Software Engineering 2

Microservices

Routing, resilience, security, communication patterns
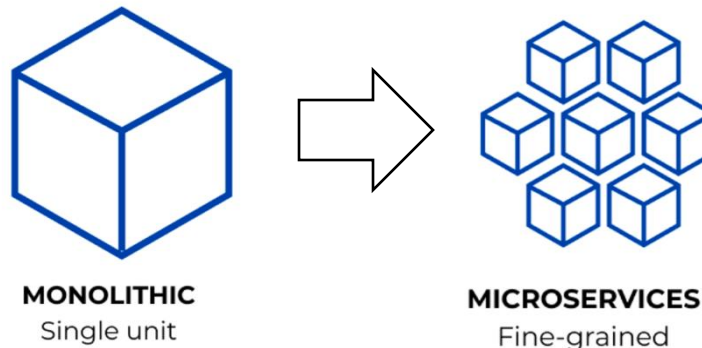
M Camilli, E Di Nitto, M Rossi

Architectural Styles

Microservices

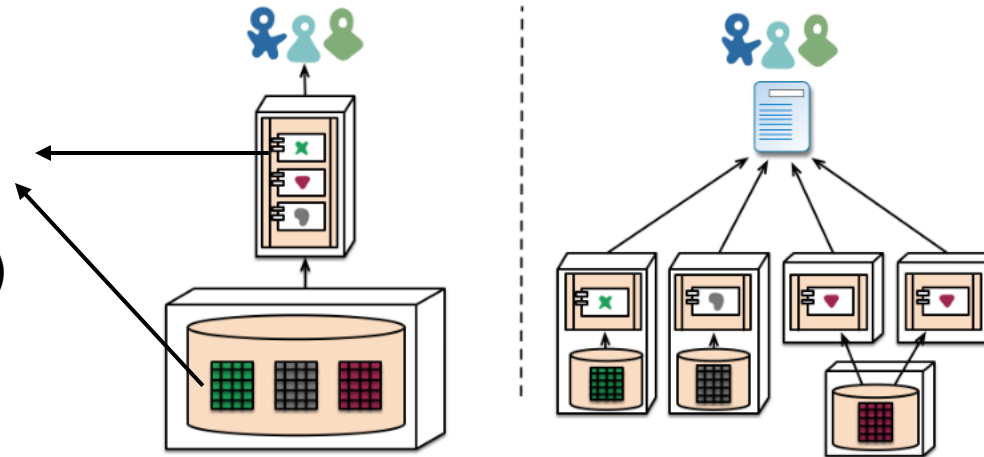# Microservice architectural style

