Politecnico
di Milano

Software Design &
Software Architecture

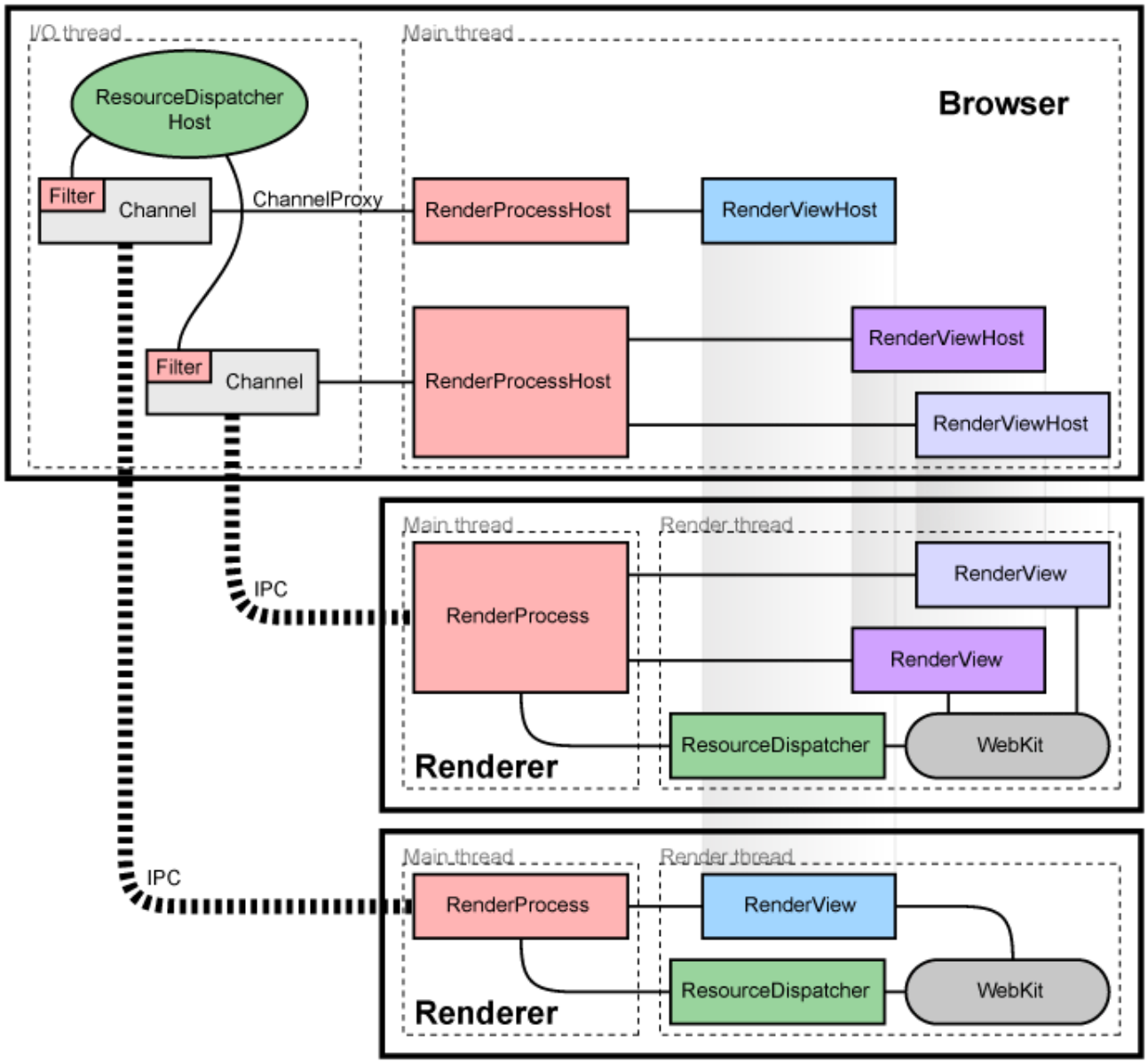
Software architecture, definition (1)

"The architecture of a software system defines that system in terms of computational components and interactions among those components."

[Shaw and Garlan, '96], Shaw and Garlan, *Software Architecture, Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

Example: Chromium Architectural overview

https://www.chromium.org/developers/design-documents/multi-process-architecture



Software Architecture, definition (2)

"The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

[Clements and Kazman, 2003] Clements and Kazman, *Software Architecture in Practice*, SEI Series in Software Engineering. Addison-Wesley, 2003.

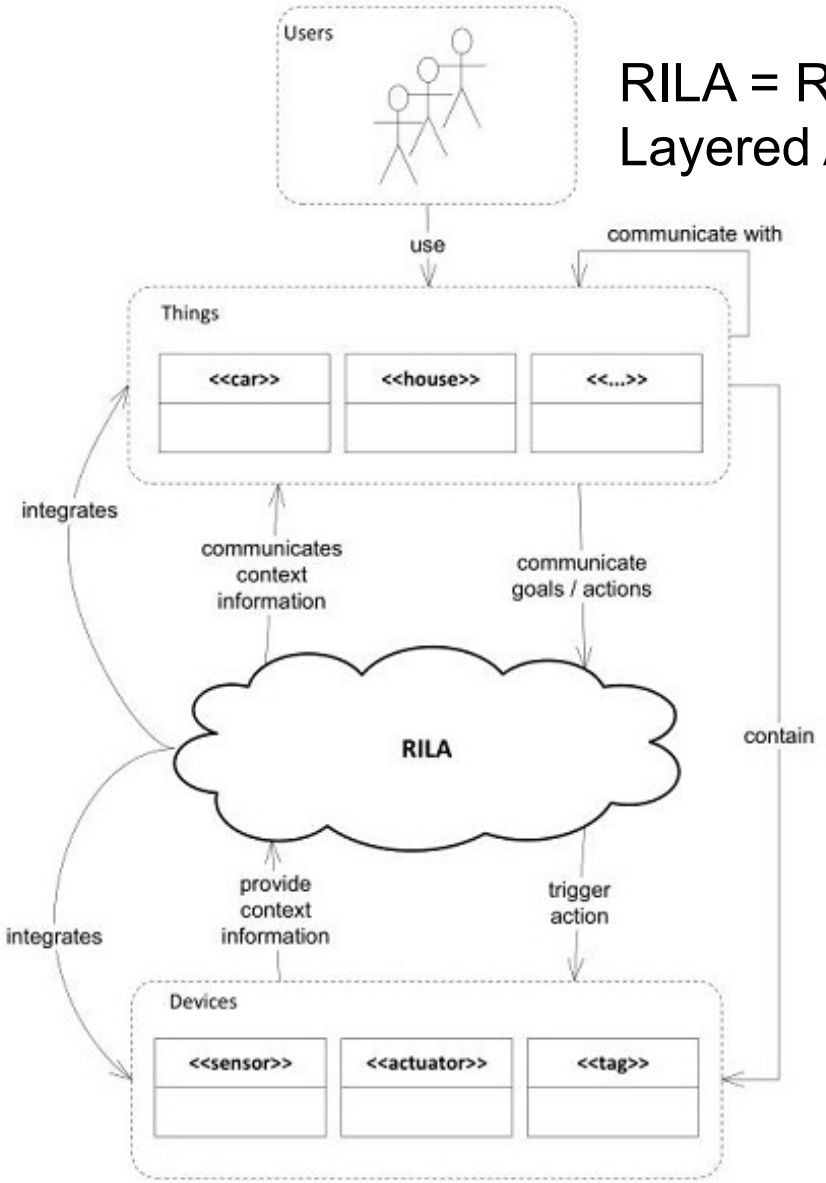
Software Architecture

- Important issues raised in this definition:
  - multiple system structures;
  - externally visible (observable) properties of components.

- The definition does not include:
  - the process;
  - rules and guidelines;
  - architectural styles.

Example: an Internet of Things reference architecture (1)

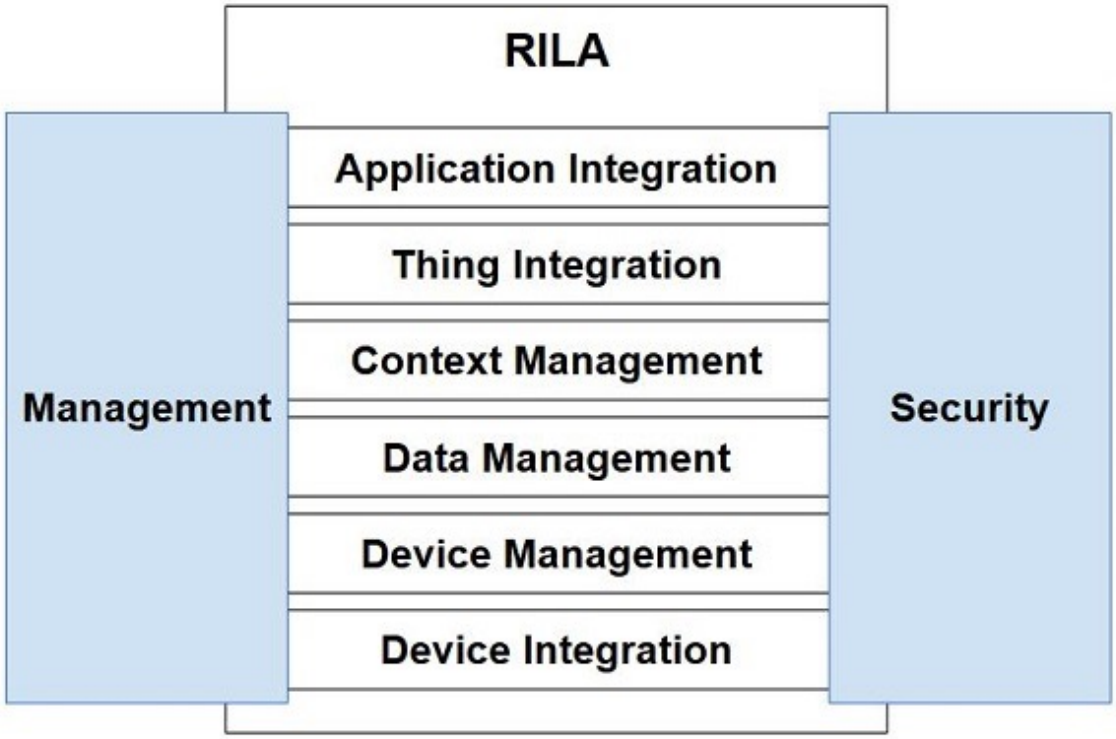
https://www.infoq.com/articles/internet-of-things-reference-architecture



RILA = Reference IoT Layered Architecture

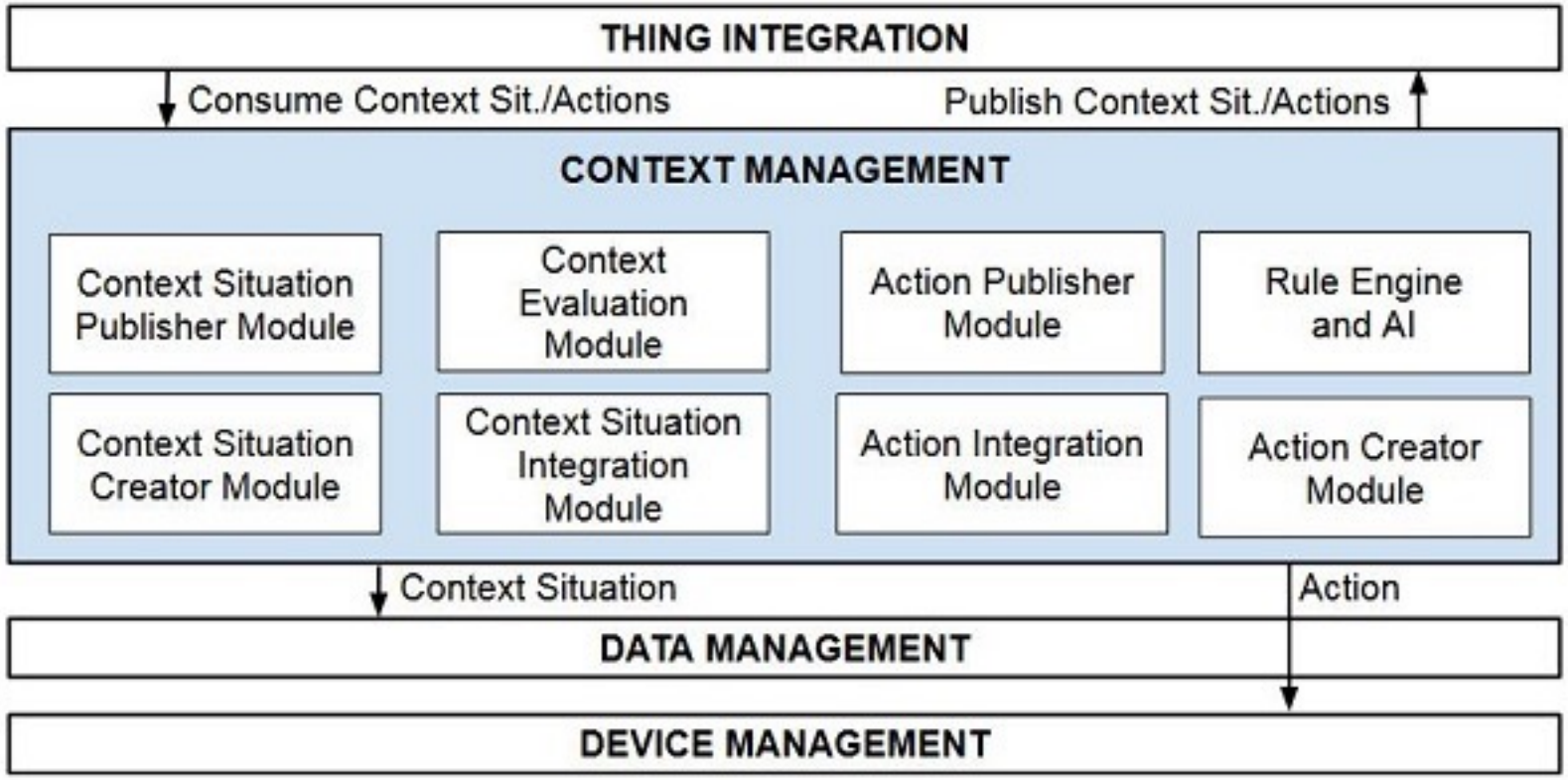
Example: an Internet of Things reference architecture (2)

