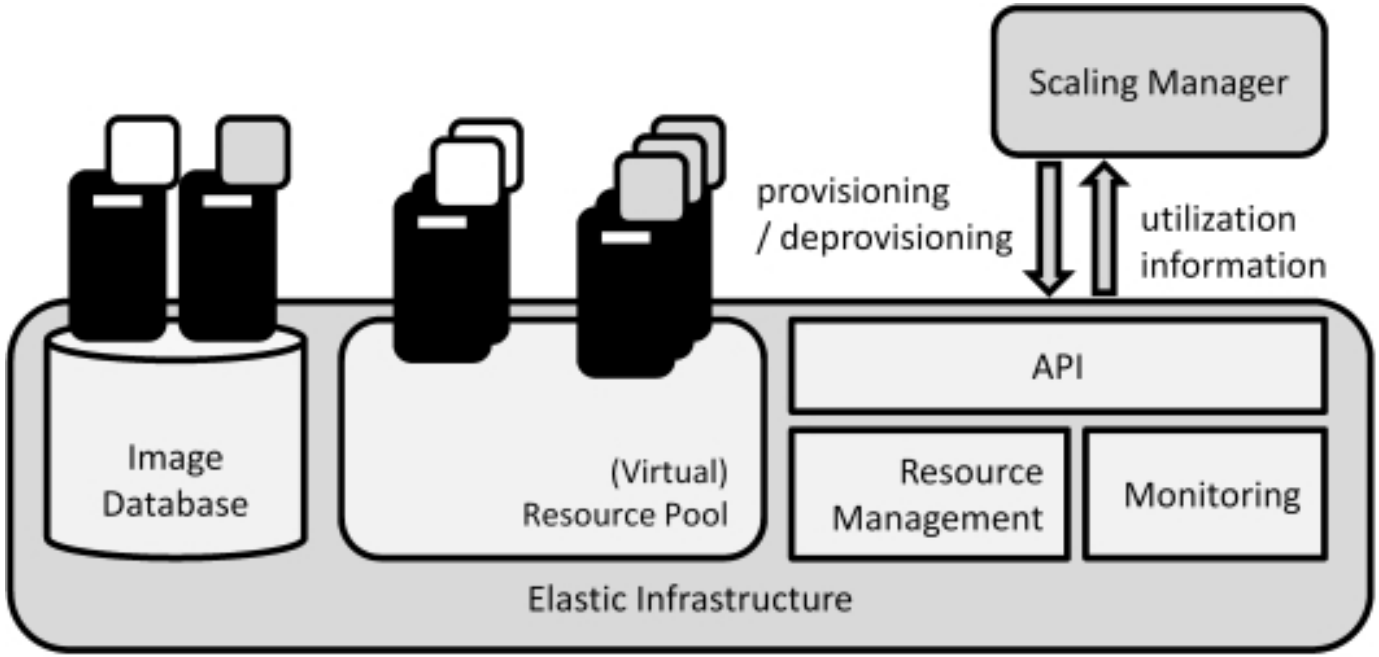


- No data is lost if an instance fails
- Multiple components have a common internal state
- The scalability of the central data store becomes the major challenge
  - Distributed data stores

Elastic component

- Context
  - A componentized application uses multiple compute nodes provided by an *elastic infrastructure*.
- Challenges
  - To benefit from the elastic infrastructure, the management process to scale-out a componentized application has to be automated.
  - Manual resource scaling would not support pay-per-use nor quickly changing workload
- Solution
  - Monitor the utilization of compute nodes that host application components and automatically adjust their numbers using the provisioning functionality provided by the elastic infrastructure.

