POLITECNICO
MILANO 1863

Software Engineering 2

Microservices

Routing, resilience, security, communication patterns

M Camilli, E Di Nitto, M Rossi

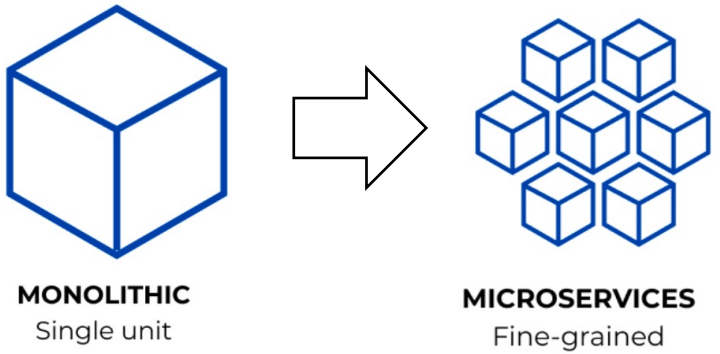
POLITECNICO
MILANO 1863

Architectural Styles

Microservices

Microservice architectural style

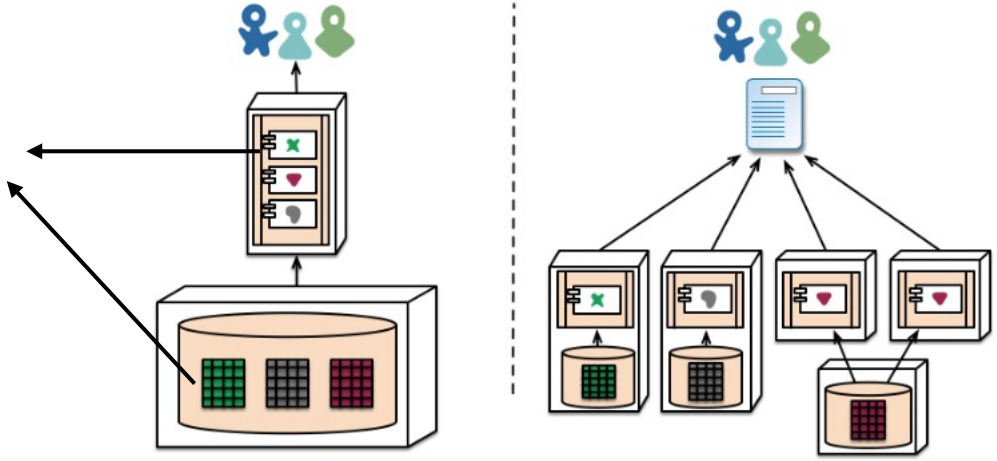
- **Before microservices**
  - Monolithic systems
  - Applications delivered as **single deployable** software **artifacts**: UI, business, and database access logic packaged together into a single artifact deployed to an application server
- *"...the microservice architectural style is an approach to 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: https://martinfowler.com/articles/microservices.html



Microservice architectural style

- Microservices
- Monolithic systems are decomposed into small specialized services and deal with a single **bounded context** in the target domain

Bounded context →
single concern according to the application domain (single area of responsibility)



Microservice architectural style

- Successful stories
  - Netflix
  - Amazon
  - eBay
  - Spotify
  - …
  - [2010-2016] They all **migrated** from monolithic systems to microservices
    - **Main goal**: improve both process and product to support **massive scale of operation** (millions of users)

From monolithic architectures to microservices

Load Balancer

Monolithic App

Account Component

Catalog Component

Recommendation Component

Customer Service Component

Database

Gateway

Account Component

Account Component

Recommendation Component

Recommendation Component

Recommendation Component

Recommendation Component

Catalog Component

Catalog Component

Catalog Component

Customer Service Component

Microservices: advantages (product)

POLITECNICO MILANO 1863

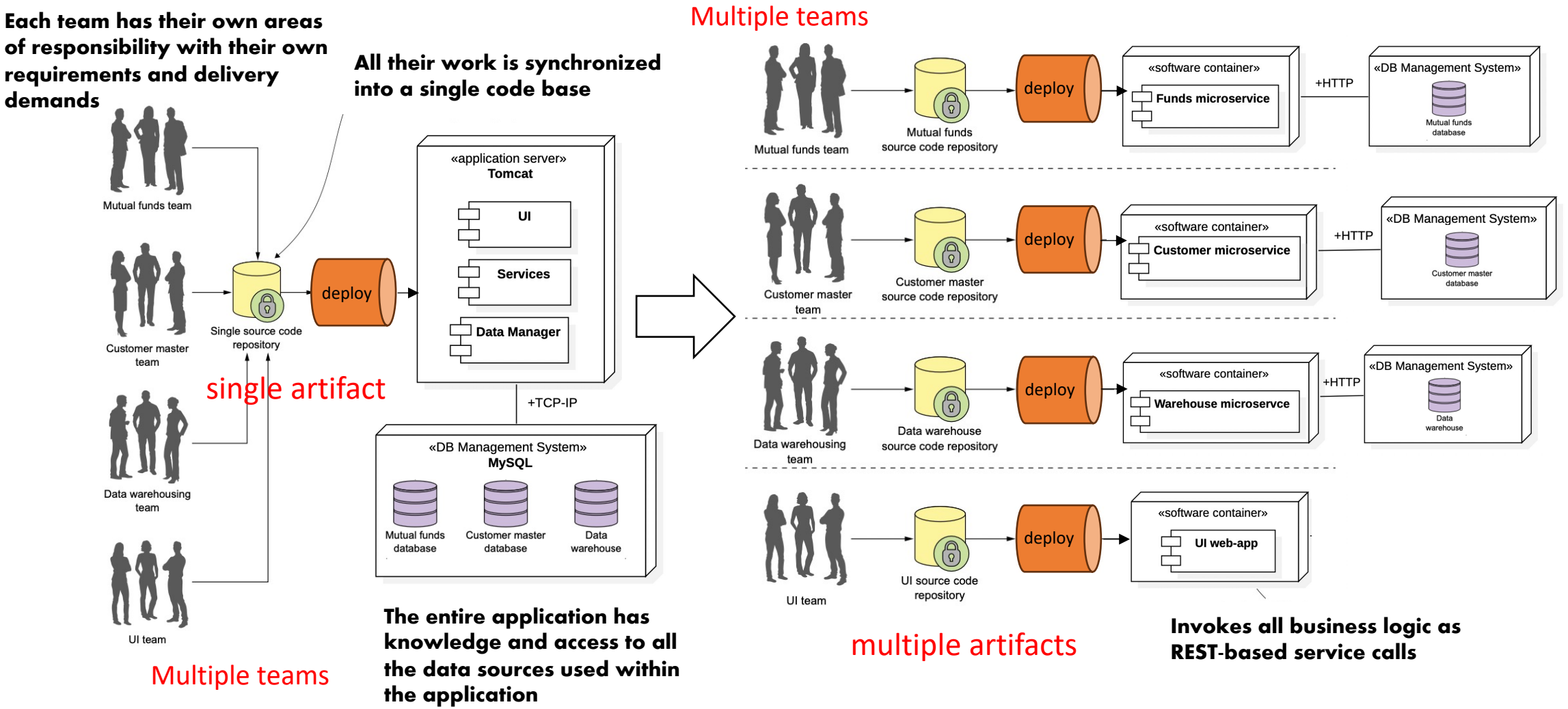
- Enable fine-grained scaling strategies
  - In monolithic systems, selective replication is not possible, the entire system must be replicated as a whole
  - Microservices enable flexible deployment and selective replication
- Reduce the scale of localized issues
  - **Example**: availability issues
    - **[2008] Before migration** to microservices, Netflix reported that a single minor syntax error in the codebase brought down the whole platform for many hours
- Improved resilience
  - If a microservice fails the others can still work, possibly with a degraded functionality (until the failure is resolved)