- **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

MONOLITHIC
Single unit

MICROSERVICES
Fine-grained

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
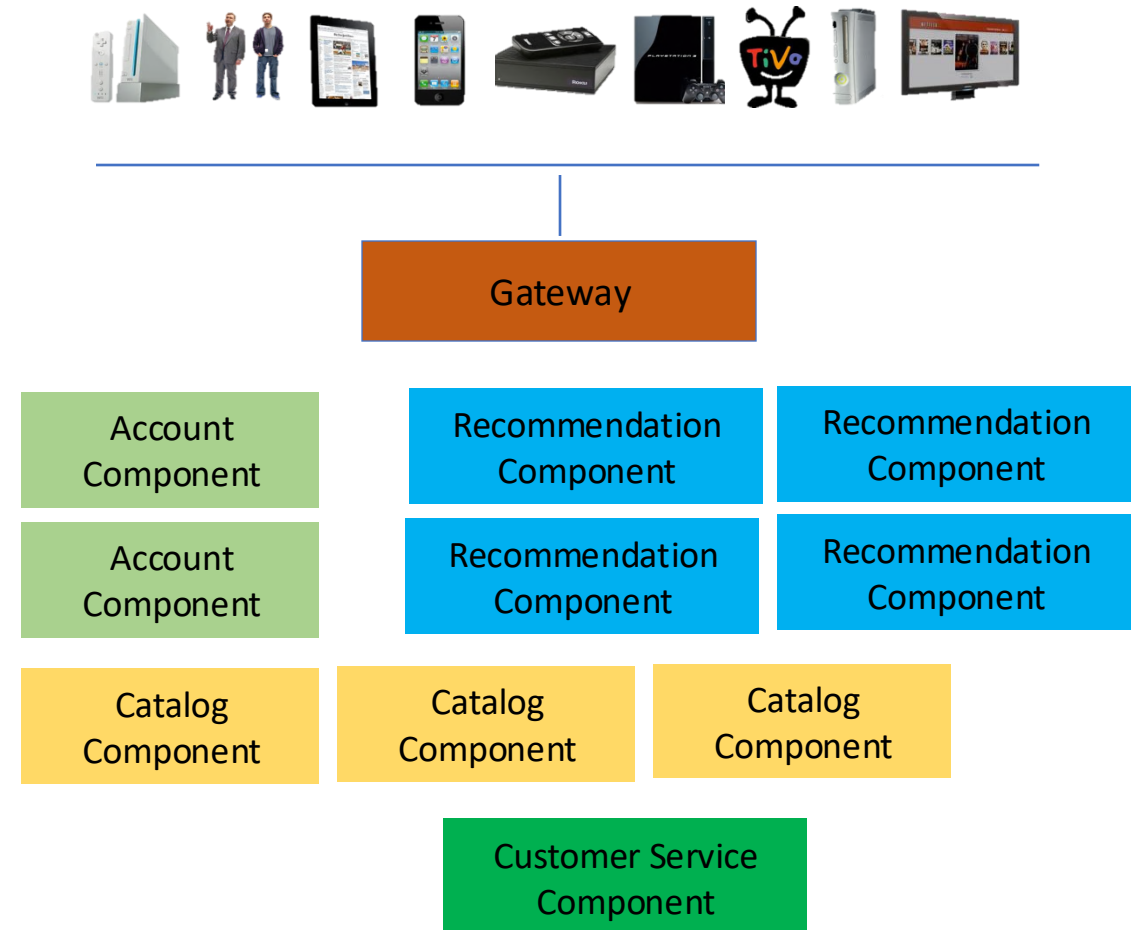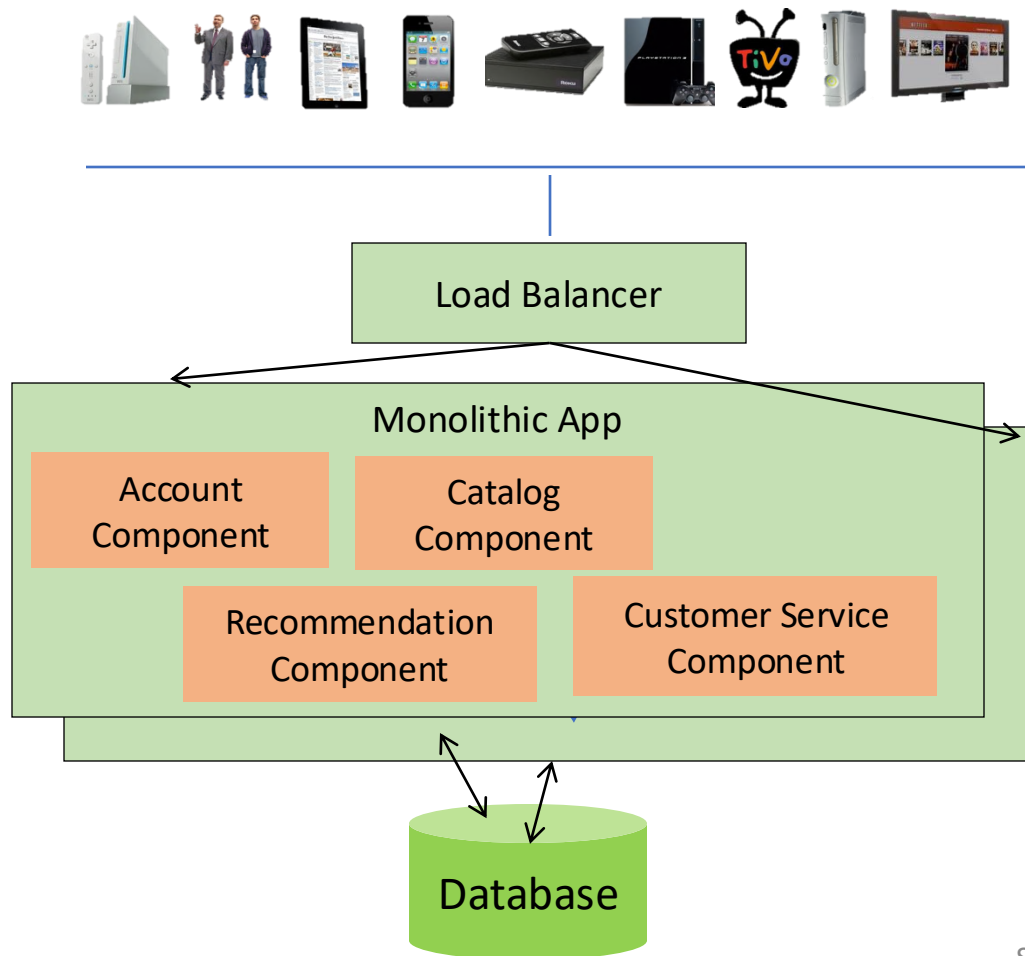(single area of responsibility)