Elastic component

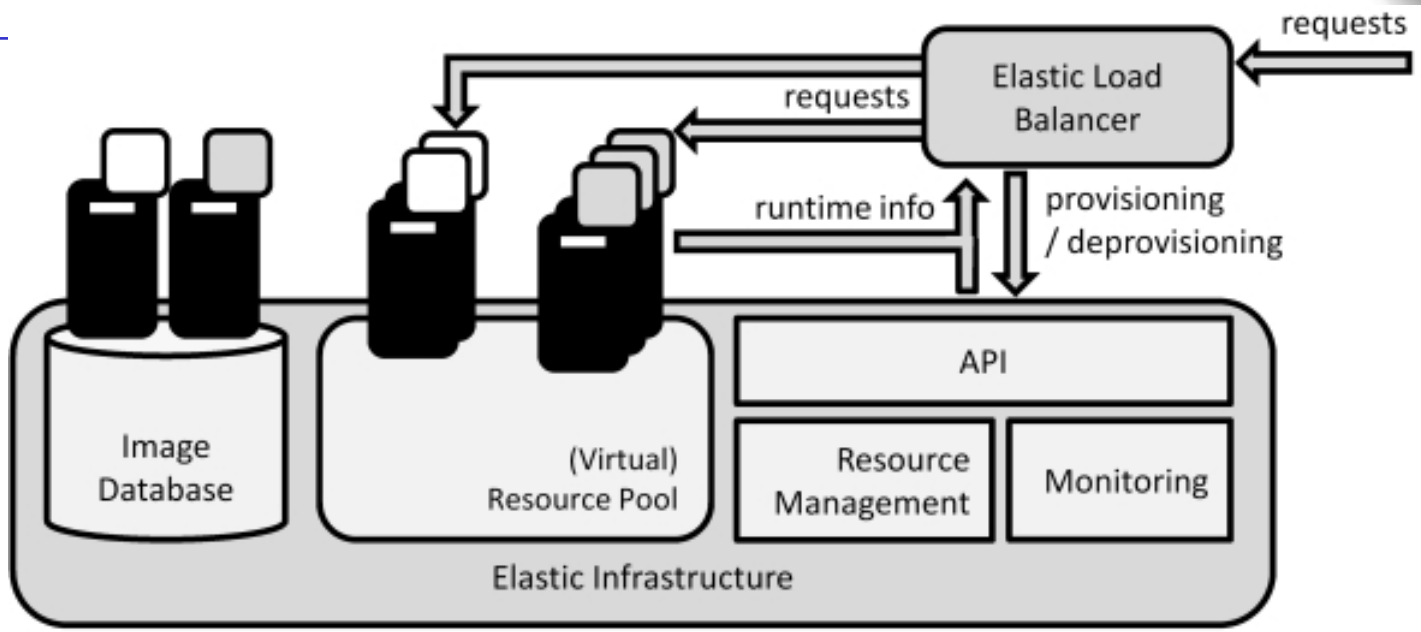


- If the utilization of compute nodes exceeds a specified threshold, additional hosting components are provisioned that contain the same application component.
- If the components are implemented as *stateless components*, the operations for adding and removal of components are significantly simplified.

Elastic load balancer

- Context
  - A componentized application uses multiple compute nodes provided by an elastic infrastructure.
- Challenges
  - As for elastic component
  - Service requests are a good indicator of workload and therefore shall be used as a basis for scaling decisions
- Solution
  - Use an elastic load balancer that determines the amount of required resources based on numbers of requests and provisions the needed resources accordingly using the elastic infrastructure's API

Elastic load balancer



- The number of requests that can be handled by a computing node is a crucial design parameter
- It has to be adjusted at run time
- Information about the time needed to provision new compute nodes can also be necessary to deduct effective scaling actions

Politecnico
di Milano

The design process

The Process of Architectural Design

- Definition:
  - *Defining the software architecture or design* is a problem-solving process whose objective is to find and describe a way:
    - To implement the system *functional requirements*...
    - While respecting the constraints imposed by the *quality, platform and process requirements...*
      - including the budget
    - And while adhering to general principles of *good quality*

Design as a series of decisions

- A designer is faced with a series of *design issues*
  - These are sub-problems of the overall design problem.
  - Each issue normally has several alternative solutions:
    - Design *options*.
  - The designer makes a *design decision* to resolve each issue.
    - This process involves choosing the best option from among the alternatives.

Making decisions

- To make each design decision, the software engineer uses:
  - Knowledge of
    - the requirements
    - the design as created so far
    - the technology available
    - software design principles and ‘best practices’
    - what has worked well in the past

Top-down and bottom-up design

- Top-down design
  - First design the very high level structure of the system.
  - Then gradually work down to detailed decisions about low-level constructs.
  - Finally arrive at detailed decisions such as:
    - the format of particular data items;
    - the individual algorithms that will be used.

Top-down and bottom-up design

- Bottom-up design
  - Make decisions about reusable low-level utilities.
  - Then decide how these will be put together to create high-level constructs.

- A mix of top-down and bottom-up approaches are normally used:
  - Top-down design is almost always needed to give the system a good structure.
  - Bottom-up design is normally useful so that reusable components can be created.