Microservices: advantages (product)

POLITECNICO MILANO 1863

- Better 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 to realize different workflows

Microservices: advantages (process)



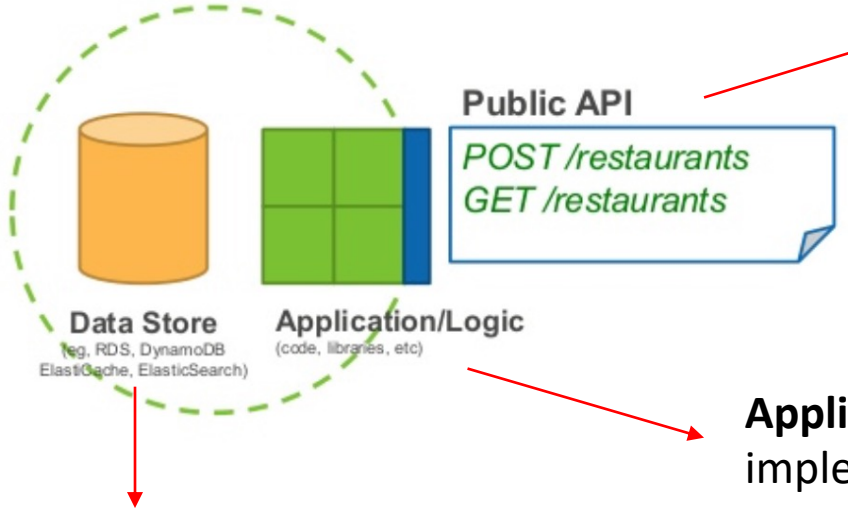
Microservices: advantages (process)

POLITECNICO MILANO 1863

- Reduced teams synchronization overhead
- Organizations have small development teams with well-defined areas of responsibility
- Underlying technical implementation of the service is irrelevant because the applications always communicate through technology-neutral protocols (REST APIs)
  - Teams decide languages and technologies according to expertise and purpose (e.g., data analysis vs video streaming)
  - Teams can experiment with new technologies on a single microservice
- Smaller codebase → easier debugging, cheaper maintenance

Anatomy of a microservice

3 main elements of a microservice



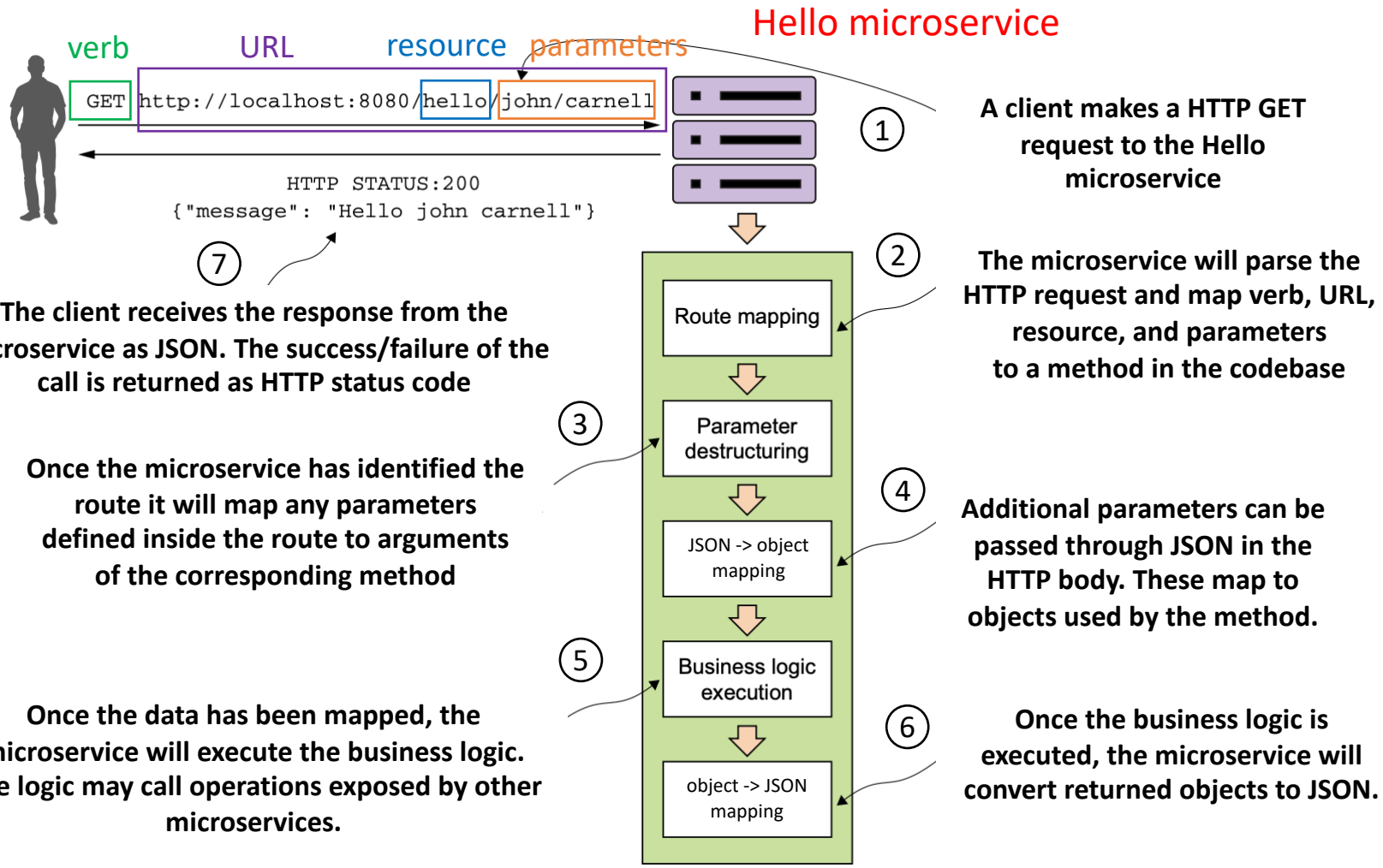
REST API
exposes core operations of the service:
- different types (HTTP verbs: GET, POST, PUT, and DELETE)
- lightweight data serialization (JSON format) to send and receive data

Application (or business) logic
implementation of core operations executed upon requests