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

Microservices: advantages (product)

- 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)

- 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)

Each team has their own areas of responsibility with their own requirements and delivery demands

All their work is synchronized into a single code base
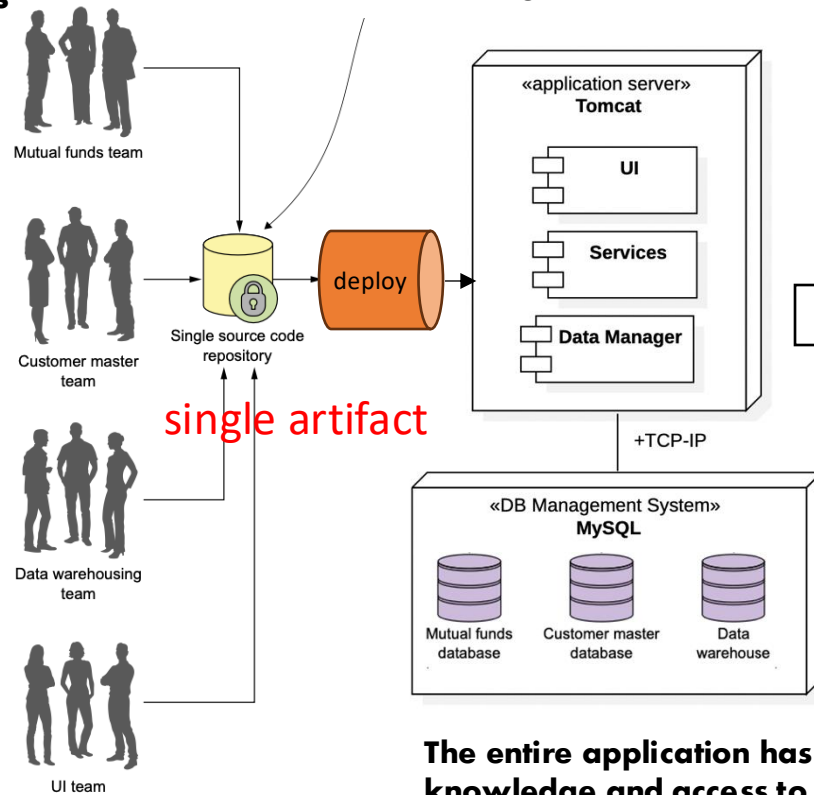
single artifact

Multiple teams

The entire application has knowledge and access to all the data sources used within the application