https://www.infoq.com/articles/internet-of-things-reference-architecture



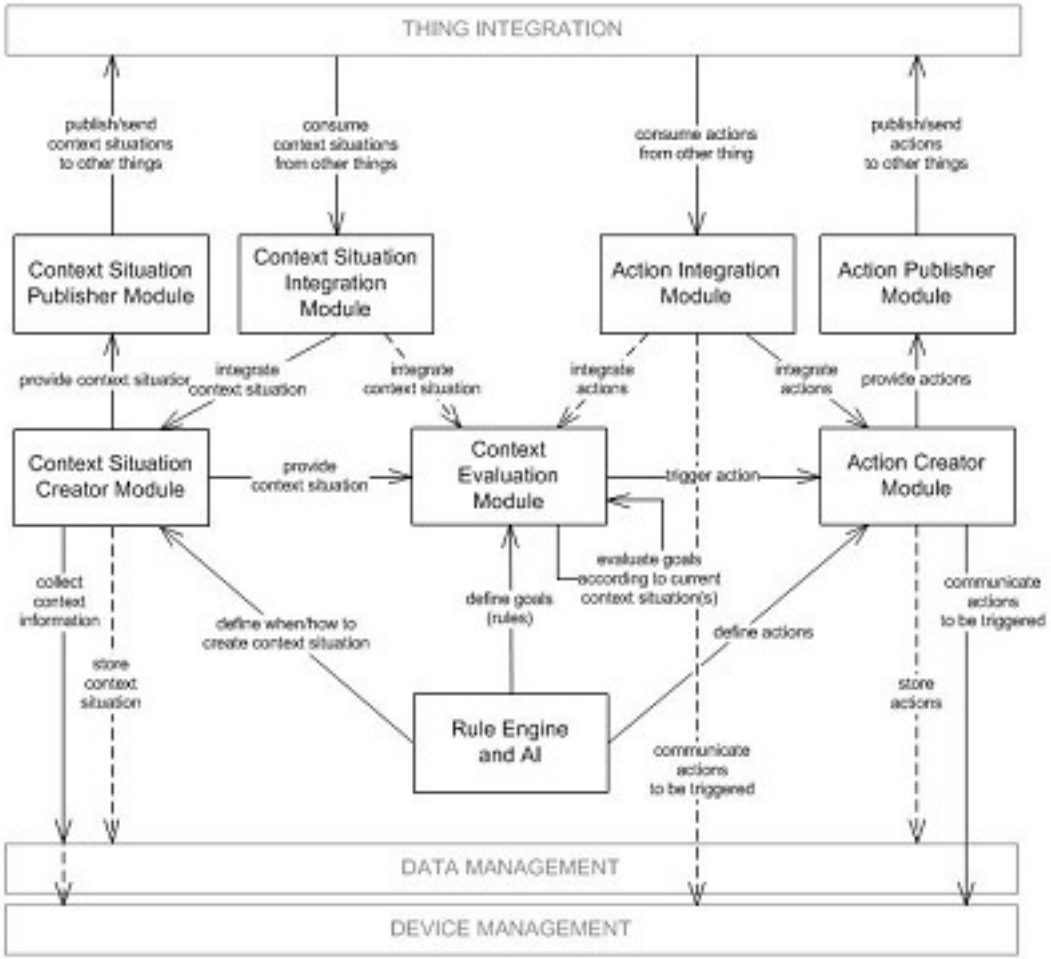
Example: an Internet of Things reference architecture (3)

https://www.infoq.com/articles/internet-of-things-reference-architecture



Example: an Internet of Things reference architecture (4)

https://www.infoq.com/articles/internet-of-things-reference-architecture



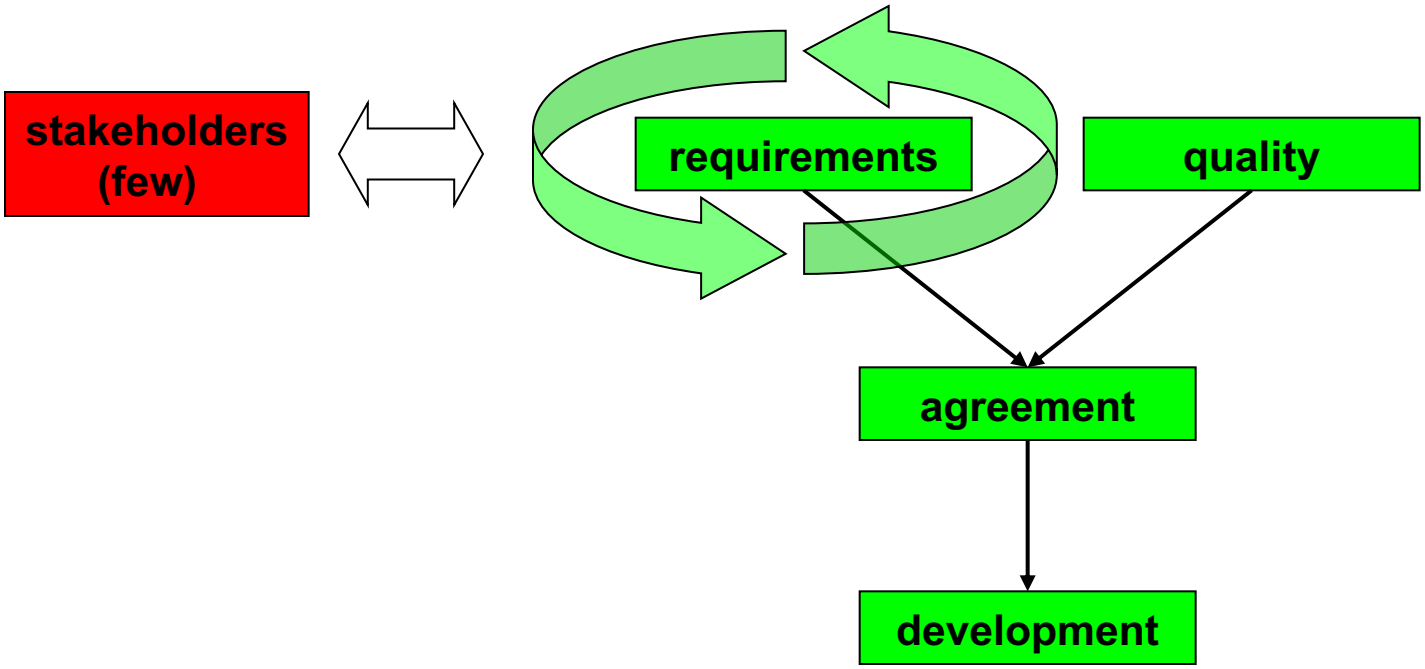
What can we understand from this layered architecture?

- Layers enable separation of concerns
- We can identify the main focuses of each layer at a glance.
- Each layer can be implemented using a different team
- Each layer remains independent from the others, provided that it respects the protocols defined for communication with the neighbour layers

Why Is Architecture Important?

- Architecture is the vehicle for communication: internal (different teams), external (teams and stakeholders)
- Architecture manifests the earliest set of design decisions
  - Introduces constraints on implementation
  - Introduces organizational structure
  - Inhibits or enables quality attributes
- Architecture is a transferable abstraction of a system
  - Product lines share a common architecture
  - Allows for template-based development
  - Basis for training

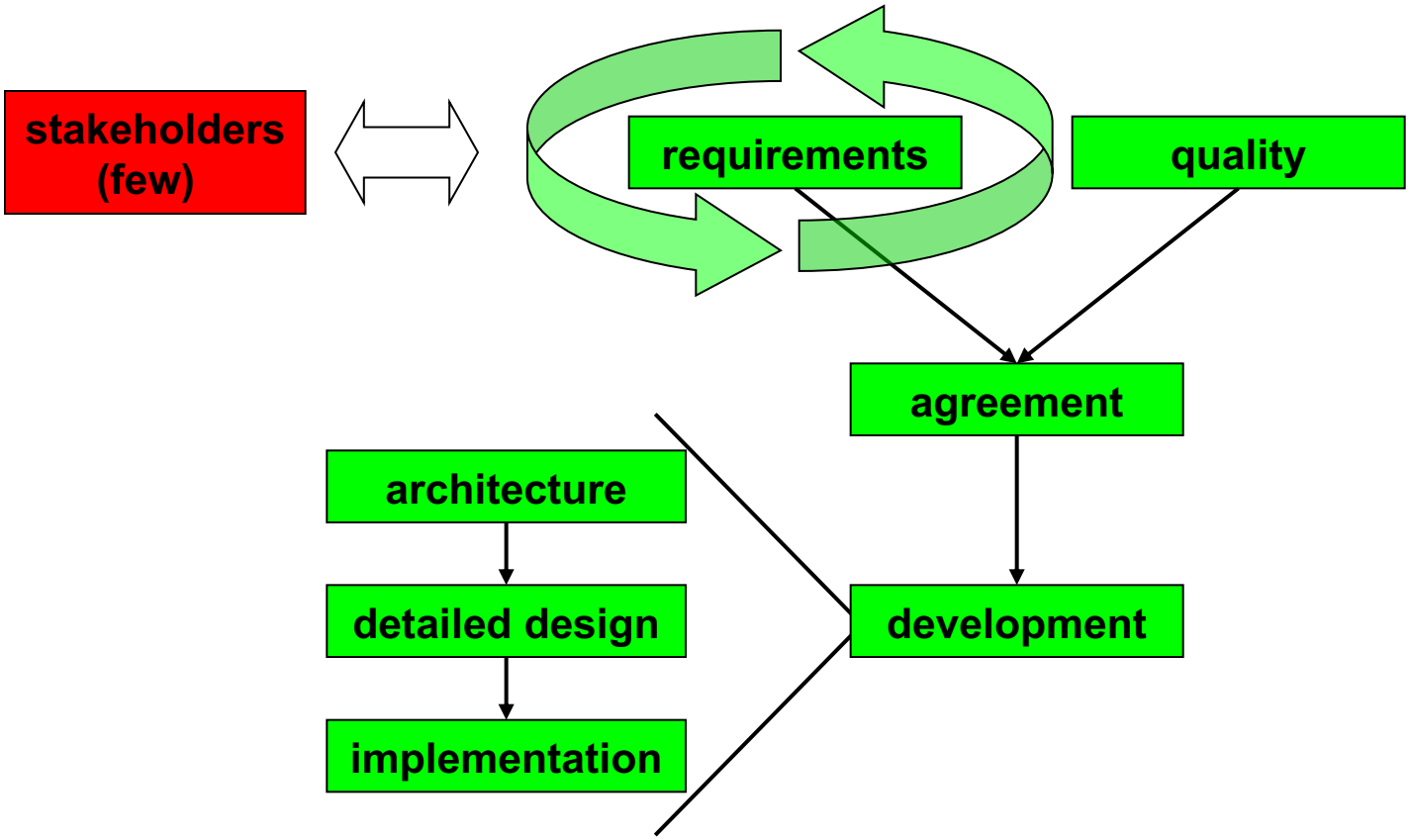
Pre-architecture life cycle



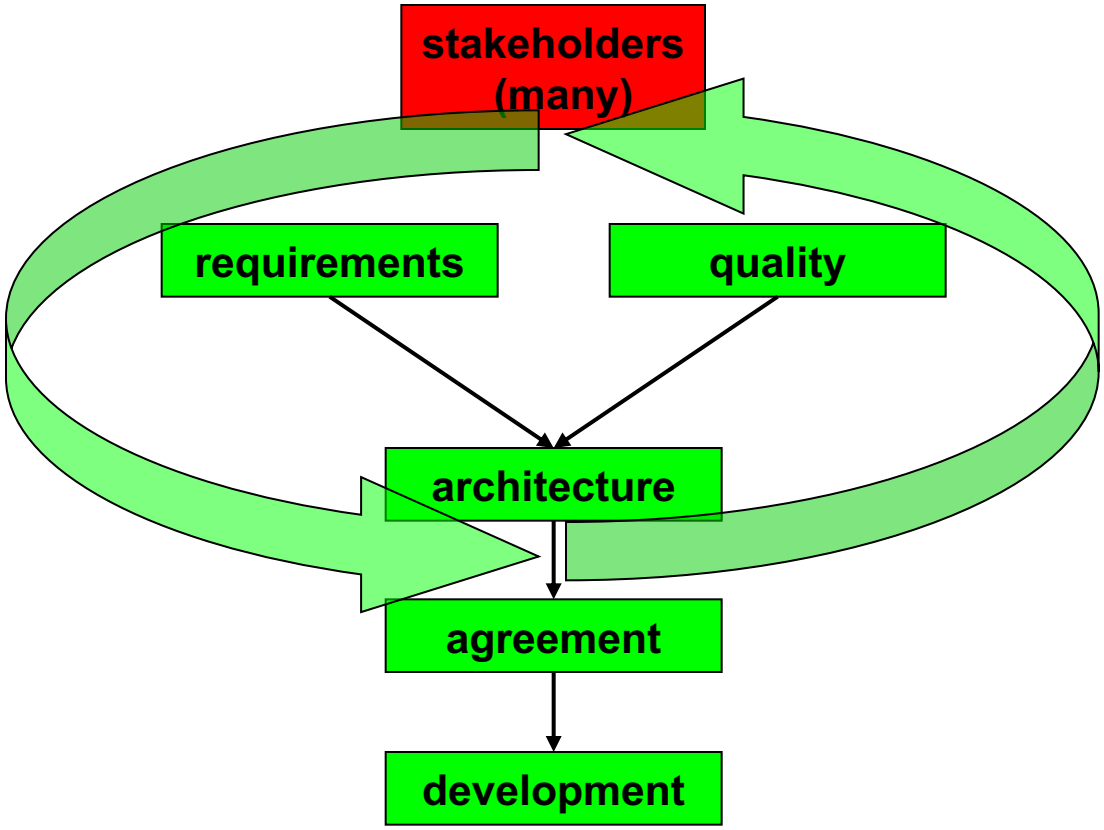
Characteristics

- Iteration mainly on functional requirements
- Few stakeholders involved
- No balancing of functional and quality requirements

Adding architecture, the easy way



Giving a more crucial role to the architecture



Characteristics

- Iteration on both functional and quality requirements
- Many stakeholders involved
- Balancing of functional, quality requirements, and architectural aspects

The architecture is intrinsic to our software!