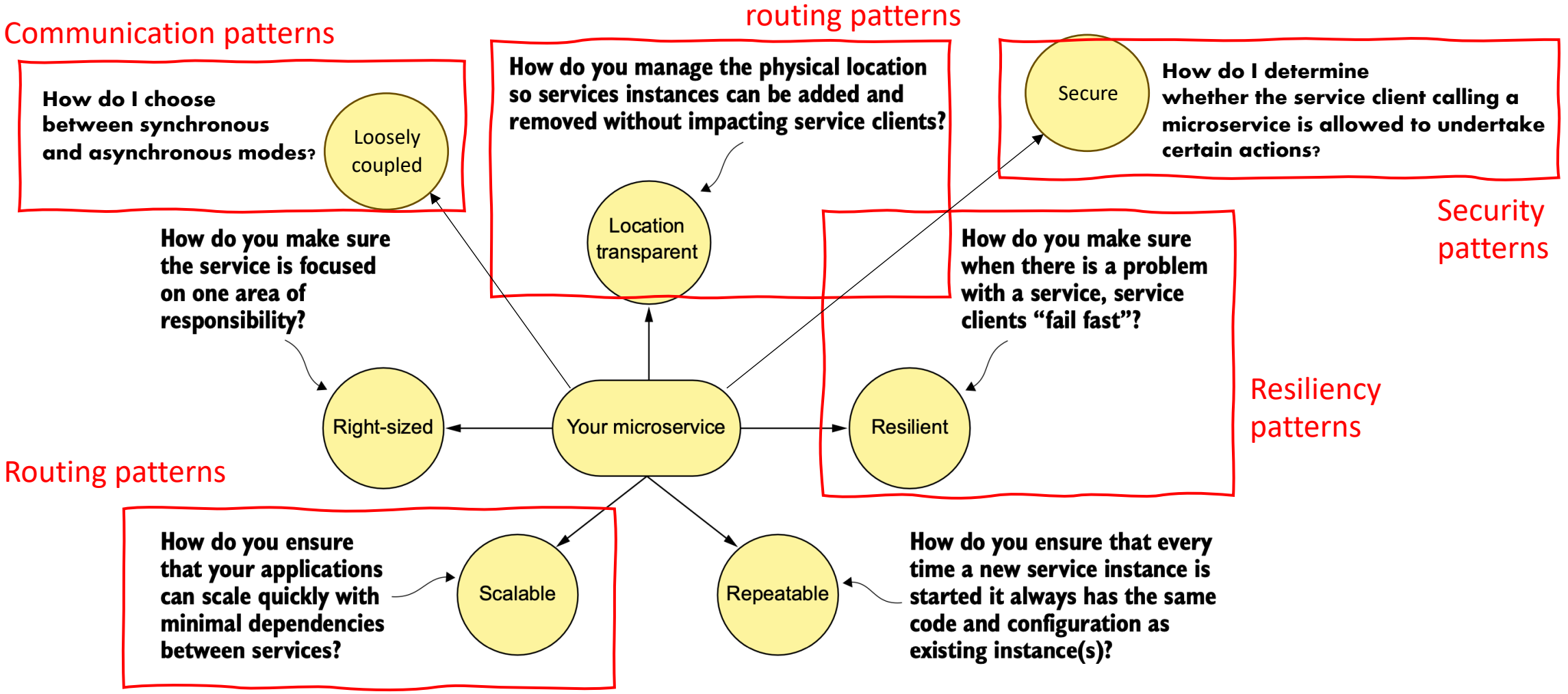
Data storage
each microservice typically has its own local data (limited data sharing, no global DBs)

Anatomy of a microservice (recap)

Common workflow of REST operations



Microservices: besides business logic



Microservices: routing patterns

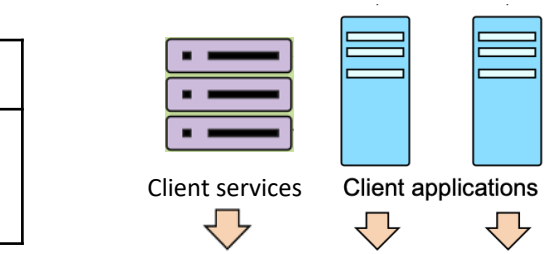
POLITECNICO MILANO 1863

- **Execution environment has shared + not pre-allocated** resources →
  - Physical location of running services is potentially unknown
  - Services need to be discovered
- Service discovery must be
  - **Highly available**: cluster of nodes to avoid single point of failure (if a node becomes unavailable, other nodes in the cluster take over)
  - **Load balanced**: service invocations are spread across all the service instances
  - **Resilient**: if service discovery becomes (temporarily) unavailable, applications should still function and locate the services
  - **Fault-tolerant**: should monitor the health status of services and take action without human intervention

Microservices: routing patterns

- Service discovery architecture (1) — general concepts

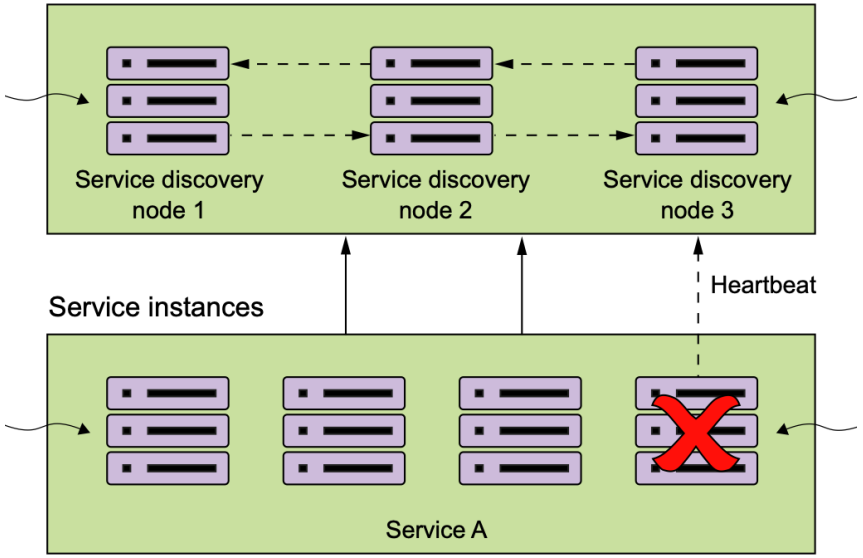
Logical name	Service ID	IP	Port
/api/v1/serivicea	ServiceA	172.18.32.100 172.18.32.100 172.18.32.101	8092 9092 9092



Clients do not have direct knowledge of service locations.

Cluster of nodes

1. A service location can be looked up by a logical name from the discovery nodes.



2. Service discovery nodes share information (location/health) among each other. Propagation can use static lists, or P2P "gossip" protocols.