Top-down vs. bottom-up

A top-down design

- A very nice end result!
- It will take time to get it right…
- Individual components harder to re-use

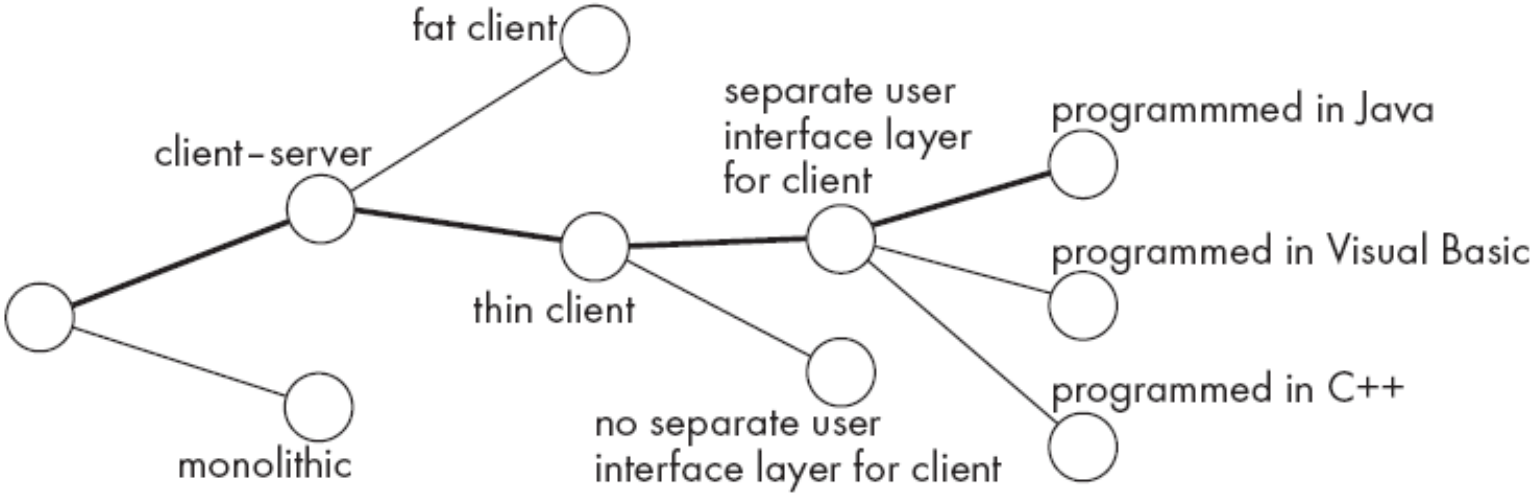
A bottom-up design

- A bit rough end result
- It will be quicker to get to…
- Individual components easier to re-use

You are to find the right trade-off!

Design space

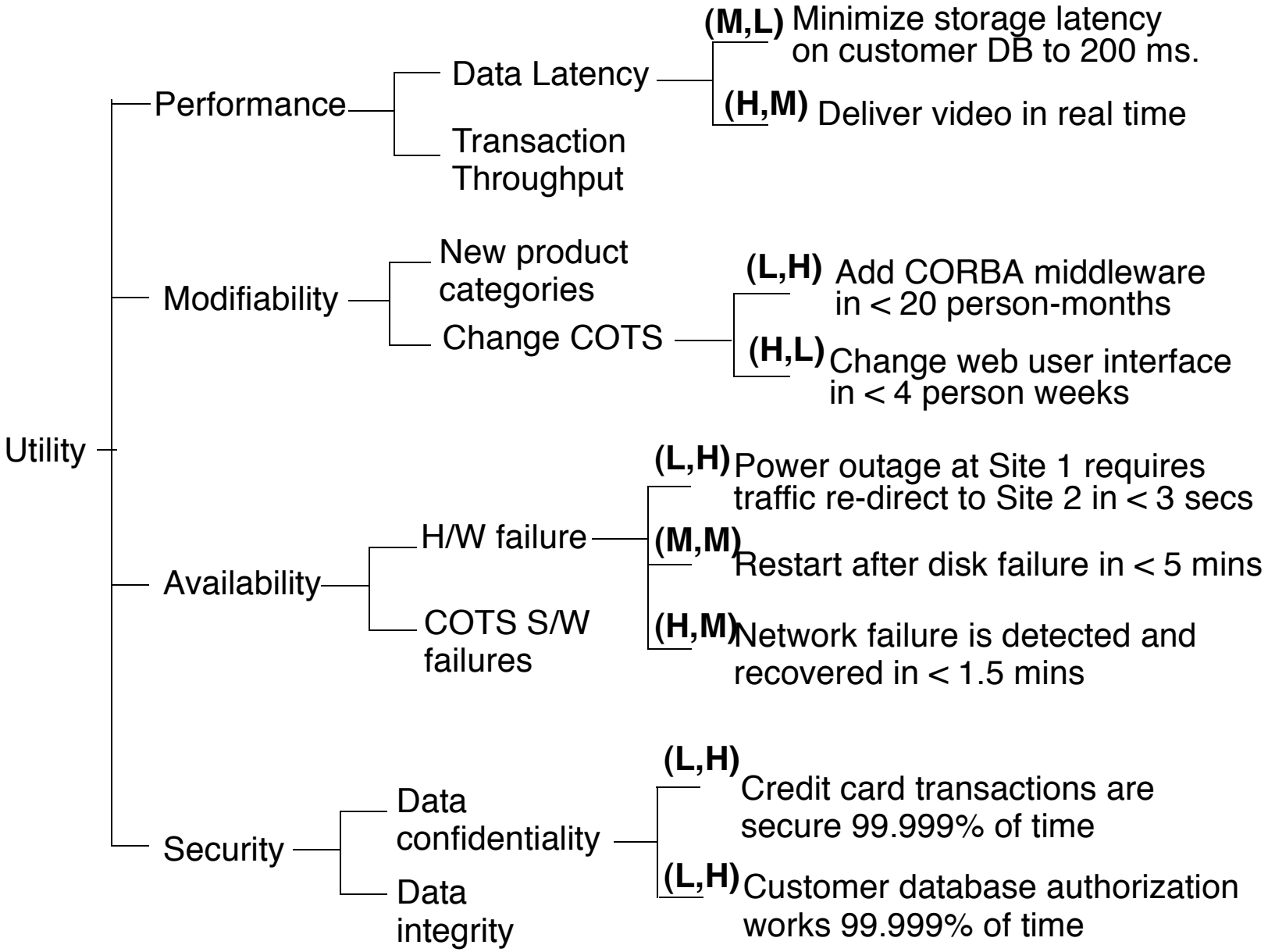
- The space of possible designs that could be achieved by choosing different sets of alternatives is often called the design space
  - For example:



ATAM: the Architecture Tradeoff Analysis Method

- It supports making the consequences of design decisions explicit
- It makes possible for stakeholders to trade off the different possibilities, and make informed decisions, with clear insight into the consequences thereof
- It supports users to determine how quality attributes interact.
  - Example1: If we decide to include an authorization component to increase security, this is likely degrading performance.
  - Example2: If we add redundancy to increase availability we increase cost and (possibly) degrade performance.

Utility Tree



Politecnico
di Milano

Analyzing architectures: quantitative and qualitative approach: two examples

Availability analysis slides credits by Danilo Ardagna

Analysis of architectures

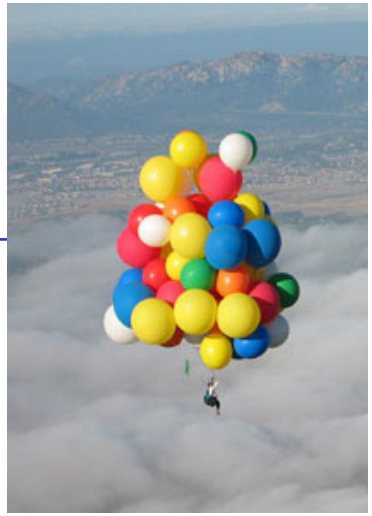
- A number of QoS dimensions of the resulting system are directly influenced by the architectural style of choice
  - Scalability
  - Reliability
  - Availability
  - Usability
  - …
- Specific methodologies to analyze these aspects exist

Analysis techniques

There are two broad classes of techniques to evaluate a software architecture:

- The first class comprises measuring techniques, and rely on *quantitative information*. Examples include architecture metrics and simulation. **(Availability and Reliability)**
- The second class comprises questioning techniques, in which one investigates how the architecture reacts to certain situations. This is often done with the help of scenarios **(ATAM method, see before)**

Availability

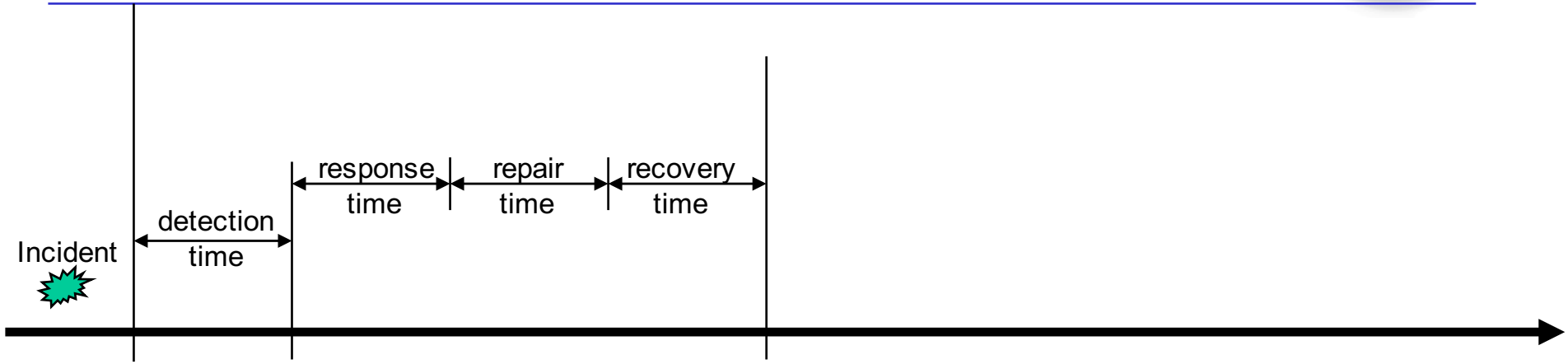


- A service shall be **continuously available** to the user
  - Little downtime and rapid service recovery
- The availability of a service depends on:
  - Complexity of the IT infrastructure architecture
  - Reliability of the individual components
  - Ability to respond quickly and effectively to faults
  - Quality of the maintenance by support organizations and suppliers
  - Quality and scope of the operational management processes