A simple example: a trivial prototype of a “banking application” - Account


```
import java.rmi.*;
public interface Account extends Remote {
    public float balance() throws RemoteException;
}

import [...]
public class AccountImpl extends UnicastRemoteObject
                         implements Account {
    // implementation of class (method balance, etc.)
}
```

A simple example: a trivial prototype of a “banking application” - AccountFactory


```
import java.rmi.*;
public interface AccountFactory extends Remote {
    public Account createAccount(String userName, int balance)
                    throws RemoteException;
    public int getLoad() throws RemoteException;
}

import [...]
public class AccountFactoryImpl extends UnicastRemoteObject
                                implements AccountFactory {
    // implementation of class
    // (methds createAccount, getLoad, etc.)
    // …
}
```

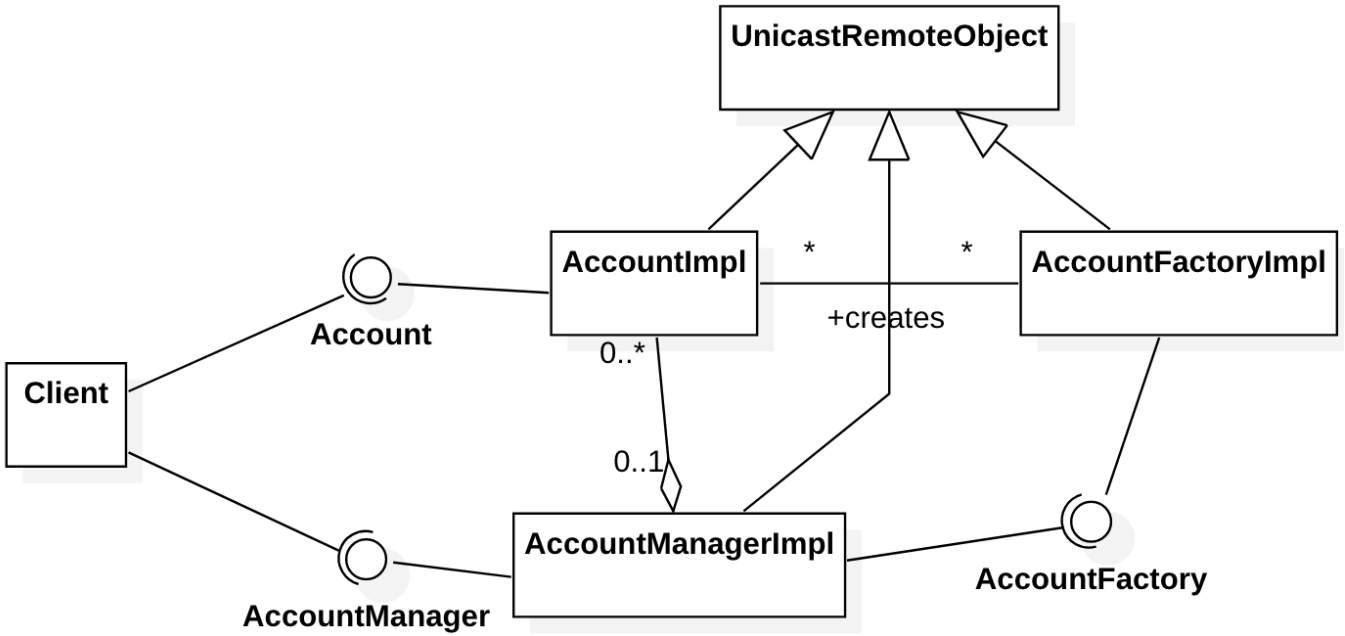
A simple example: a trivial prototype of a “banking application” - AccountManager


```
import java.rmi.*;
public interface AccountManager extends Remote {
        public Account open(String name) throws RemoteException;
}

import [...]
public class AccountManagerImpl extends UnicastRemoteObject
                                implements AccountManager {

    private Map<…> accounts = new HashMap< >();
    // implementation of class (attributes, method open, etc.)
    public void addFactory(String hostName) { … }
    public AccountFactory selectFactory() { … }
}
```

A simple example: a trivial prototype of a “banking application” - Client


```
import java.rmi.*;
public class Client {
    public static void main (String[] args) {
        try {
            // … some stuff
            AccountManager manager =
                (AccountManager) Naming.lookup( "//" + args[0] +
                                                "/AccountManager );
            String name = "Jack B. Quick";
            Account account = manager.open(name);
            float balance = account.balance();
            System.out.println ("The balance in " + name +
                                "'s account is $" + balance);
        } catch (Exception e) {  e.printStackTrace(); }
} }
```

A simple example: a trivial prototype of a “banking application” - Low level design view



- What info do I get from this?
  - Hierarchy
  - The client uses remote objects (sends messages to objects that extend UnicastRemoteObject)
  - Client uses two interfaces
  - The other interface is used by AccountManager

What happens if changes are applied on the source code?

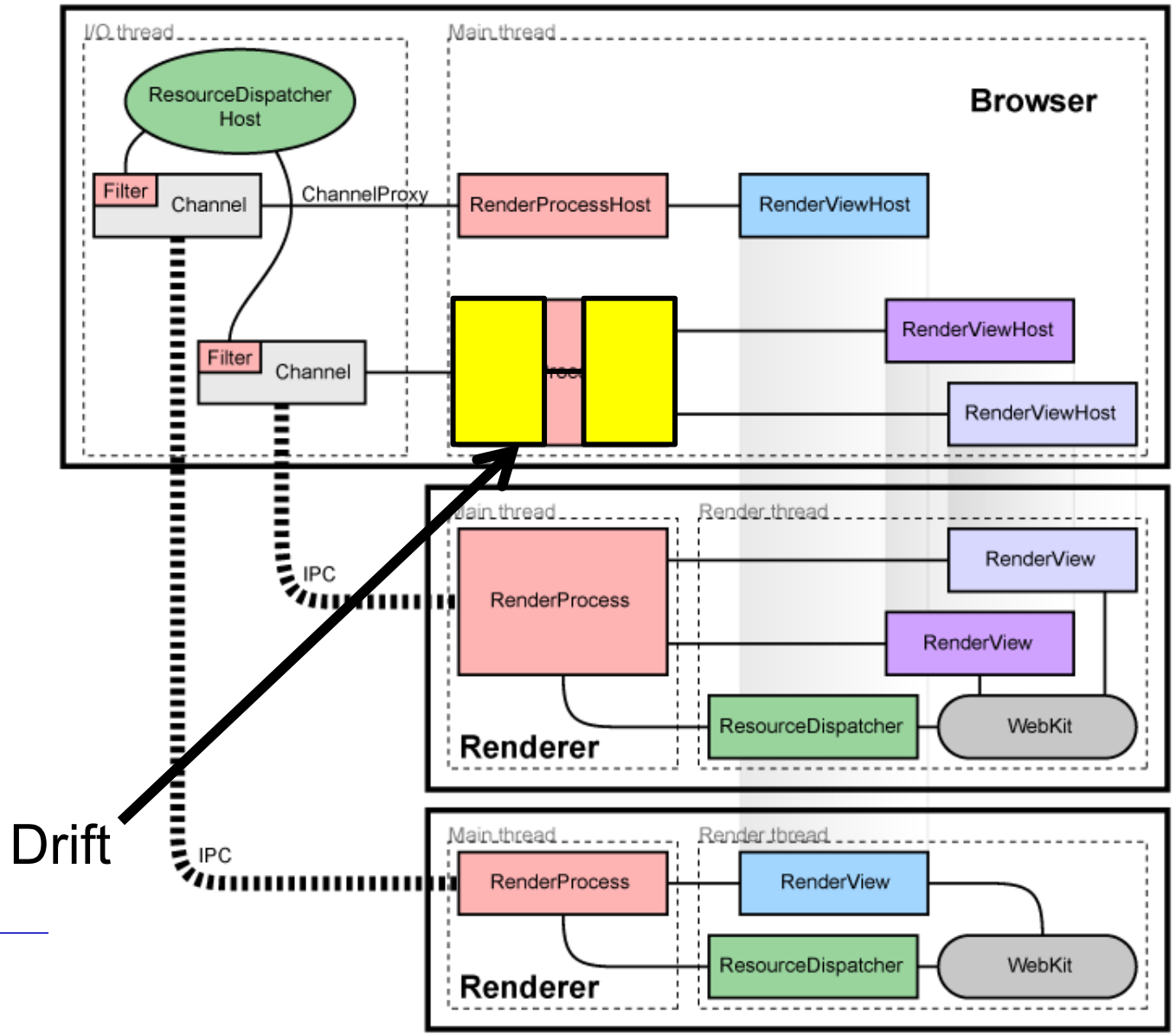
- Architectural degradation
  - Changes in the software system are not reflected in the corresponding architectural description
  - Misalignment between documentation and implementation
- Why does this happen?
  - Lack of a documented architecture
  - Inadequate processes and supporting tools
  - Developer sloppiness (or technical debt)
  - Perception of short deadlines

Drift and erosion

- Two kinds of degradation
  - **Drift:** the change is not included nor implied by the architecture but it does not necessarily imply violations
  - **Erosion**: the change violates prescriptive design decisions

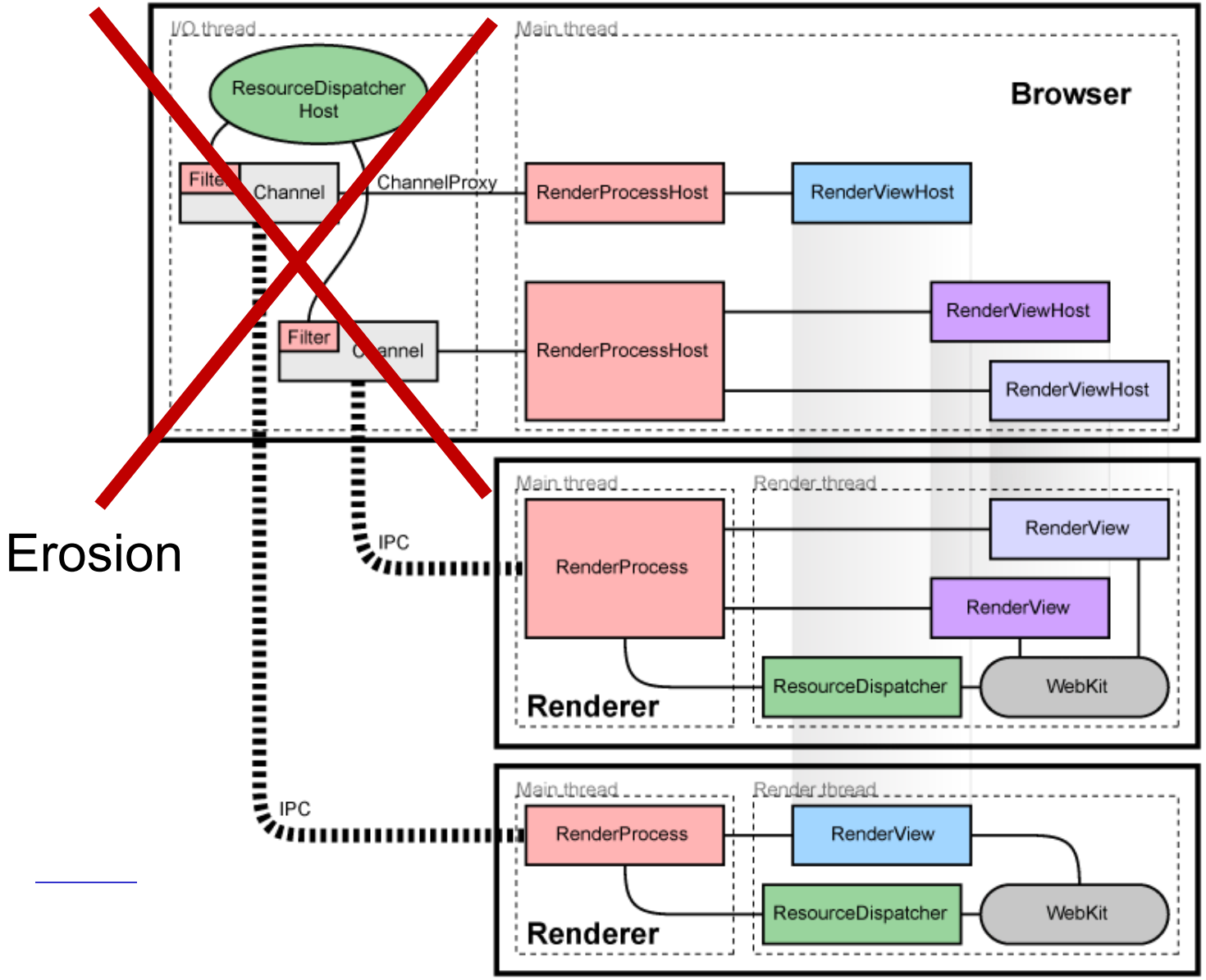
Example – Chromium Architecture

https://www.chromium.org/developers/design-documents/multi-process-architecture



Example – Chromium Architecture

https://www.chromium.org/developers/design-documents/multi-process-architecture



Software Architecture & Quality

- The notion of quality is central: the SA is derived according to qualities of the system at the earliest possible stage.
- External qualities: performance, security, availability, functionality, usability
- Internal qualities: modifiability, portability, reusability, integrability, testability