Dynamic set (instances are added/removed)

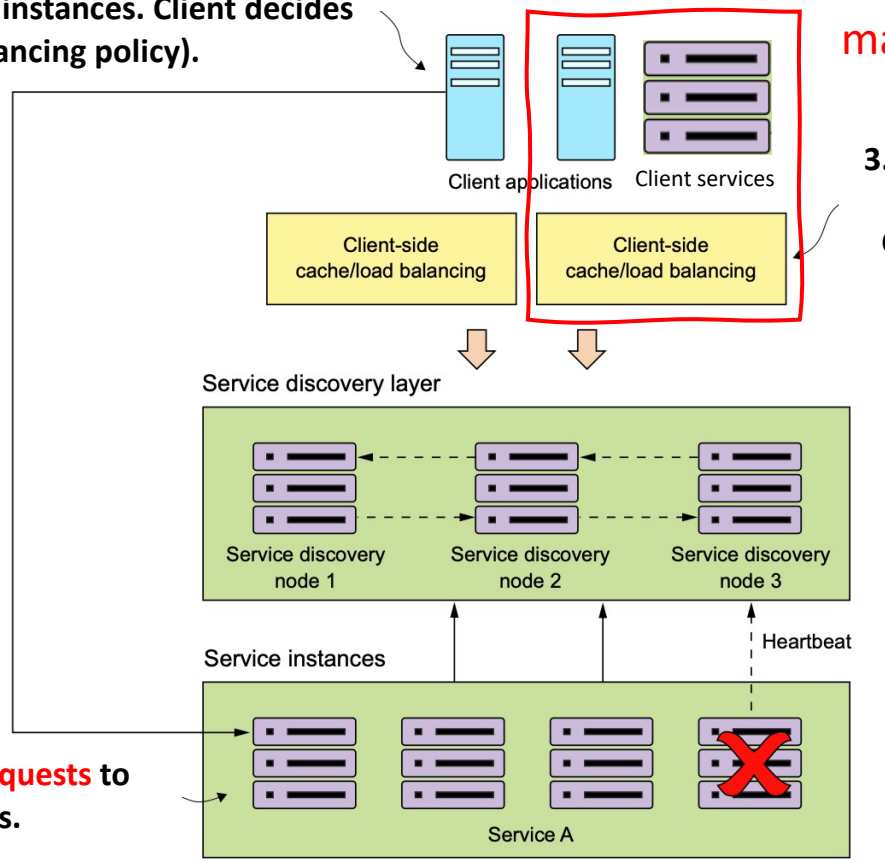
0. When a service instance comes online it registers its location (IP/port) to one discovery service instance. Instances of the same service are registered under the same service ID.

3. Service instances send periodic heartbeat (push mode) to service discovery. If an instance dies, the discovery layer removes its location.

Microservices: routing patterns

- Service discovery architecture (2) – client-side cache and load balancing

1. Client checks its local cache for the location of service instances. Client decides how to spread requests to instances (load balancing policy). Usually "round-robin."



3. Periodic refresh of cached data by contacting the discovery engine. Client cache is eventually (but not always) consistent.

2. Client sends direct requests to service instances.

Microservices: routing patterns

POLITECNICO MILANO 1863

- Service discovery architecture with cache — possible issues
  - Does not ensure **consistency** of cached data →
    - When a client contacts a dead instance, the client invalidates the local cache and forces a refresh by contacting the service discovery engine.

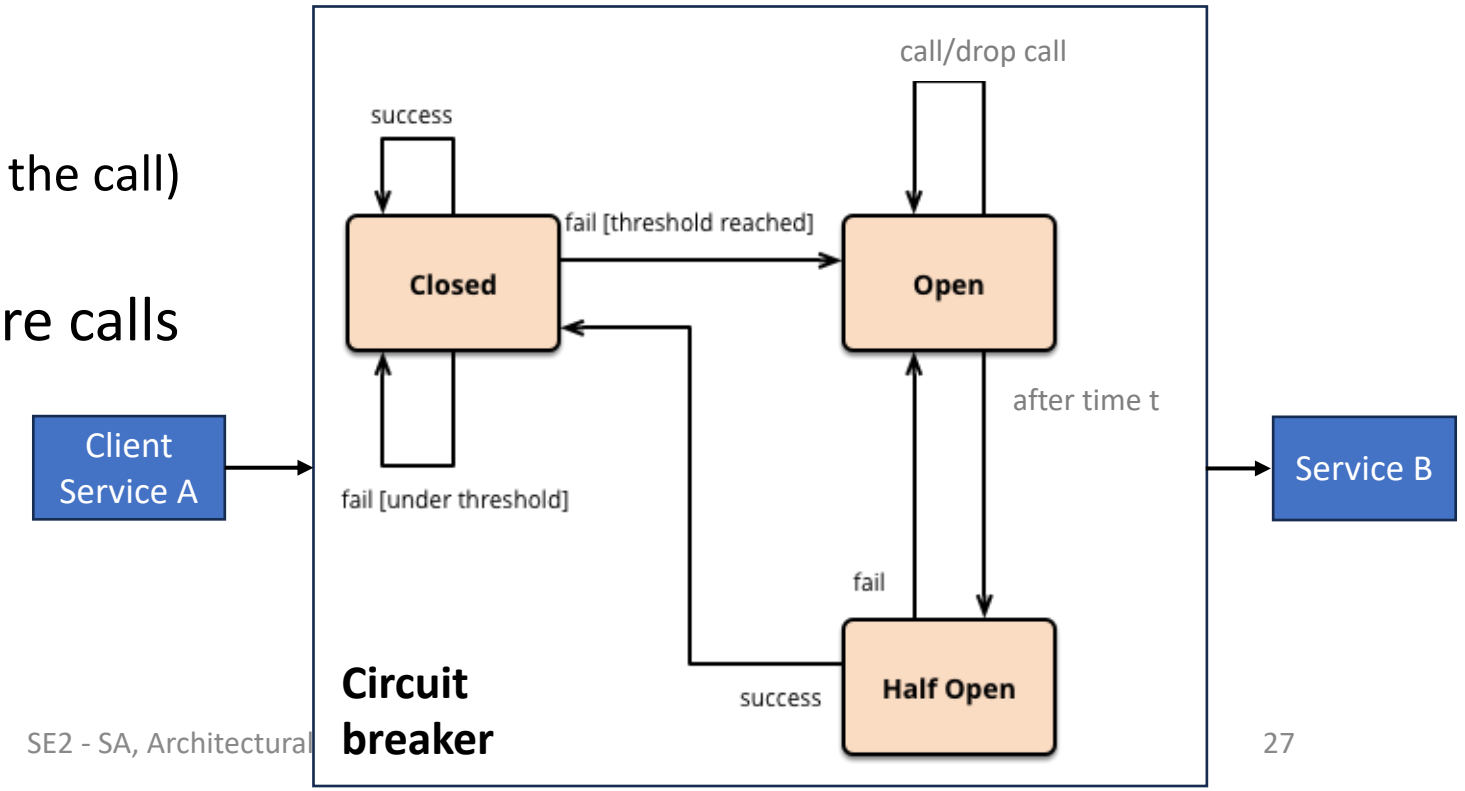
Microservices: resiliency patterns

POLITECNICO MILANO 1863

- "How do I make sure when there is a **problem** with a service, clients can avoid it before **recovery**?"
  - Service discovery provides some degree of resilience in the simple case in which a service instance dies (no heartbeat)
  - There are other subtle issues, for instance, remote resources could:
    - Throw errors (e.g., temporary bursts of exceptions)
    - Perform poorly (e.g., temporary slowdown)
- Goal: allow clients to "fail fast"
  - Avoid useless resource consumption
  - Avoid ripple effects