Multiple teams
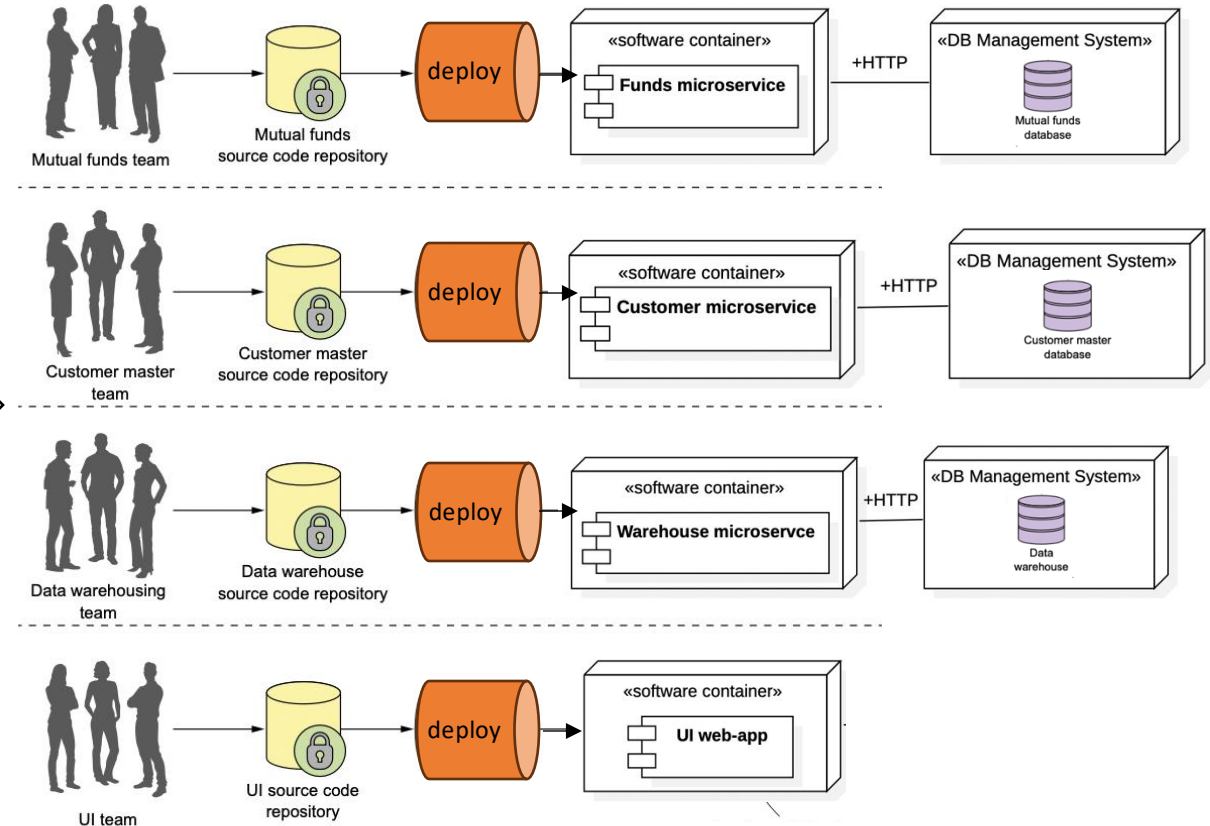
multiple artifacts

Invokes all business logic as REST-based service calls

Microservices: advantages (process)

- 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

Public API

POST /restaurants
GET /restaurants

Data Store
(eg, RDS, DynamoDB
ElastiCache, ElasticSearch)

Application/Logic
(code, libraries, etc)
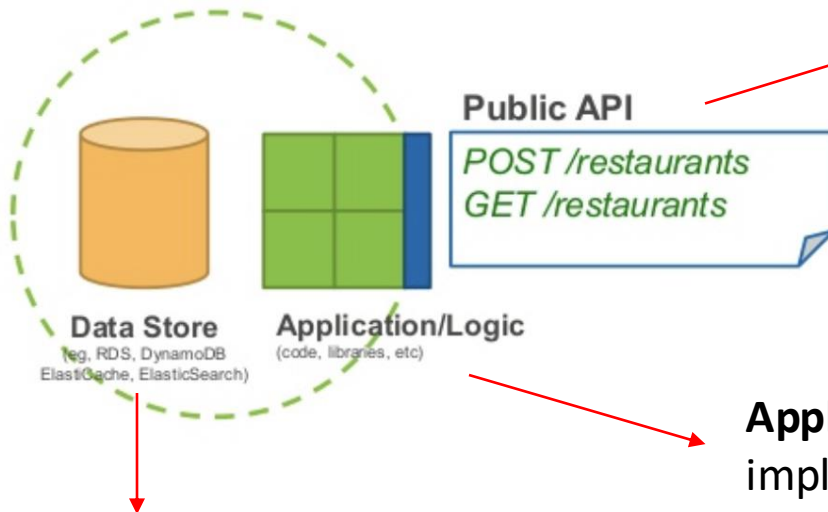
REST API

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

Hello microservice

verb URL resource parameters

GET `http://localhost:8080/hello/john/carnell`

HTTP STATUS:200
{"message": "Hello john carnell"}

① A client makes a HTTP GET request to the Hello microservice

⑦ The client receives the response from the microservice as JSON. The success/failure of the call is returned as HTTP status code

② **Route mapping** — The microservice will parse the HTTP request and map verb, URL, resource, and parameters to a method in the codebase

③ Once the microservice has identified the route it will map any parameters defined inside the route to arguments of the corresponding method — **Parameter destructuring**

④ **JSON -> object mapping** — Additional parameters can be passed through JSON in the HTTP body. These map to objects used by the method.

⑤ Once the data has been mapped, the microservice will execute the business logic. The logic may call operations exposed by other microservices. — **Business logic execution**

⑥ **object -> JSON mapping** — Once the business logic is executed, the microservice will convert returned objects to JSON.