Politecnico
di Milano

Architectural styles

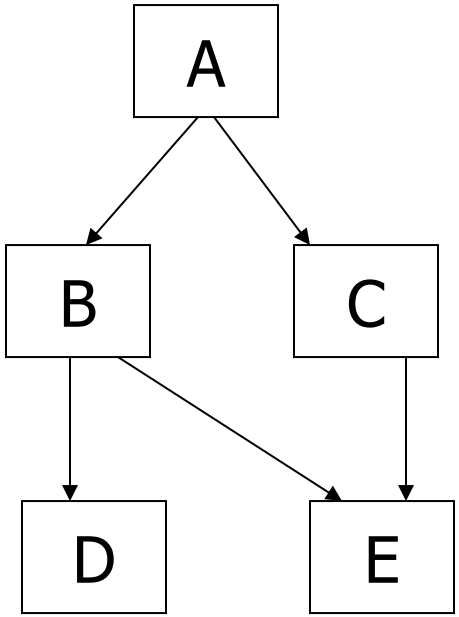
Architectural Style

- "*an architectural style determines the **vocabulary** of components and connectors that can be used in instances of that style, together with a **set of constraints** on how they can be combined. These can include **topological** constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with **execution** semantics—might also be part of the style definition."* [Garland & Shaw]

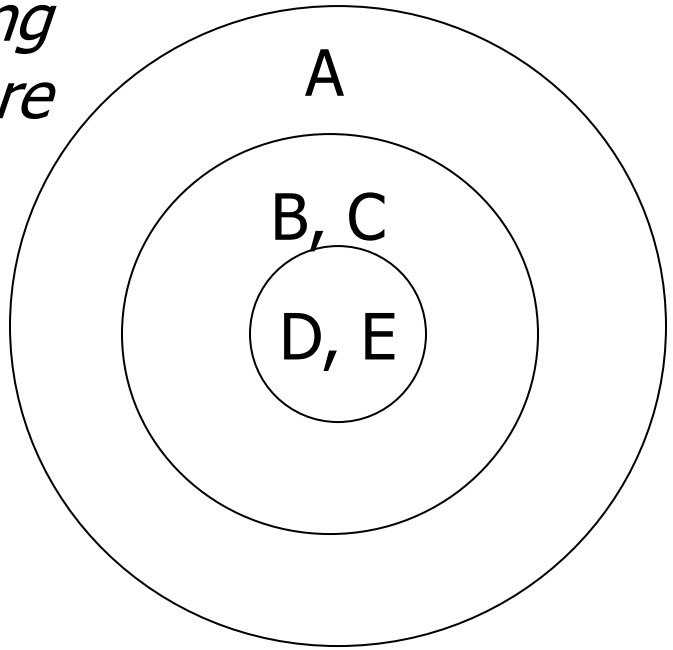
- [Garland & Shaw] David Garlan and Mary Shaw, "*An Introduction to Software Architecture",* Jan 1994, CMU-CS-94-166
  - http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf

Layered style

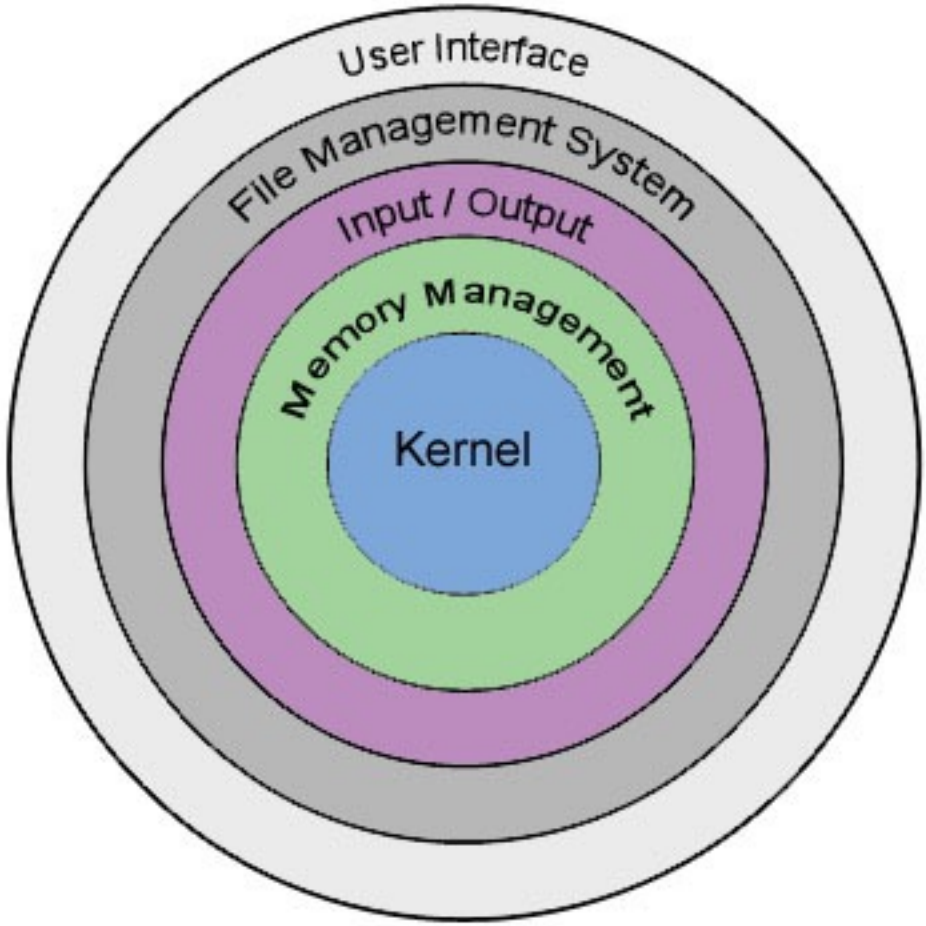
- The system is organized through **abstraction levels**, as a hierarchy of abstract machines
- **Hierarchy** is given by the **USE** relation



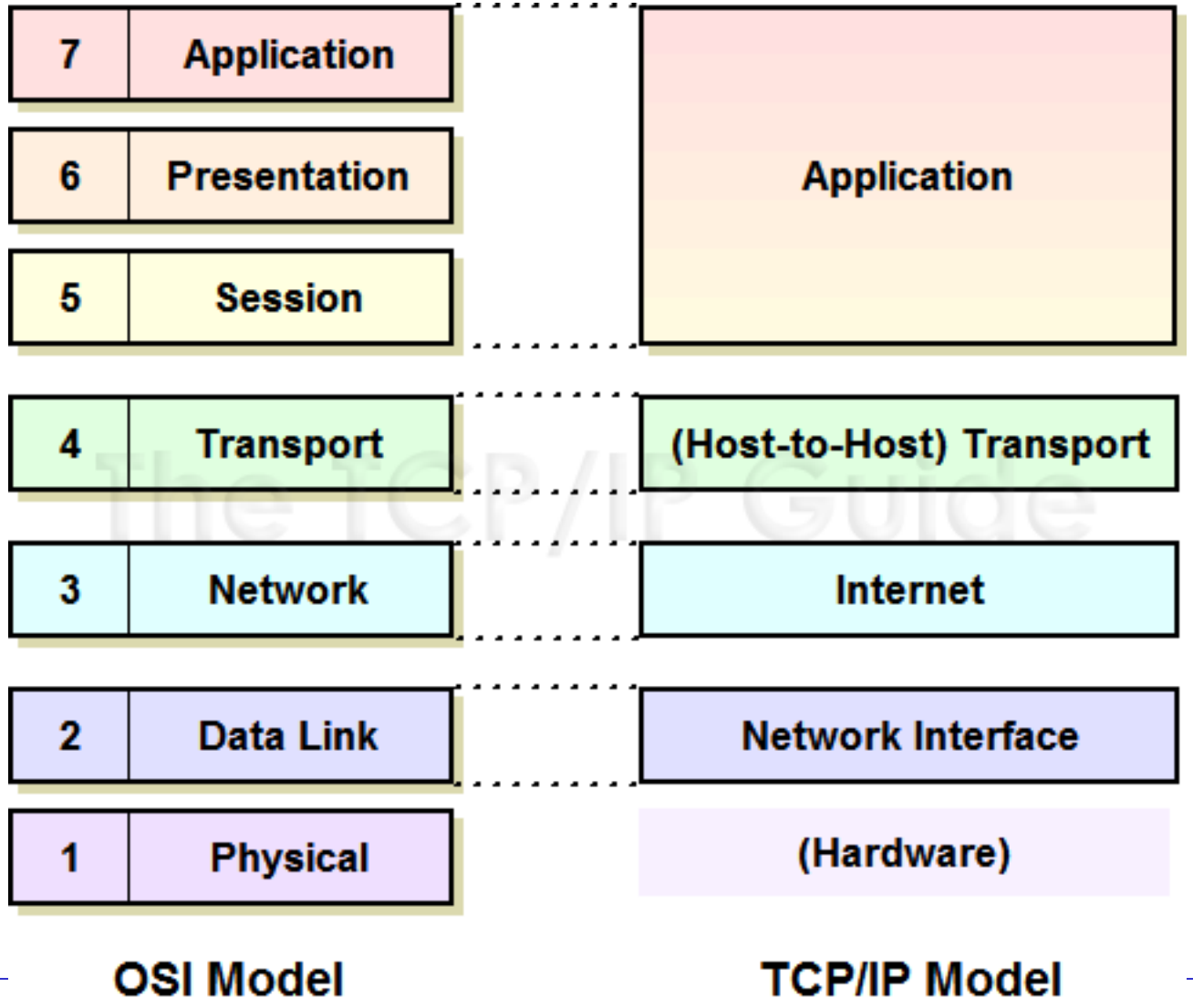
onion-ring structure



Well-known layered systems: An Operating System



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



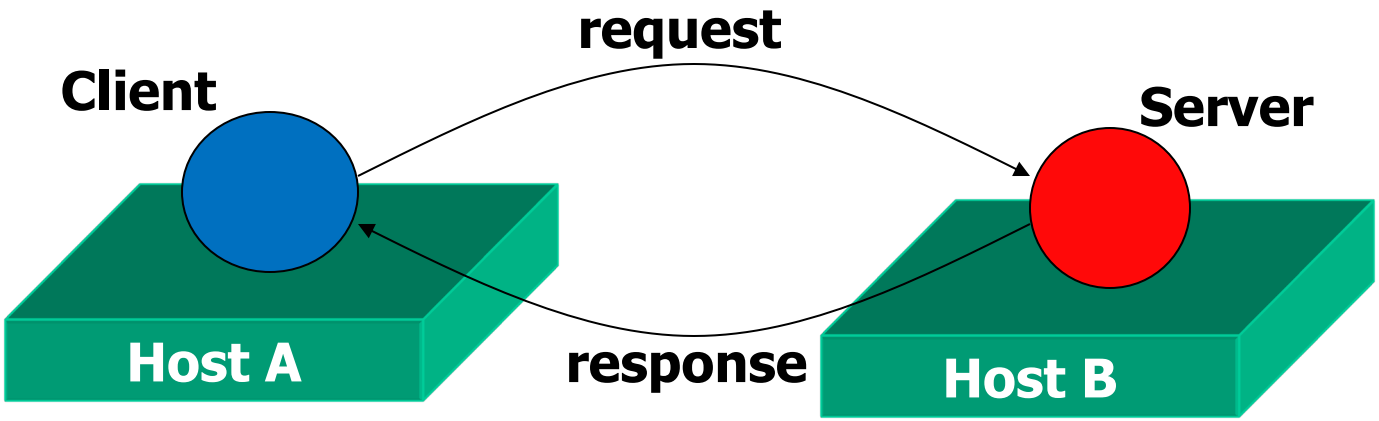
When to use a layered architecture

- Abstraction layers to dominate complexity
- When it is possible to identify a specific (bounded) concern for each layer
- When it is possible to identify clearly the communication protocols between layers