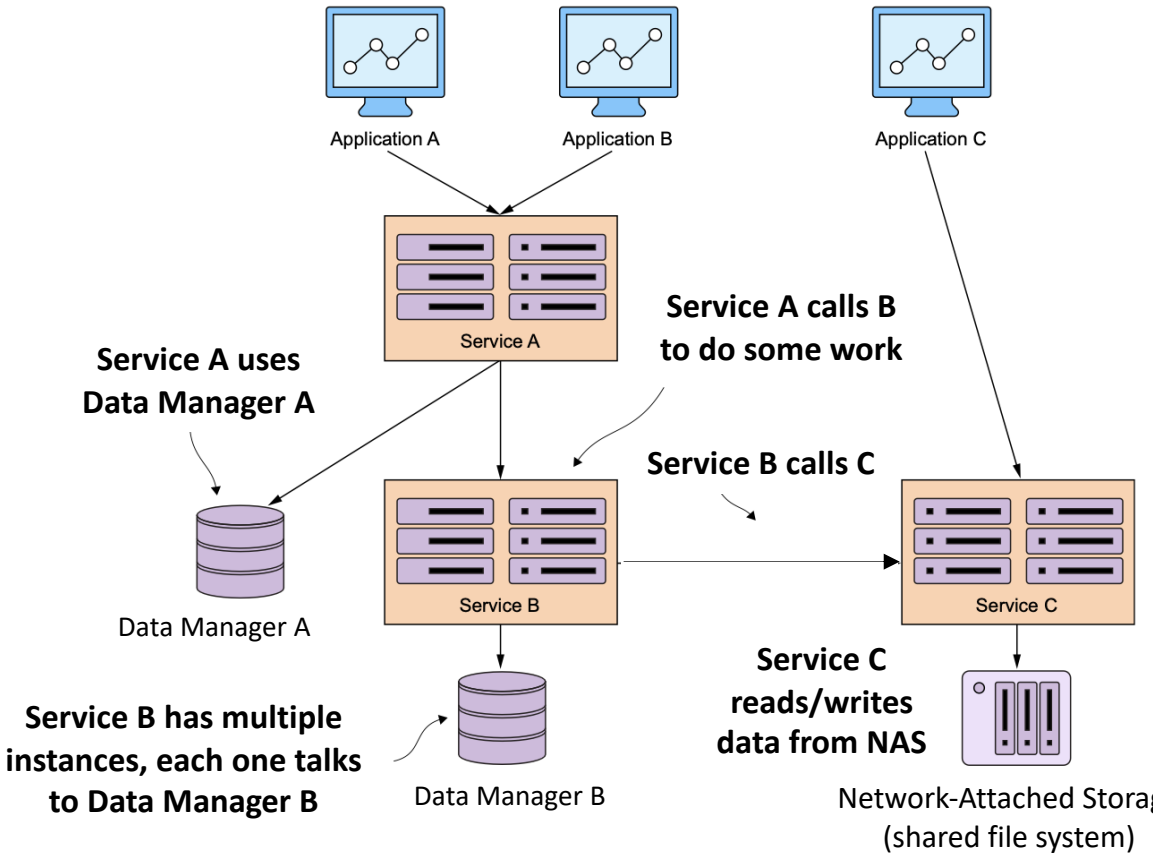
Microservices: resiliency patterns

- Circuit breaker (CB) — **Client-side** resiliency pattern
  - CB acts as a **proxy** for a remote service
  - When a remote service is called, the CB **monitors** the call
  - **Possible failures**:
    - CB receives 5xx error
    - Call takes «too long» (CB kills the call)
  - «**too many**» **failures** → circuit breaker inhibits future calls



Microservices: resiliency patterns

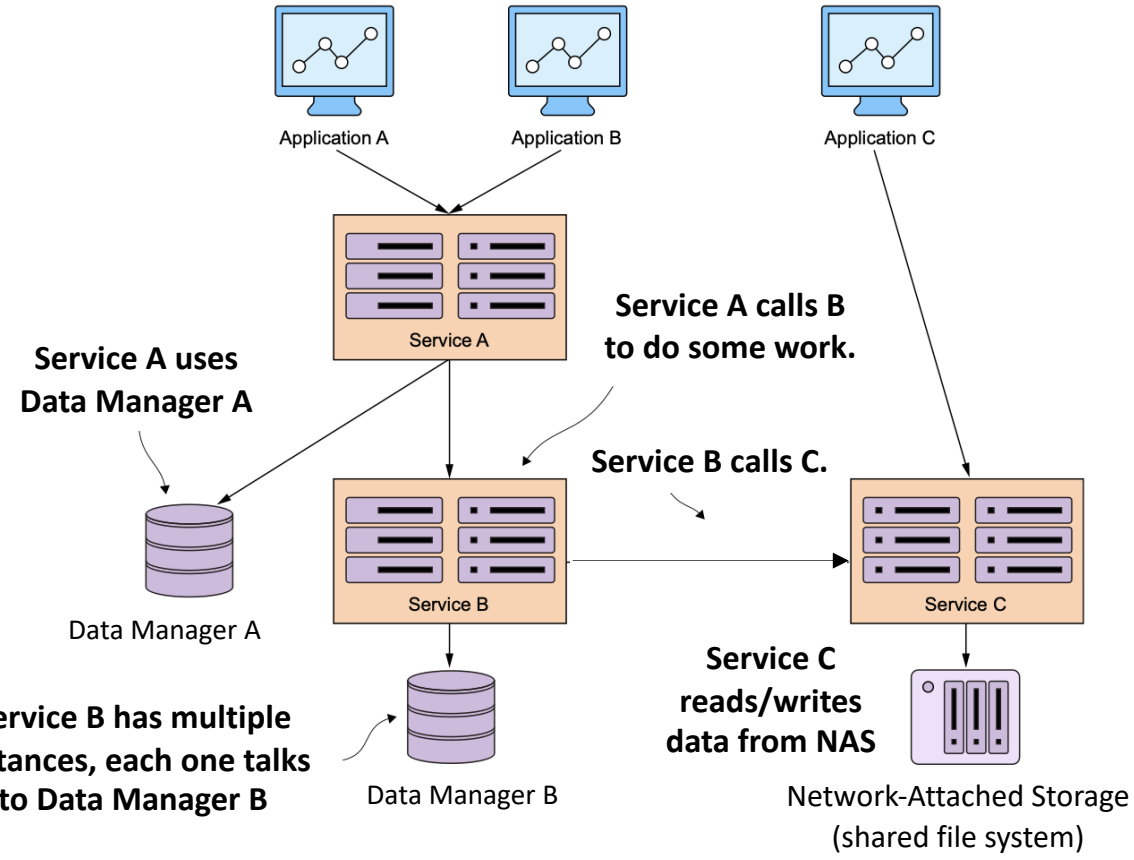
- Example — circuit breaker in action



- Scenario
  - During the night operators change the config of NAS
  - The day after, reads to a particular disk subsystem start performing extremely slowly
  - Developers of Service B did not anticipate slowdowns occurring with calls to C
  - Service B writes into Data Manager B and gets data from Service C within the same transaction