Microservices: besides business logic

routing patterns

Communication patterns

How do I choose between synchronous and asynchronous modes?

Loosely coupled

How do you manage the physical location so services instances can be added and removed without impacting service clients?

Location transparent

Secure

How do I determine whether the service client calling a microservice is allowed to undertake certain actions?

Security patterns

How do you make sure the service is focused on one area of responsibility?

Right-sized

Your microservice

Resilient

How do you make sure when there is a problem with a service, service clients "fail fast"?

Resiliency patterns

Routing patterns

How do you ensure that your applications can scale quickly with minimal dependencies between services?

Scalable

Repeatable

How do you ensure that every time a new service instance is started it always has the same code and configuration as existing instance(s)?
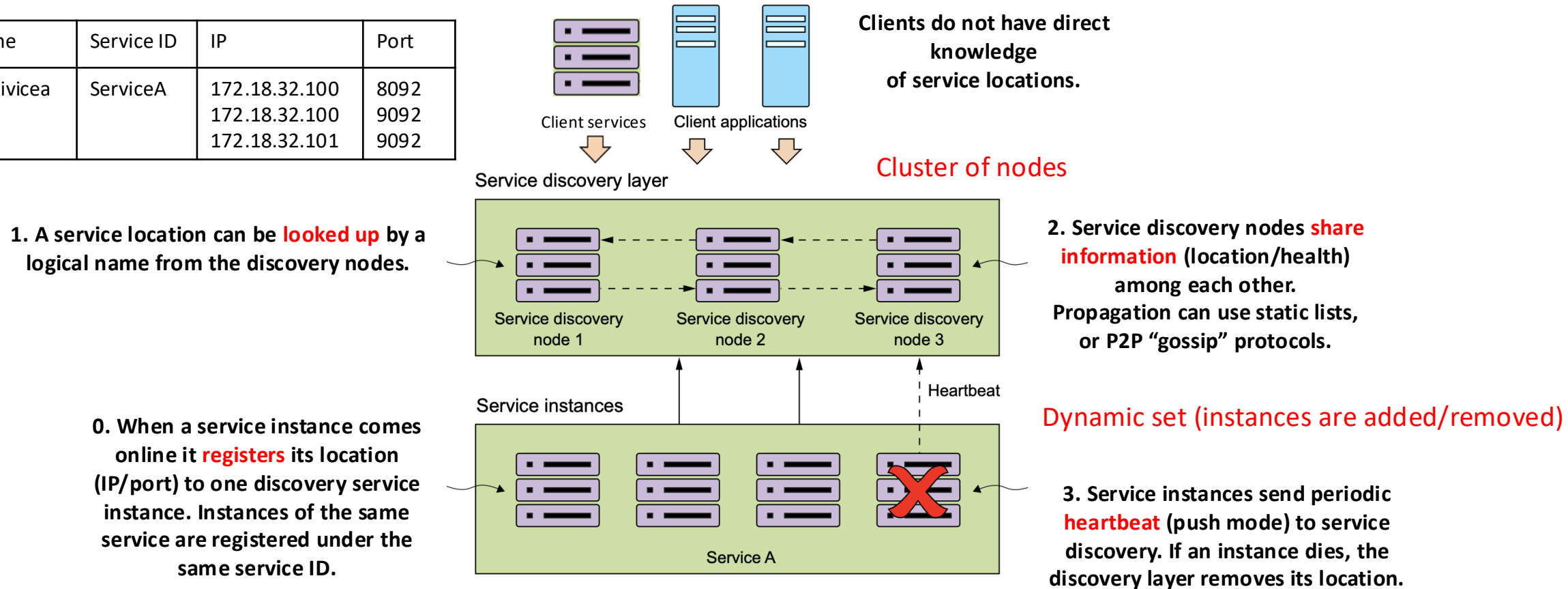
Microservices: routing patterns

- **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.