Client/Server

- Well-known architectural style for distributed applications
- Client and server are decoupled
  - Different processes with well-defined interface
  - Server is accessible only through interface
  - Server is unaware of specific client instances
  - Client and server have independent lifecycles
  - Client and server are defined by a set of hw/sw modules
  - May be distributed or run in the same compute unit
- Client and server play different roles
  - Server exports a set of services
  - Client uses the services and initiates communication
  - Communication through messages or through remote invocations
  - A client may be server for other components

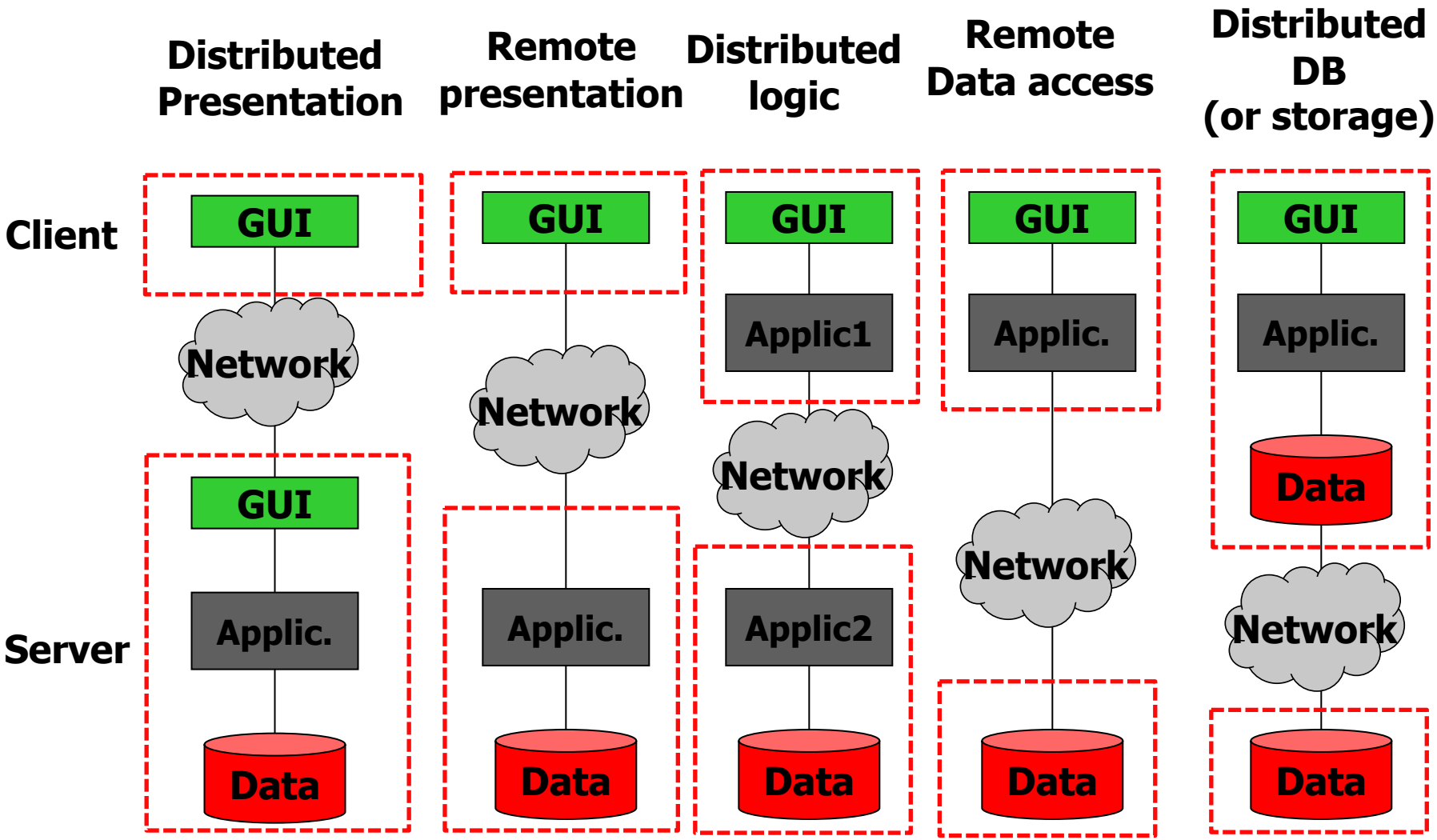
Client /Server: a general schema



When to adopt a client-server architecture

- Remote communication
- Separation of concerns between the clients and the server
  - Note that a client-server architecture is a kind of layered architecture with two layers, also called tiers in this case
- Possibility to execute on heterogeneous computational units (virtual/physical)
  - E.g., servers on powerful machines (complex computations), clients on commodity machines

Client/Server = two-tier architecture

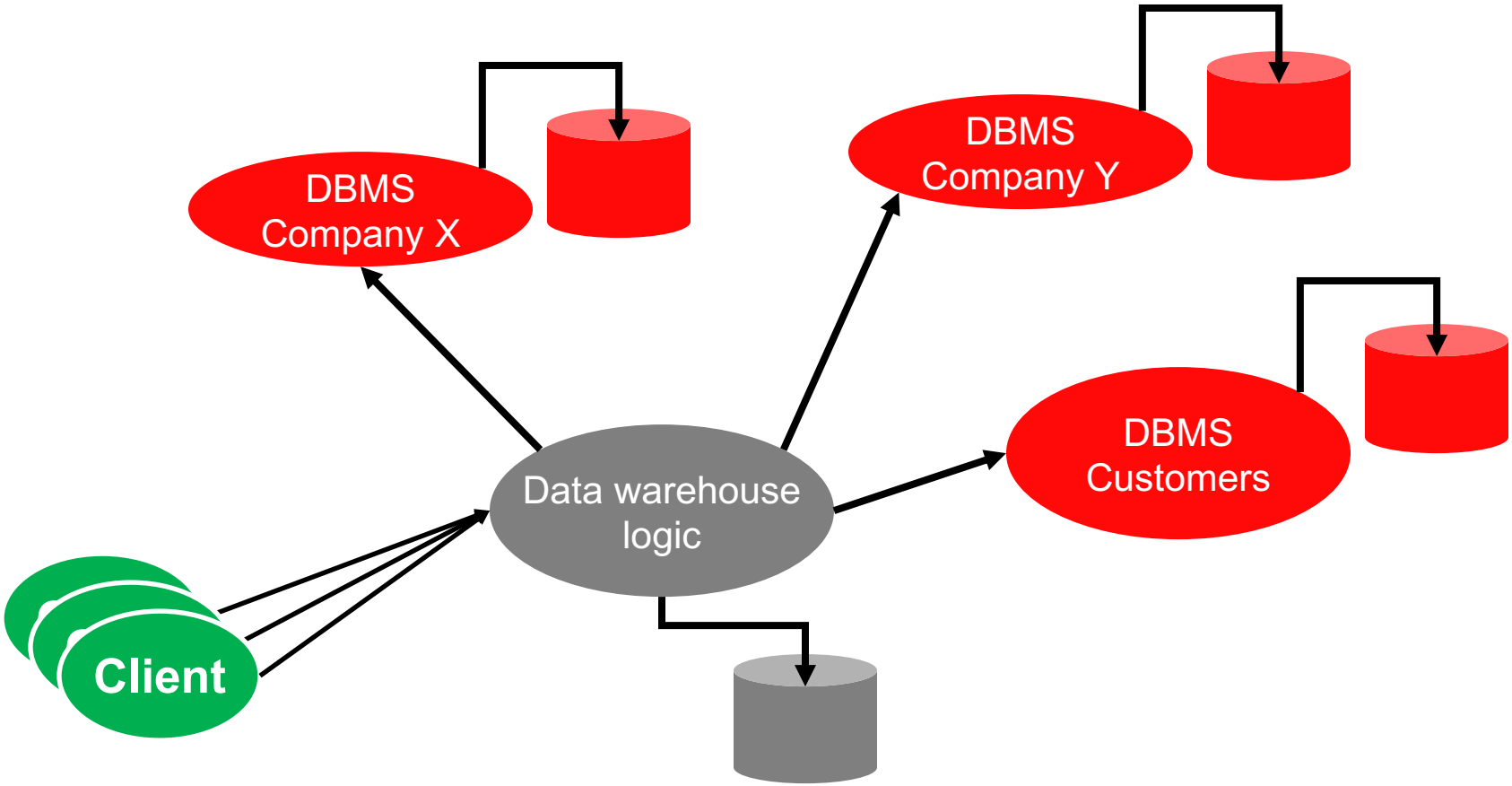


Three-tier architecture

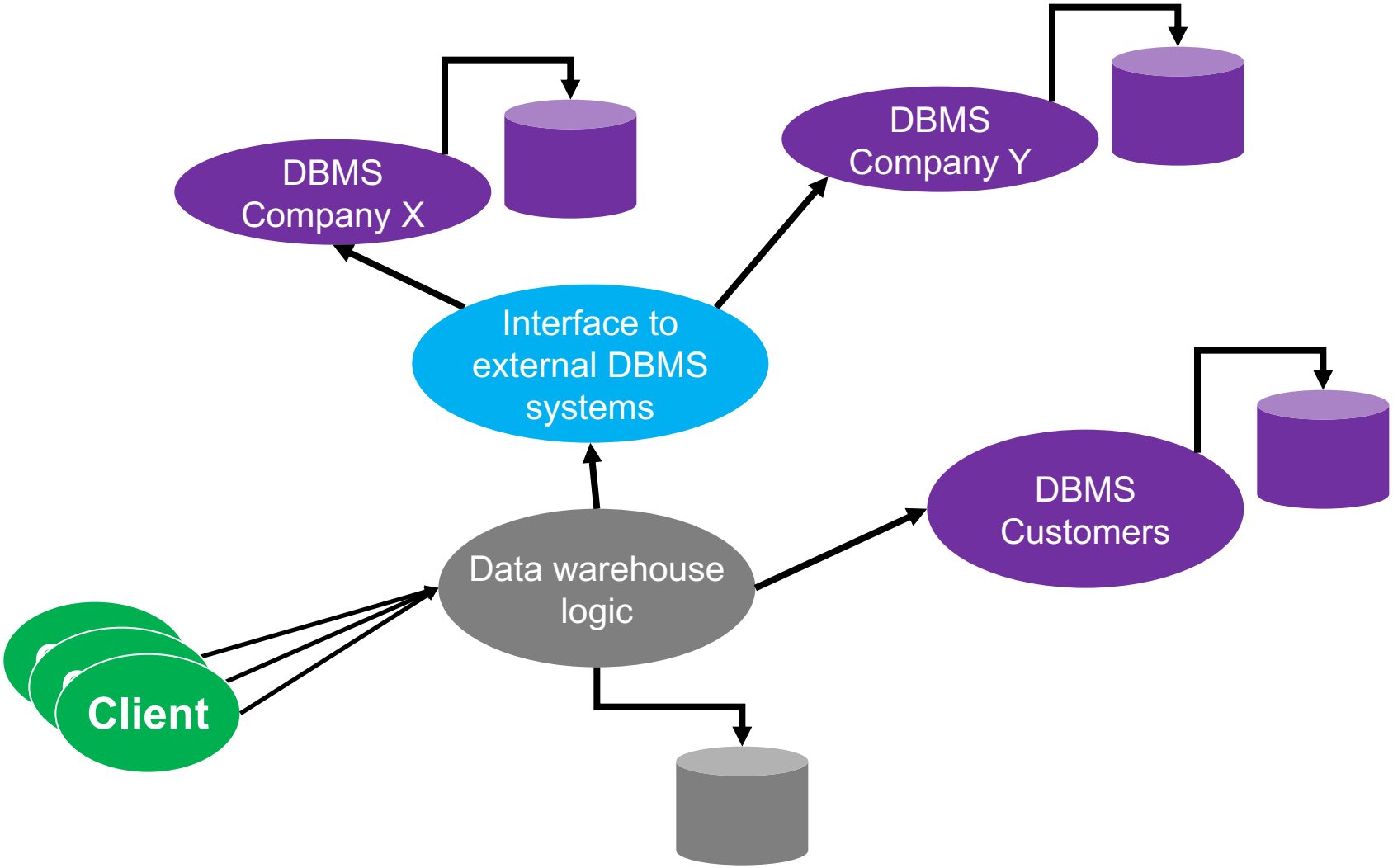
- In the typical scheme, the mid level has all the application logic
- Advantage
  - decoupling of logic and data, logic and presentation