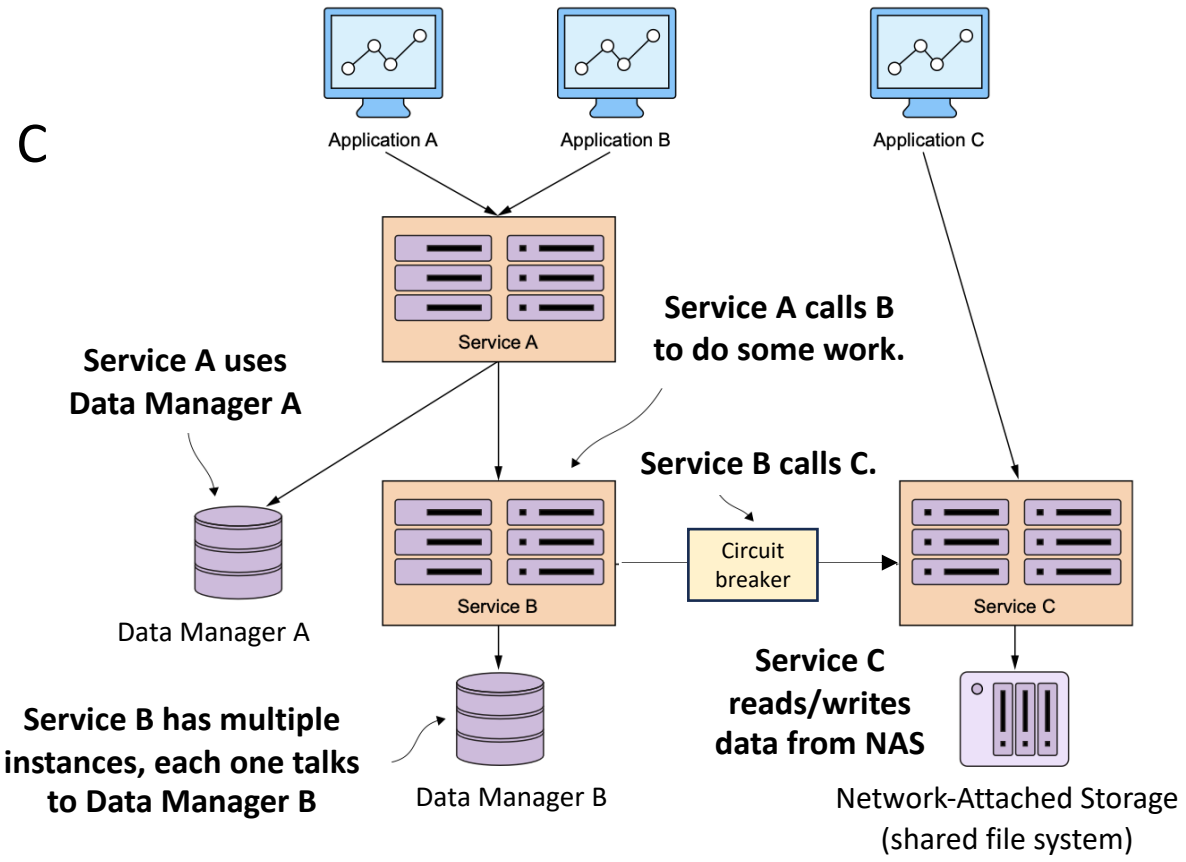
Microservices: resiliency patterns

- Example — circuit breaker in action
- **What happens? (ripple effect)**
  - C starts running slowly
  - Pending requests to C grow
  - Number of concurrent DB connections grow
  - Resources become exhausted because Service C never completes
  - Service A starts running out of resources because it calls B that is slow because of C



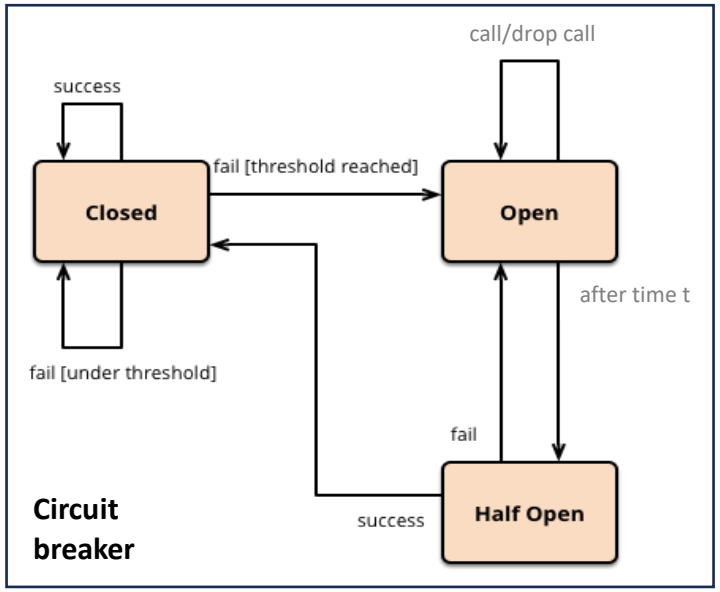
Microservices: resiliency patterns

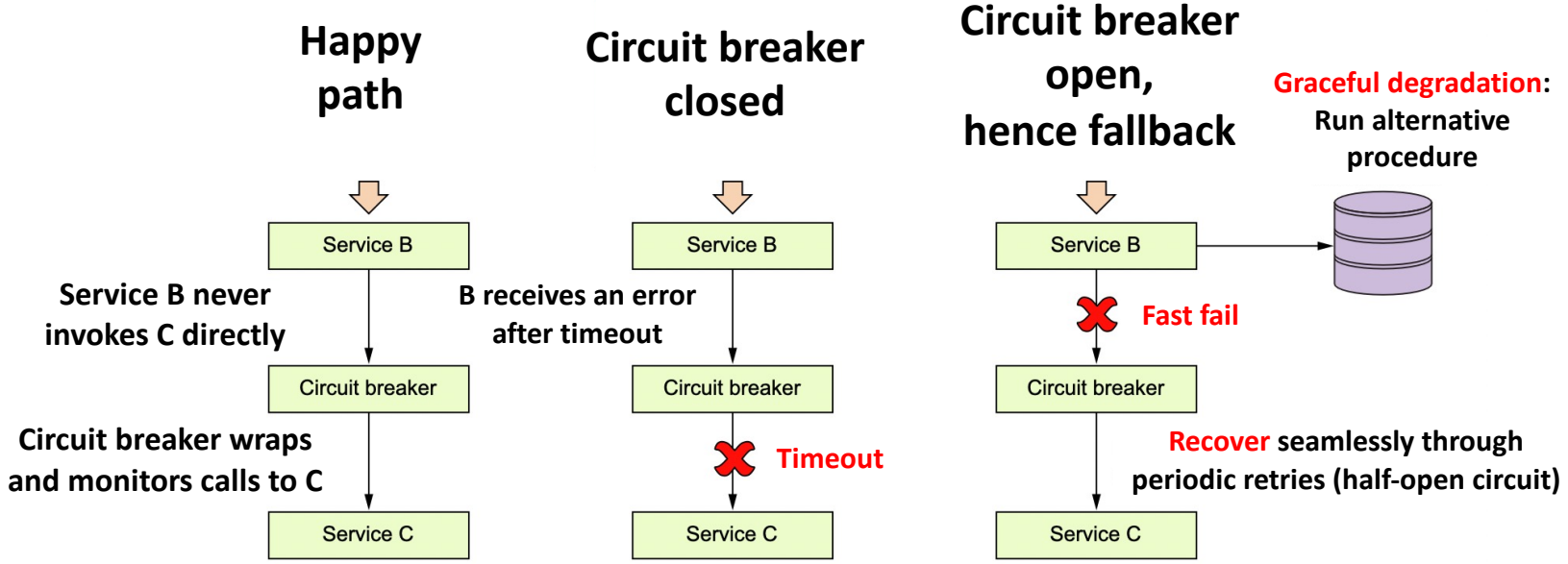
- Example — circuit breaker in action
- **How to avoid this problem?**
  - Insert a circuit breaker between B and C