Client services   Client applications

Service discovery layer

Cluster of nodes

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

Service discovery node 1   Service discovery node 2   Service discovery node 3

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

Service instances

Heartbeat

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.**

Service A

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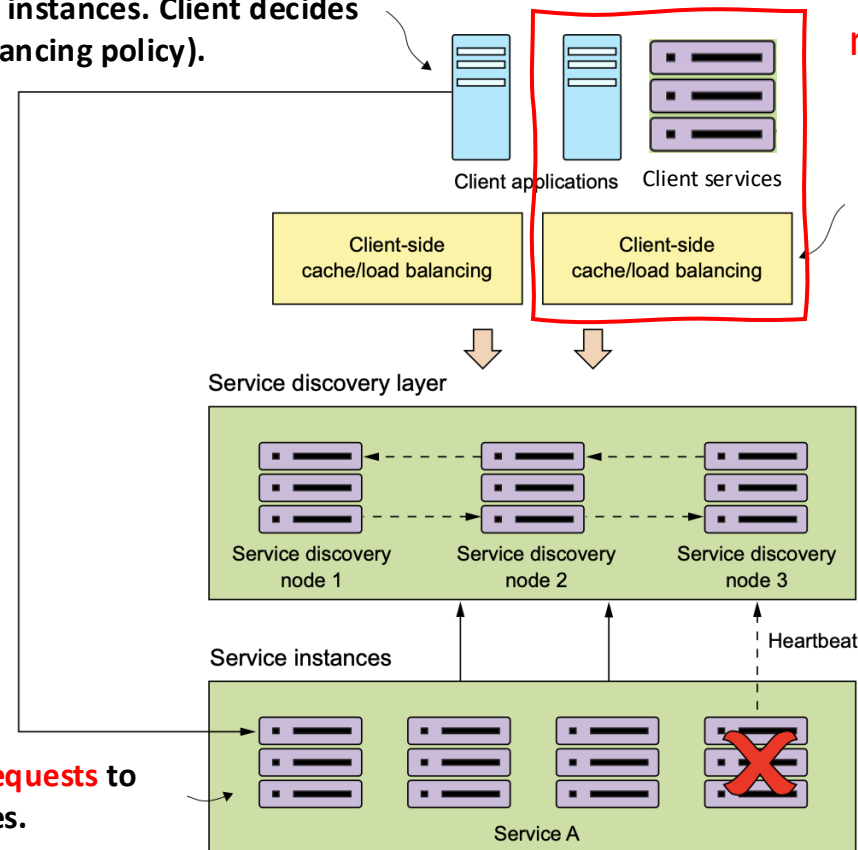
Microservices: routing patterns

- ## 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."

client machine

Client applications   Client services

Client-side cache/load balancing

Client-side cache/load balancing

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

Service discovery layer

Service discovery node 1   Service discovery node 2   Service discovery node 3

Service instances

Heartbeat

2. Client sends direct requests to service instances.

Service A

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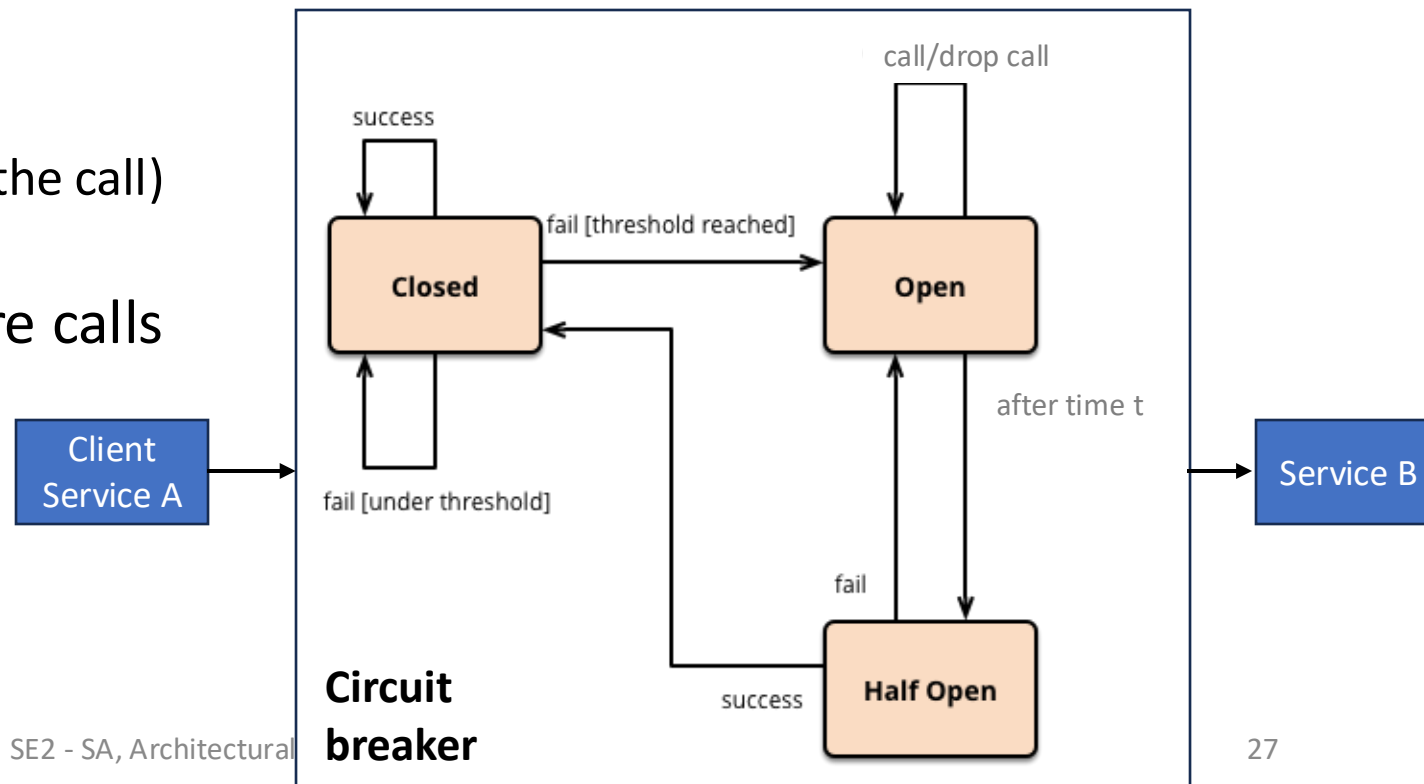
- Service discovery architecture with cache — possible issues
  - Does not ensure **consistency** of cached data $\rightarrow$
    - When a client contacts a dead instance, the client invalidates the local cache and forces a refresh by contacting the service discovery engine.