An example

Data warehouse for a supermarket



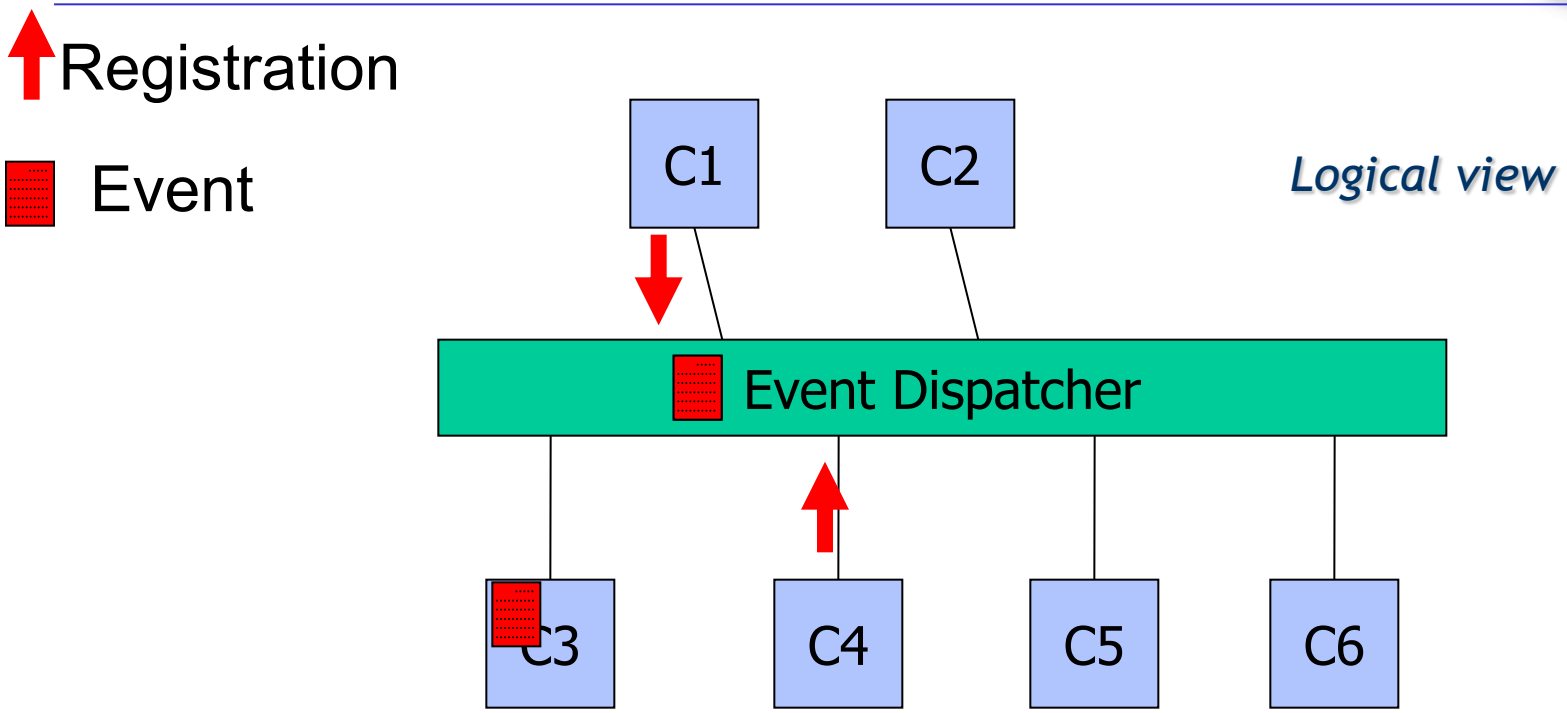
From 3 to N levels



Relevant technologies

- JEE / Jakarta EE framework
  - https://jakarta.ee/
- Spring framework
  - https://spring.io/

Event-based systems



- Components can register to / send events
- Events are broadcast 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 Pros & Cons

+ Very common in modern development practices
  + E.g., continuous integration / deployment (such as GitHub Actions)
+ Easy addition/deletion of components
  + Publishers/subscribers are decoupled
  + The event dispatcher handles this dynamic set
– Potential scalability problems
  – The event dispatcher may become a bottleneck (under high workloads)
– Ordering of events
  – Not guaranteed, not straightforward

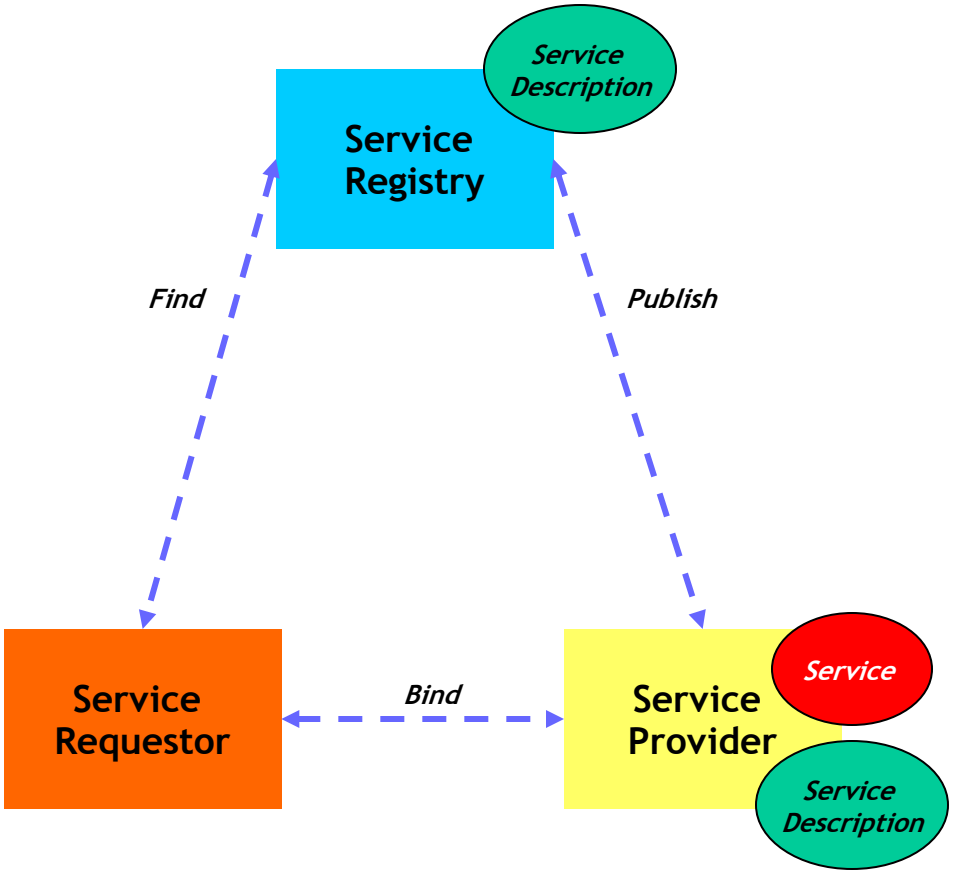
Other characteristics

- 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
- Flexible
  - one-to-many, many-to-one, many-to-many

Technologies

- Apache Kafka
  - https://kafka.apache.org/
- RabbitMQ
  - https://www.rabbitmq.com/

Service-oriented architecture (SOA)

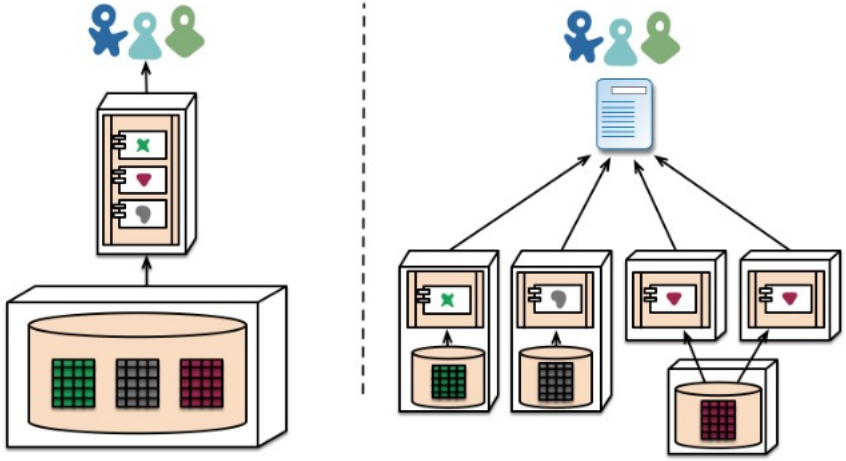


SOA Pros & Cons

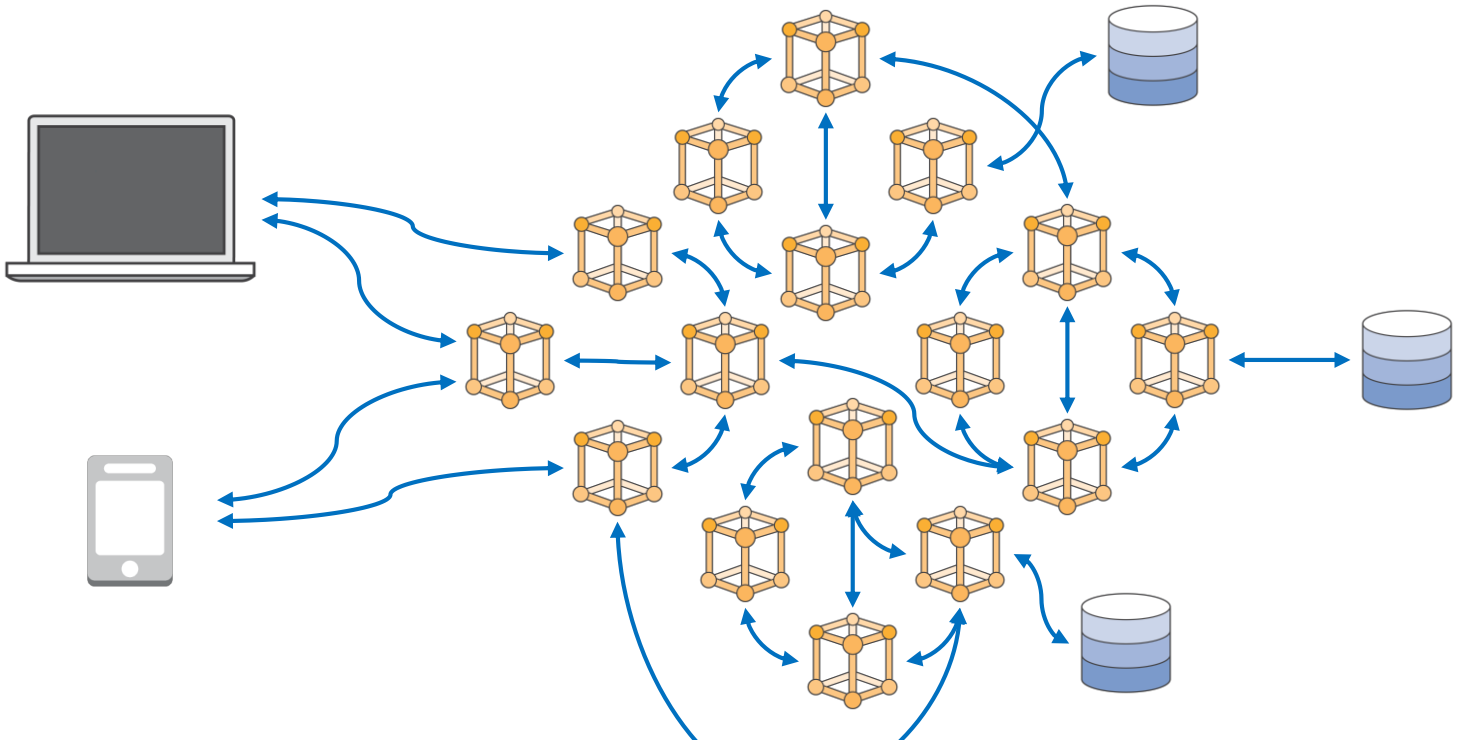
- Pros
  - Enables reuse of registries and providers across organizational boundaries
  - Relies on runtime discovery and dynamic binding
  - Services offered through a well-defined **Application Programming Interface (API)** → Interface documented in a machine-interpretable form
- Cons
  - Dynamic orchestration of discovered services is not trivial
    - A lot of glue code is needed
    - SOA systems are usually monolithic
  - SOAP (XML based protocol) considered too heavy

Microservice architectural style

- Evolution of SOA
  - https://martinfowler.com/articles/microservices.html
- Monolithic systems are decomposed into small specialized services and deal with a single “bounded context” in the target domain



Microservice architectural style



*"...the microservice architectural style is an approach to developing a single application as a suite of **small services**, each running in its own process and communicating with **lightweight mechanisms**, often an HTTP resource API"*

-- Martin Fowler

The starting point for microservices - monolithic architecture

Lots of users → high workload

Load Balancer

Receives requests and sends them to proper replica

monolithic replicas

Monolithic App (e.g., e-commerce)

- Account Component
- Catalog Component
- Recommendation Component
- Customer Service Component

Database

Big problems (1/2)

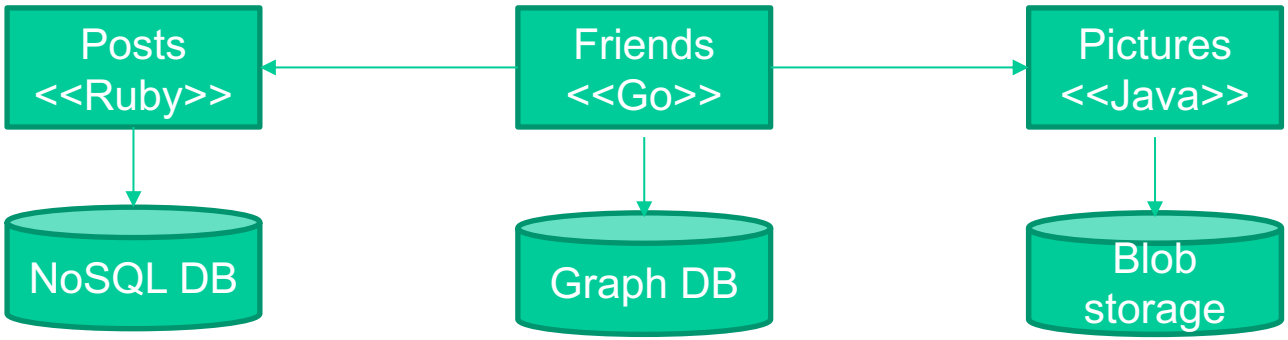
- Issues with frequent deployments
  - Need to redeploy everything to change one component
  - Interrupt long running background jobs
  - Increase risk of failure
- Overloaded containers
- Obstacle to scaling development
  - Difficult fault localization
  - Difficult maintenance
- Require long-term commitment to a technology stack