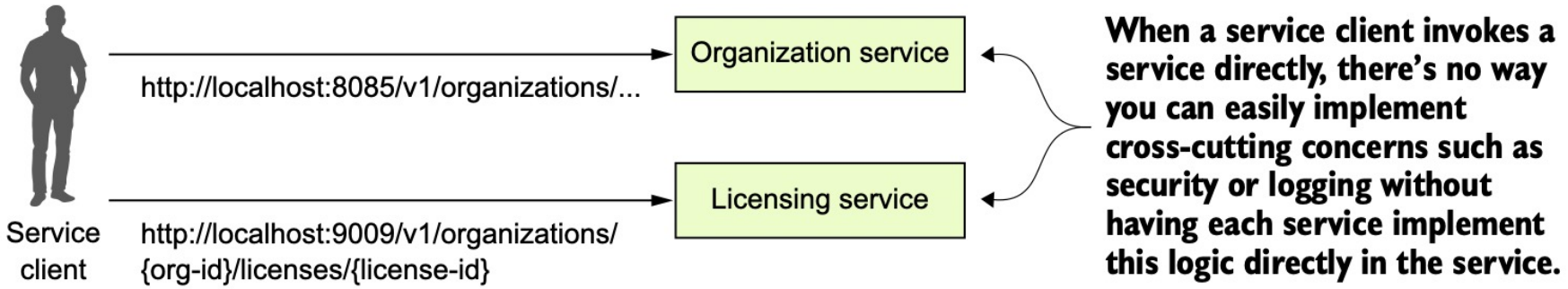
Microservices: resiliency patterns

- Example — circuit breaker in action
- Circuit breaker between microservice B and C avoids ripple effect for Applications





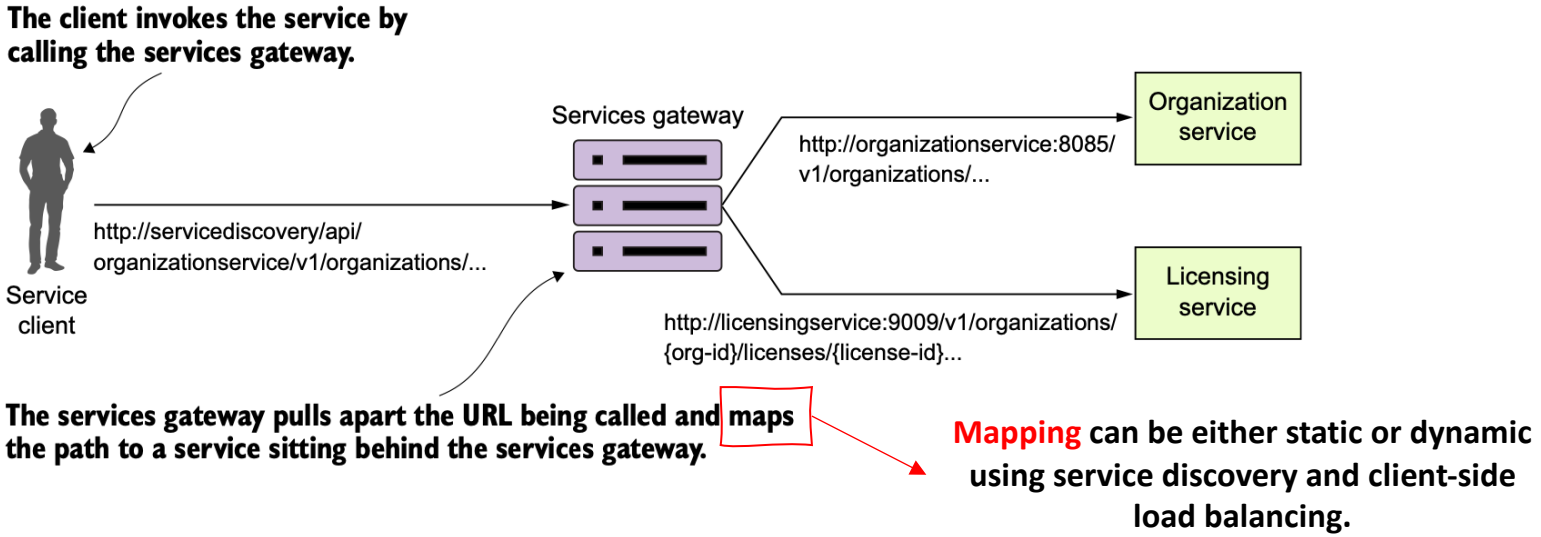
Microservices: security patterns



- **Problem**: what about **cross-cutting** concerns? (e.g., security, logging)
  - Service (or API) gateway

Microservices: security patterns

- Service (or API) gateway
  - Acts as a mediator. Sits between a service client and a service being invoked
  - Service clients talk only to the gateway
  - Gatekeeper for all traffic to microservices → can easily implement **authentication/authorization** mechanisms



Microservices: security patterns

- Service (or API) gateway — is it a single point of failure?
  - Yes, it can be.
  - **Solution**
    - A Service gateway has **multiple instances**
    - A **server-side load balancer** receives requests and spreads them across available instances

EXTRA MATERIAL

Microservices: security patterns

POLITECNICO MILANO 1863

- Authentication/authorization through OAuth2
  - Mainstream **token-based** security protocol (e.g., Facebook, GitHub, …)
  - Typically implemented or used by a service gateway