Reliability

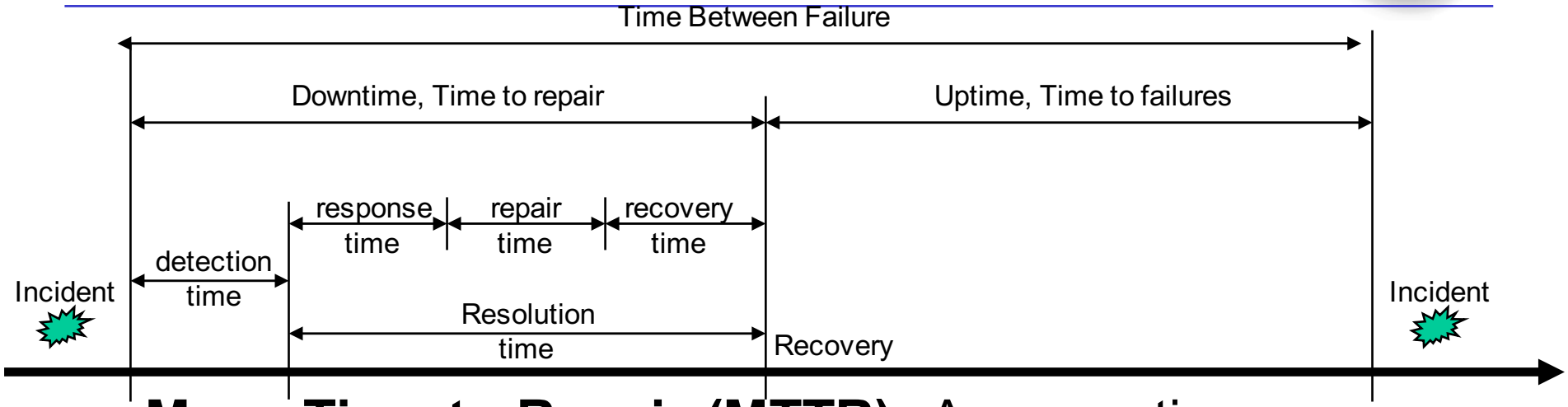
- **Adequate reliability** means that the service is available for an agreed period without interruptions
- The reliability of a service increases if downtime can be prevented
- Reliability is determined by:
  - Reliability of the components used to provide the service
  - Ability of a service or component to operate effectively despite failure of one or more subsystems
  - Preventive maintenance to prevent downtime

System life-cycle



- **Time of occurrence:** Time at which the user becomes aware of the fault
- **Detection time:** The service provider is informed of the fault
- **Response time:** Time required by the service provider (diagnosis) to respond to the user
- **Repair time:** Time required to restore the service or the components that caused the fault
- **Recovery time:** Time required to restore the system (re-configuration, re-initialization,…)