Big problems (2/2)

- Forbid fine-grained scaling strategies
  - It is not possible to scale “Product Catalog” up differently from “Customer Service”
- May lead to availability issues
  - [~2008] Netflix reported that a single minor mistake (i.e., missing “;”) brought down the whole platform for many hours

Microservices key benefits: technology heterogeneity

- Each service uses its own technology stack
  - Technology stack can be selected to fit the task best
    - E.g., Data analysis vs Video streaming
  - Teams can experiment with new technologies within a single microservice
    - E.g., we can deploy the two versions and do A/B testing
  - No unnecessary dependencies or libraries for each service

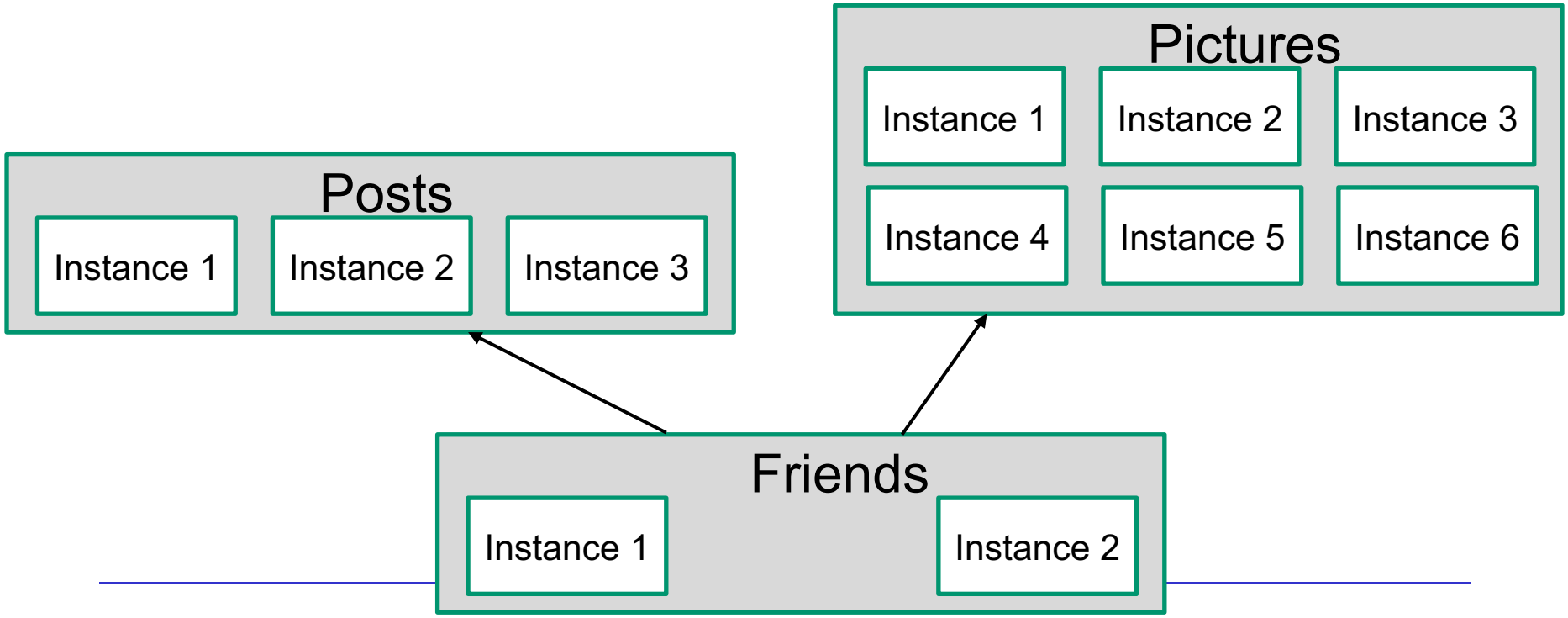


Microservices key benefits: resilience

- If a microservice fails the others can still work, possibly with a degraded functionality (until the failure is resolved)
- Note: microservices can introduce other types of failures related to network and communication (microservices systems are typically more “chatty”)

Microservices key benefits: scaling

- Each microservice can be scaled **independently**
- Identified bottlenecks can be addressed directly
  - Parts of the system that do not represent bottlenecks can remain simple and un-scaled



Microservices key benefits: independent codebases and deployment

- Each service has its own software repository
  - Favor distributed teams
  - No cross-dependencies between codebases
- Tools work faster
  - Building, testing, refactoring, deployment takes seconds
- Each service can be deployed independently
  - Independent CI/CD
- Startup takes a smaller amount of time

Microservices key benefits: reuse and composability

- The functionality offered by a microservice can be used and reused in multiple contexts
  - E.g., authentication
- It is possible to compose multiple microservices in different ways

Microservices key benefit: replaceability

- Monolithic systems are often big and complex and their replacement is risky and costly

- Given its small size, replacing a microservice implementation is much easier
  - The team can develop a new implementation
  - As soon as the service is ready for operation, it can be moved in the operational environment
  - The previous service can be shutdown

Evolution at NETFLIX

2008

- Single DataCenter
- Everything in one WebApp

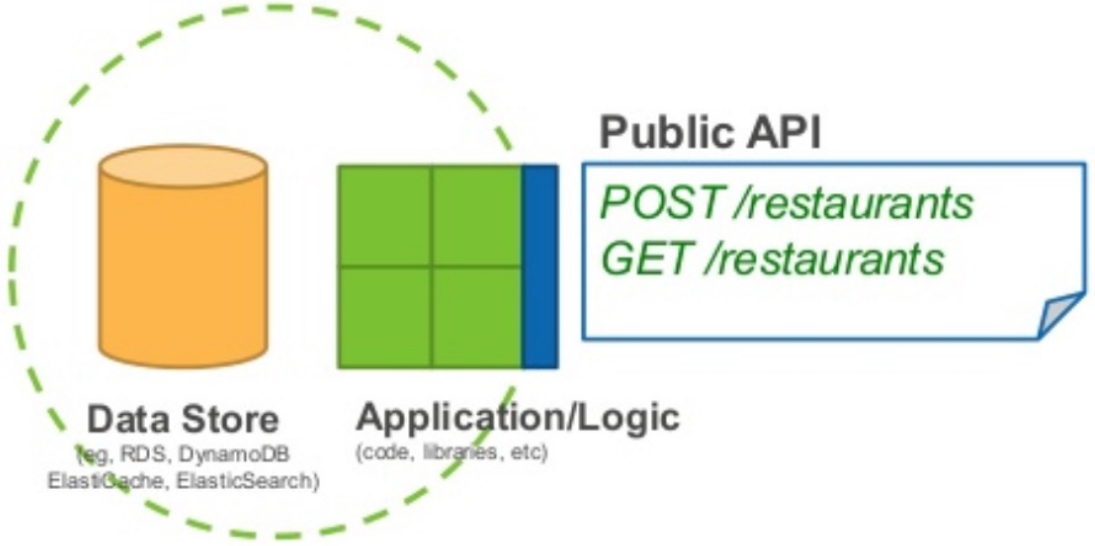
~2010

- AWS Cloud
- 100s of fine-grained Services

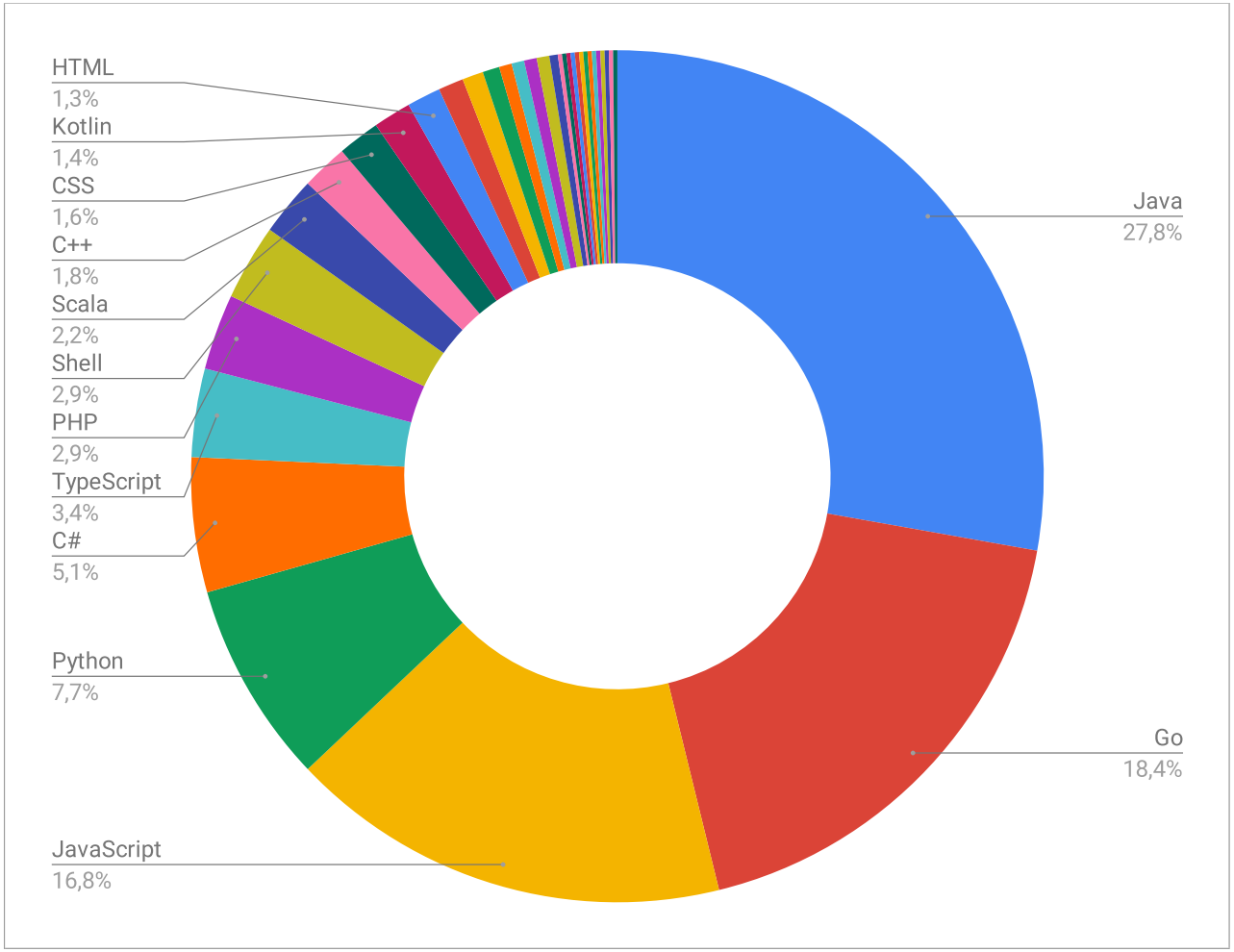
Current scale at NETFLIX

- 30% of the Internet traffic
- 500+ microservices
- 2+ billion API gateway requests daily
- Each Gateway API call requires average six calls to backend services
- Over 800 different client devices

Anatomy of a microservice



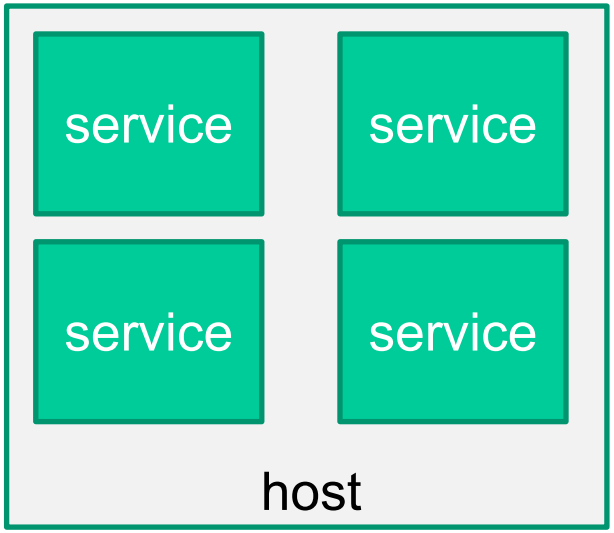
Used languages

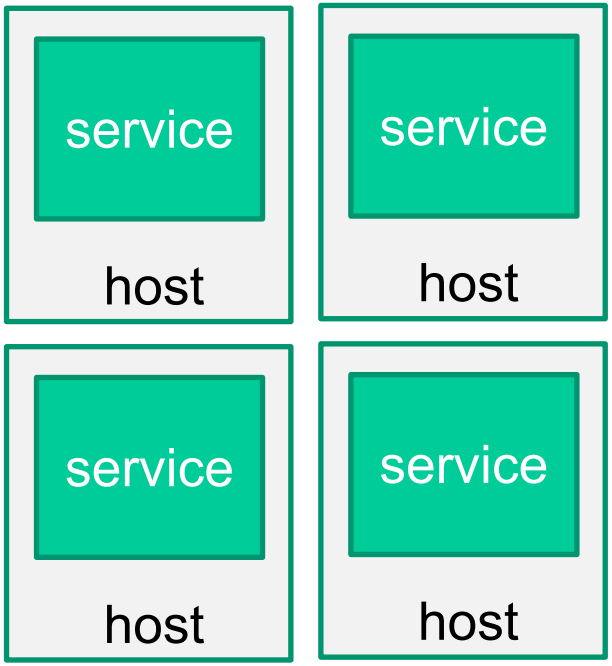


Critical issues with microservice architectures

- We deal with distributed systems
- Decomposition of monolithic systems require
  - Careful identification of microservices
  - Communication, integration and distributed transactions
  - Careful definition of the guiding principles and practices for the microservice architecture (microservices patterns/antipatterns)
  - Deployment
  - Testing
  - Monitoring