Microservices: resiliency patterns

- "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"
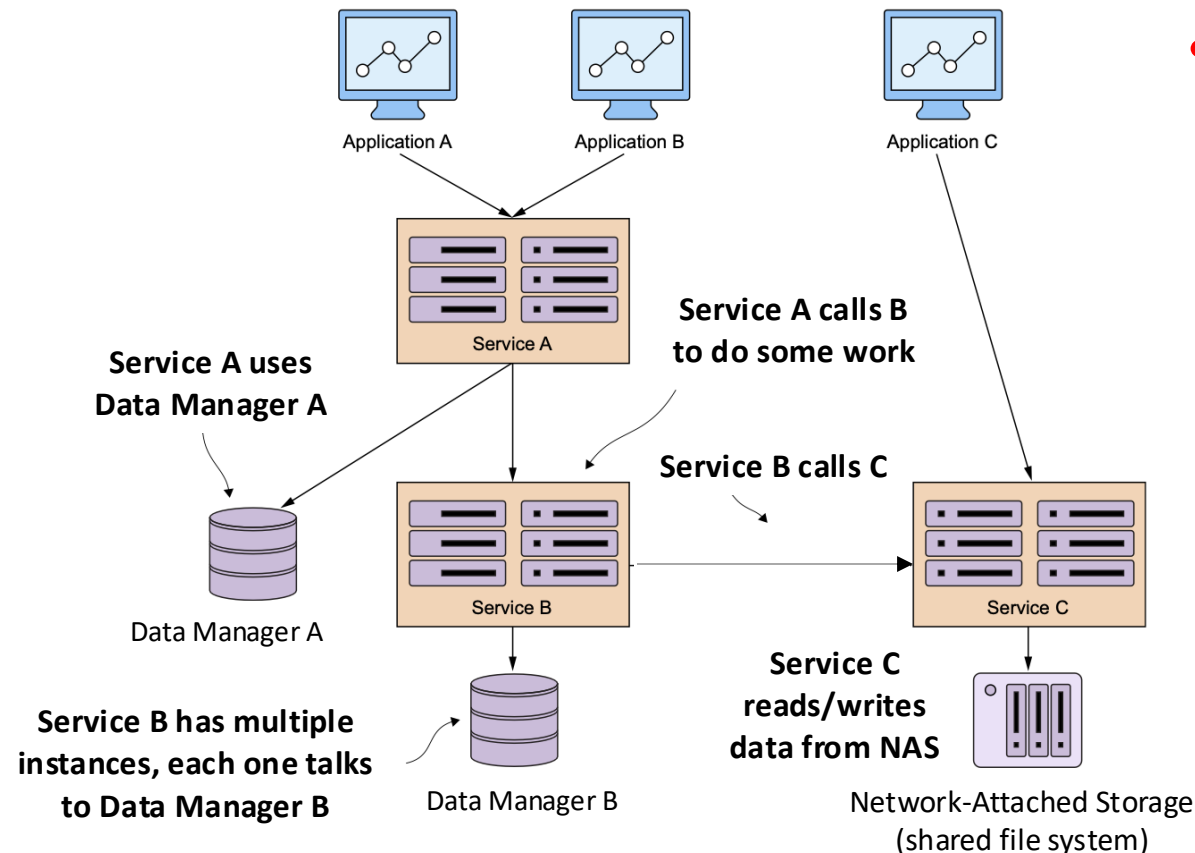 - Avoid useless resource consumption
  - Avoid ripple effects