System life-cycle

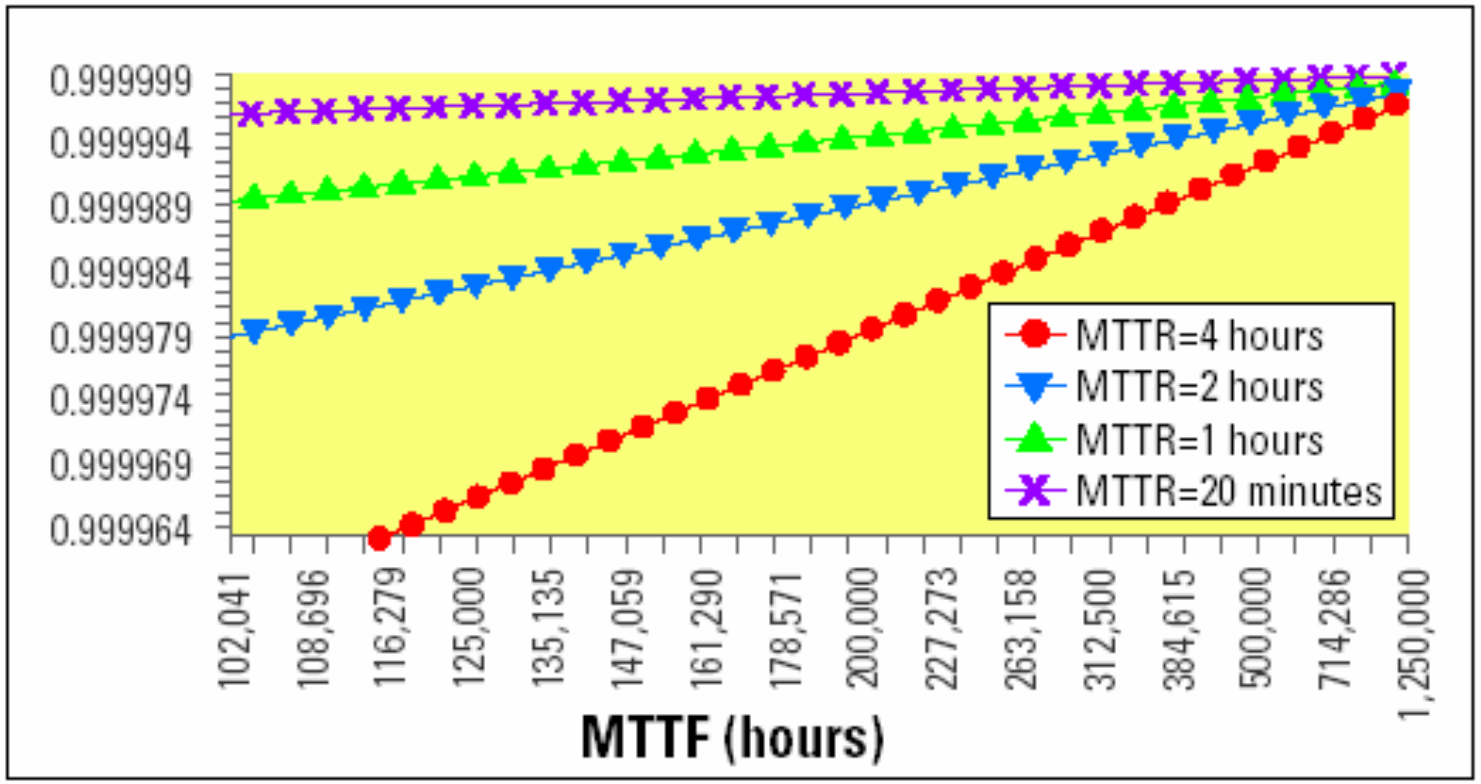


- **Mean Time to Repair (MTTR):** Average time between the occurrence of a fault and service recovery, also known as the downtime
- **Mean Time To Failures (MTTF):** Mean time between the recovery from one incident and the occurrence of the next incident, also known as uptime
- **Mean Time Between failure (MTBF):** Mean time between the occurrences of two consecutive incidents

Availability vs. Reliablity

- **Availability**: The probability that a component is working properly at time *t*
  - $A = MTTF / (MTTF+MTTR)$
- **Reliability**: The probability that a component has always been working properly during a time interval (0,*t*)
  - $R = e^{-\lambda t}$ $\lambda = 1/MTTF$
- Reliability requires that the component never fails in the interval (0,*t*)
- From the availability perspective a given component could have failed in the interval (0,*t*), but it could have been repaired before t

Availability, MTTF, MTTR



MTBF = MTTF+MTTR (if MTTR small, MTBF ≅ MTTF)

Nines notation

- Availability is typically specified in nines notation
- For example 3-nines availability corresponds to 99.9%, 5-nines availability corresponds to 99.999% availability

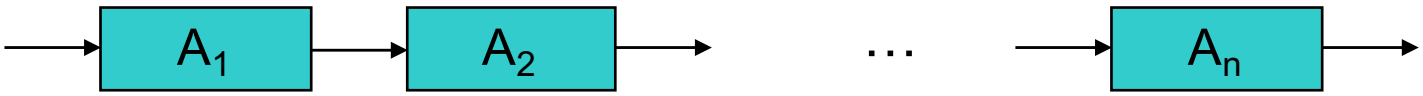
Availability	Downtime
90% (1-nine)	36.5 days/year
99% (2-nines)	3.65 days/year
99.9% (3-nines)	8.76 hours/year
99.99% (4-nines)	52 minutes/year
99.999% (5-nines)	5 minutes/year

Availability

- Calculated by modeling the system as an interconnection of parts in series and parallel
- If failure of a part leads to the combination becoming inoperable, the two parts are considered to be operating in **series**
- If failure of a part leads to the other part taking over the operations of the failed part, the two parts are considered to be operating in **parallel**