EXTRA MATERIAL

Microservices: security patterns

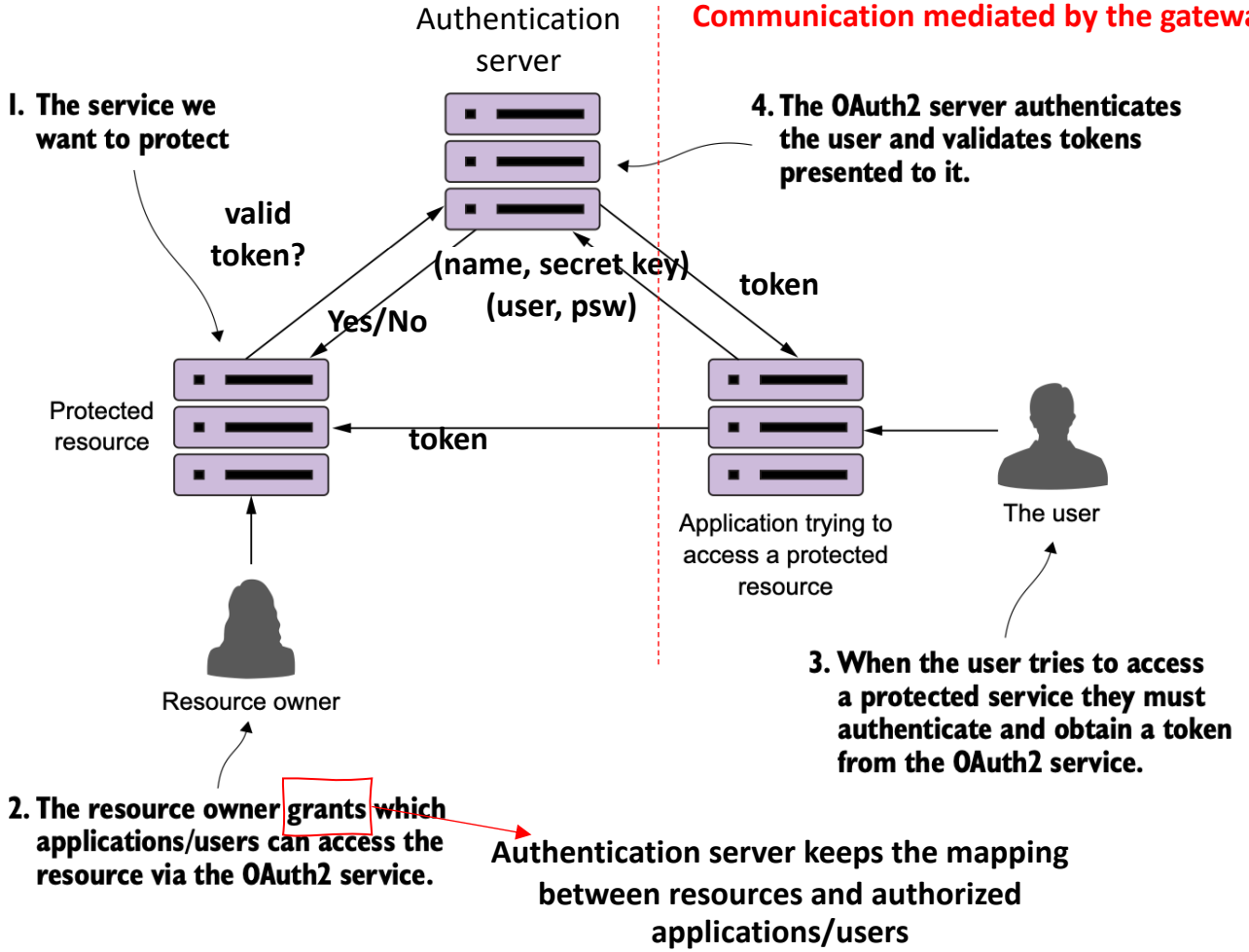
POLITECNICO MILANO 1863

- Components of OAuth2
  - **Protected resource**: a resource (in our case, a microservice) you want to protect and ensure that only authenticated users can access
  - **Resource owner**: A resource owner defines which applications are allowed to access the microservices. Each application registered by the resource owner has a (name, secret key) pair
  - **Application**: application that is going to call the service on a behalf of a user
  - **OAuth2 authentication server**: authentication server sits between the application and the services being consumed. The server allows the user to authenticate without having to pass their user credentials to every service the application is going to call

EXTRA MATERIAL

Microservices: security patterns

- OAuth2 mechanism
  - The four components interact together to authenticate the user
  - The user presents their credentials once to receive a token
  - Microservices can propagate the token in case of nested calls



Microservices: communication patterns

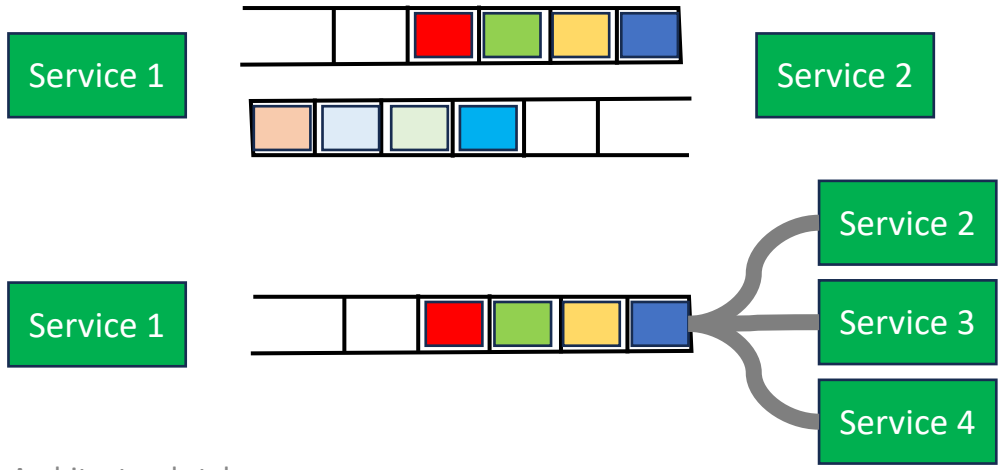
POLITECNICO MILANO 1863

- Synchronous vs asynchronous communication
  - Why this choice is important
    - **Synchronous** communication requires the two communicating parties to be ready to communicate at the same time → **tight runtime coupling**
    - **Asynchronous** communication allows each counterpart to enter in the communication at its own pace

Microservices: communication patterns

- Approach: using an event-driven framework to decouple the two parts

- Can support multiple communication styles
  - **Notification**: one way, see above
  - **Request/response**: two ways
  - **Publish/subscribe**: multicast



Microservices: communication patterns

POLITECNICO MILANO 1863

- Advantages
  - **Loose coupling, higher flexibility, scalability, and availability**
    - Counterparts do not know each other, they just send/receive messages
    - The set of services sending/receiving messages can change dynamically
    - If one of the two services is down, the other one can still work through the queue
    - If a receiver is busy, new replicas can be created to handle the messages stored in a queue
- Disadvantages
  - More complex to develop

Microservices: technologies

- **Spring boot**
  - De facto standard framework for developing microservices in Java
  - https://spring.io/guides/gs/spring-boot/
- **Spring Cloud Netflix**:
  - Integration of the following patterns: Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon)
  - https://cloud.spring.io/spring-cloud-netflix/reference/html/
- **Spring Cloud Stream**:
  - Framework for building event-driven microservices connected with shared messaging systems
  - Makes use of a variety of "binder implementations" (i.e., frameworks used to exchange messages), e.g., Apache Kafka
  - https://spring.io/projects/spring-cloud-stream

References

- Carnell, John, and Illary Huaylupo Sánchez. Spring microservices in action. Simon and Schuster, 2021
- Spring Boot Reference Documentation: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/