Multiservice vs single service per host deployment

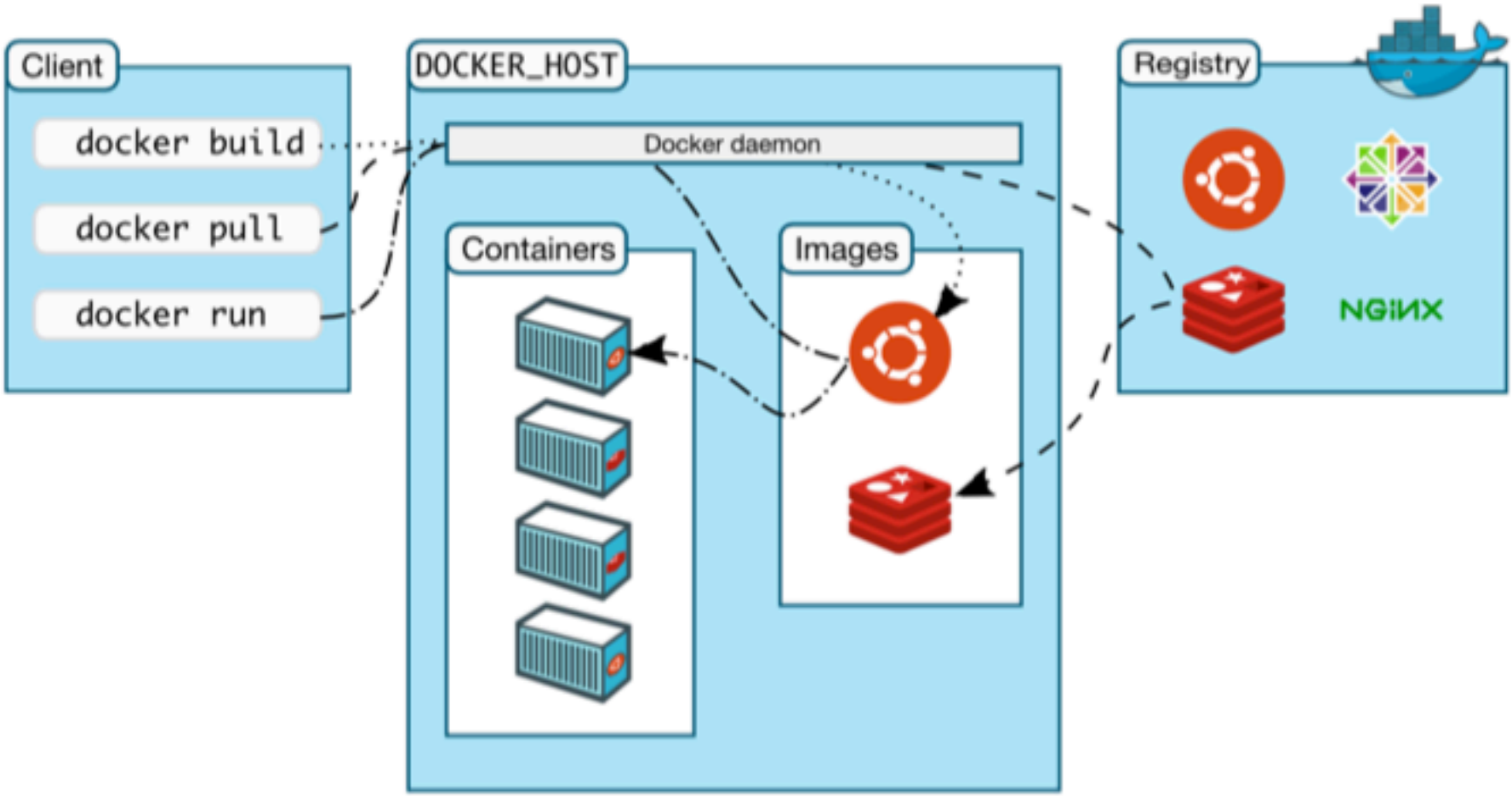




Requires more resources
Makes scalability easier and even

- Various mechanisms available for host virtualization

Possible container technology - Docker



Politecnico
di Milano

… More Architectural Styles: Cloud Patterns

http://www.cloudcomputingpatterns.org

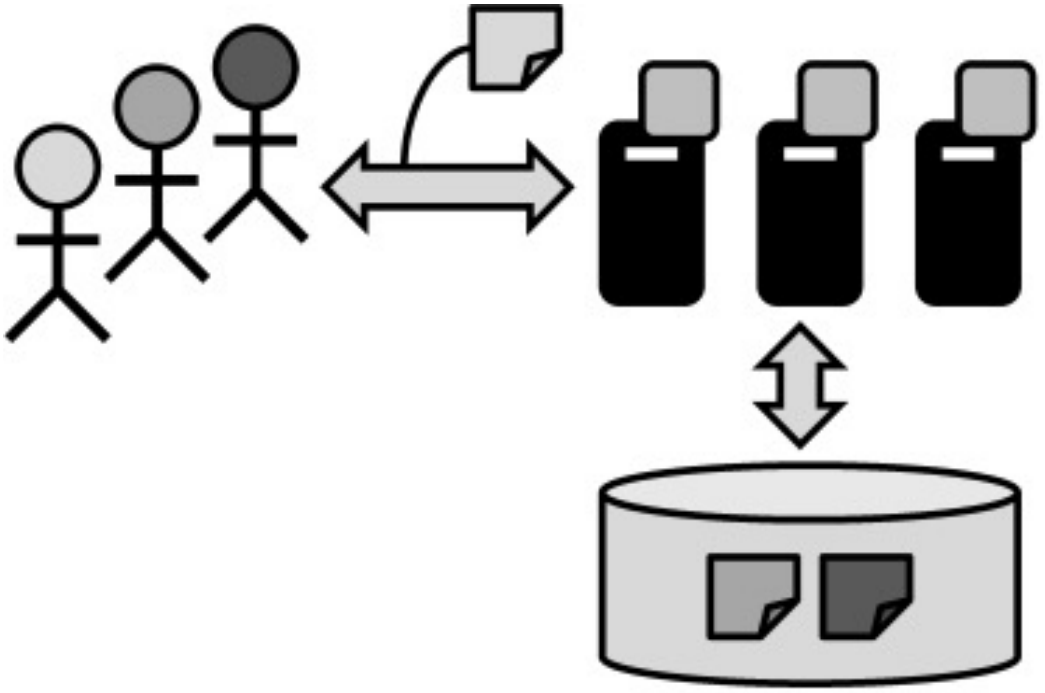
Classification of patterns

- 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

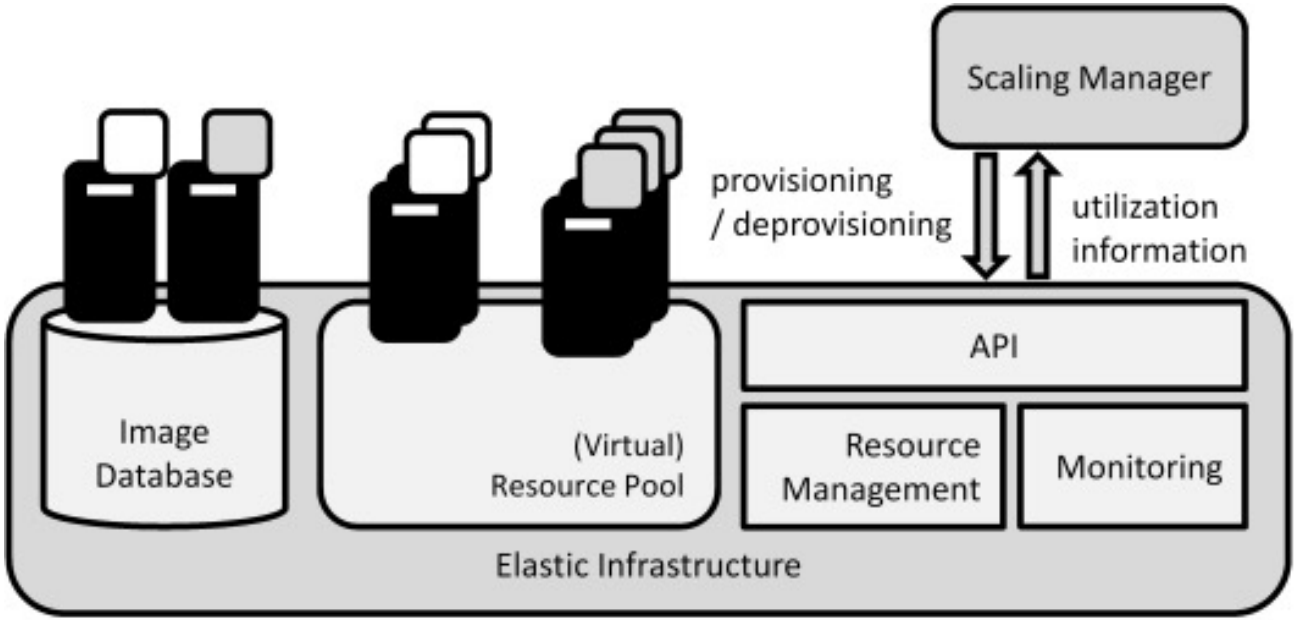


- No data is lost if an instance fails
- Multiple components have a common internal state
- The scalability of the central data store becomes the major challenge
  - Distributed data stores

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

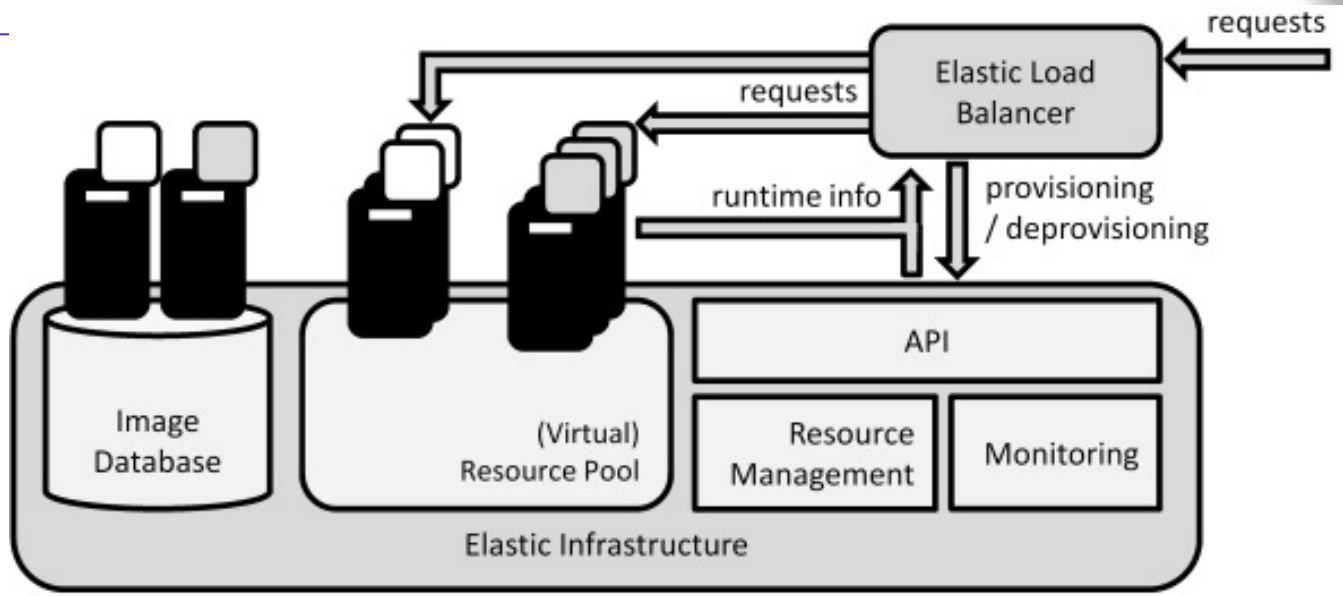


- If the utilization of compute nodes exceeds a specified threshold, additional hosting components are provisioned that contain the same application component.
- If the components are implemented as *stateless components*, the operations for adding and removal of components are significantly simplified.

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

Assignments for the next class

- Watch the videos you find here
  - On software design descriptions and principles: https://polimi365-my.sharepoint.com/:v:/g/personal/10143828_polimi_it/EaE5ix1izNxGiLLtzBsr7GABI6M_BqvtBPeH4qtuVMOnhQ?e=hydFes
  - On the design process: https://polimi365-my.sharepoint.com/:v:/g/personal/10143828_polimi_it/EcbU-pgqCblMiw2hIZIUFSABffi1zSZCjXuYrgt1cQfntw?e=ebYZC9
- They will be used as a basis for discussion in class next time