Availability in series

- The combined system is operational only if both every part is available
- The combined availability is the **product** of the availability of the component parts



$$A = \prod_{i=1}^{n} A_i$$

Availability in series – A numerical example

	Availability	Downtime
Component 1	99% (2-nines)	3.65 days/year
Component 2	99.999% (5-nines)	5 minutes/year
Combined	98.999%	3.65 days/year

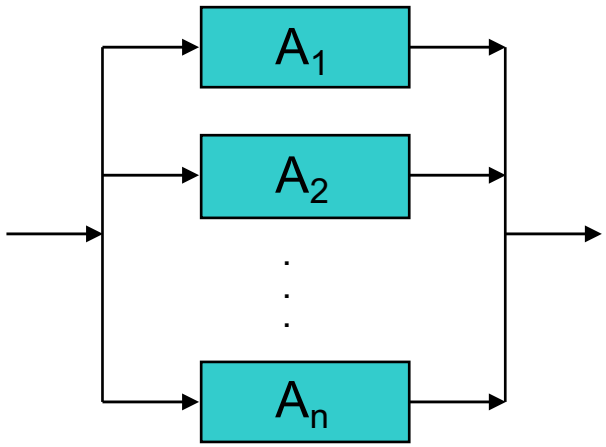
Downtime=(1-A)*365 days/year

The availability of the entire system is negatively affected by the low availability of Component 1

A chain is as strong as the weakest link!

Availability in parallel

- The combined system is operational if at least one part is available
- The combined availability is **1 - (all parts are unavailable)**



$$A = 1 - \prod_{i=1}^{n} (1 - A_i)$$

Availability in parallel – A numerical example

	Availability	Downtime
Component 1	99% (2-nines)	3.65 days/year
Component 2	99% (2-nines)	3.65 days/year
Combined	99.99% (4-nines)	52 minutes/year

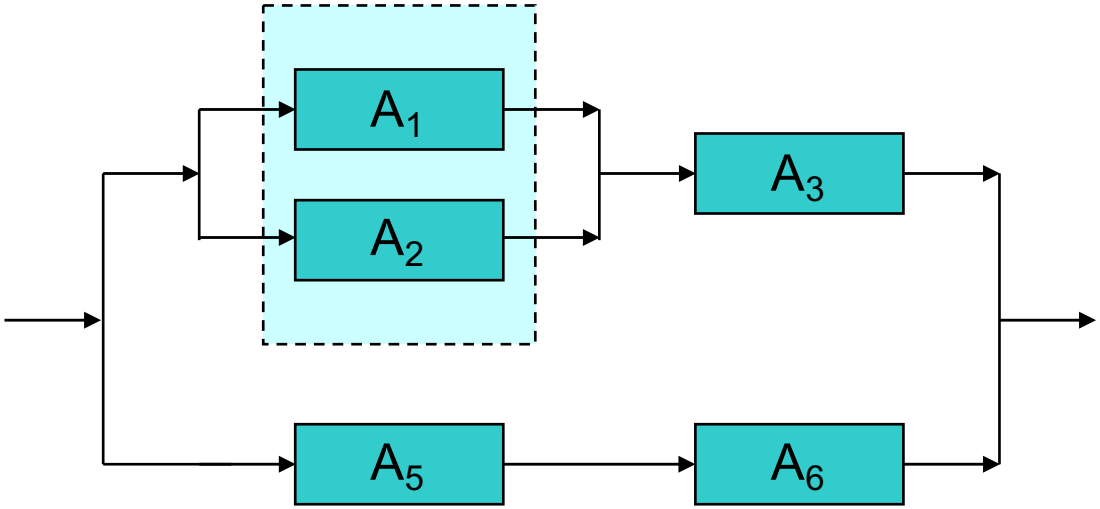
Downtime=(1-A)*365 days/year

Even though very low availability components are used, the overall availability of the system is much higher

Mission critical systems are designed with redundant components!

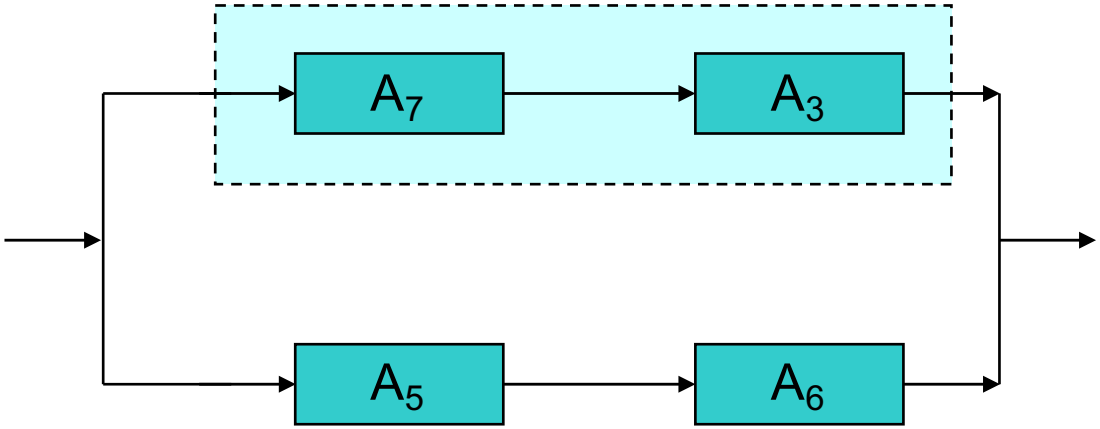
Availability of complex systems

$A_7 = 1-(1-A_1)(1-A_2)$

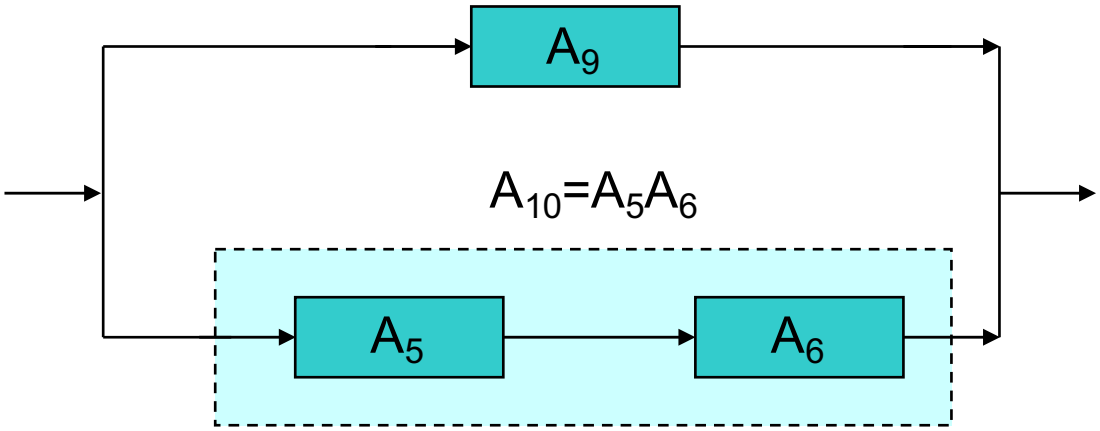


Availability of complex systems

$A_8 = A_7 A_3$

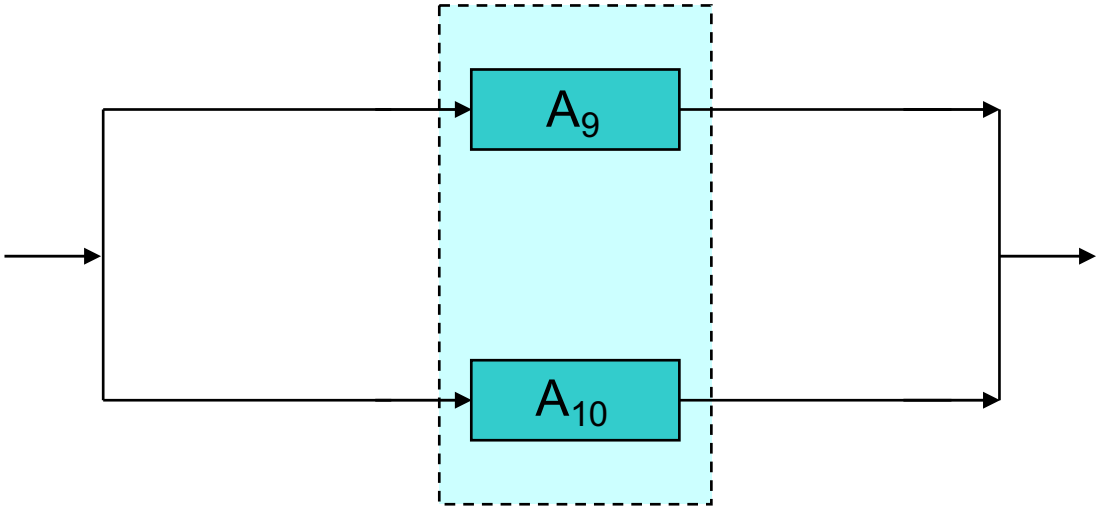


Availability of complex systems



Availability of complex systems

$$A=1-(1-A_9)(1-A_{10})$$



… so far…

- Defining the architecture of a software system means
  - Identifying the styles that are most suitable
  - Instantiate them in a specific structure made of meaningful components and connectors
  - Identify the views that are most relevant for representation
  - Assess the qualities of the architecture
  - Record all design decisions and the process that you followed to get to them