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

Application A   Application B   Application C

Service A uses
Data Manager A

Data Manager A

Service B has multiple
instances, each one talks
to Data Manager B

Service A

Service A calls B
to do some work

Service B calls C

Service B

Data Manager B

Service C

Service C
reads/writes
data from NAS

Network-Attached Storage
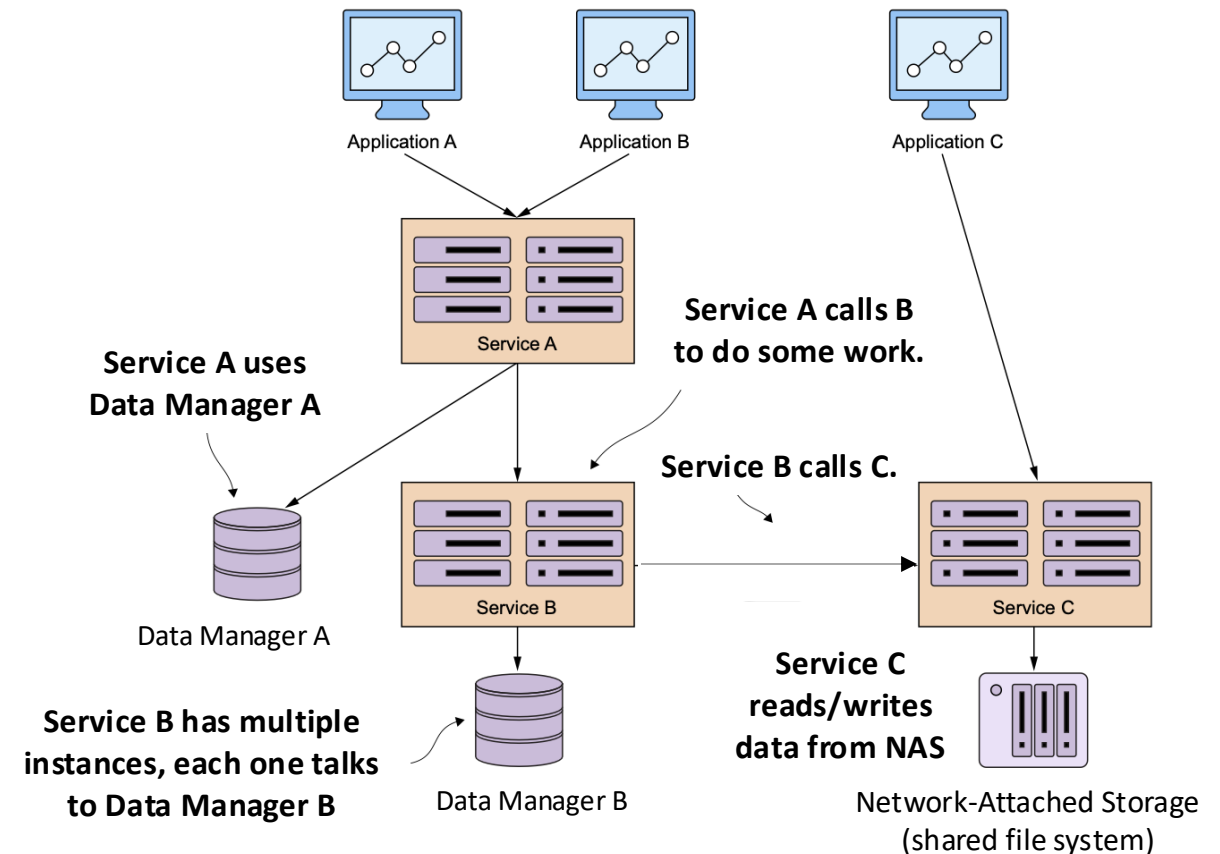(shared file system)

- 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

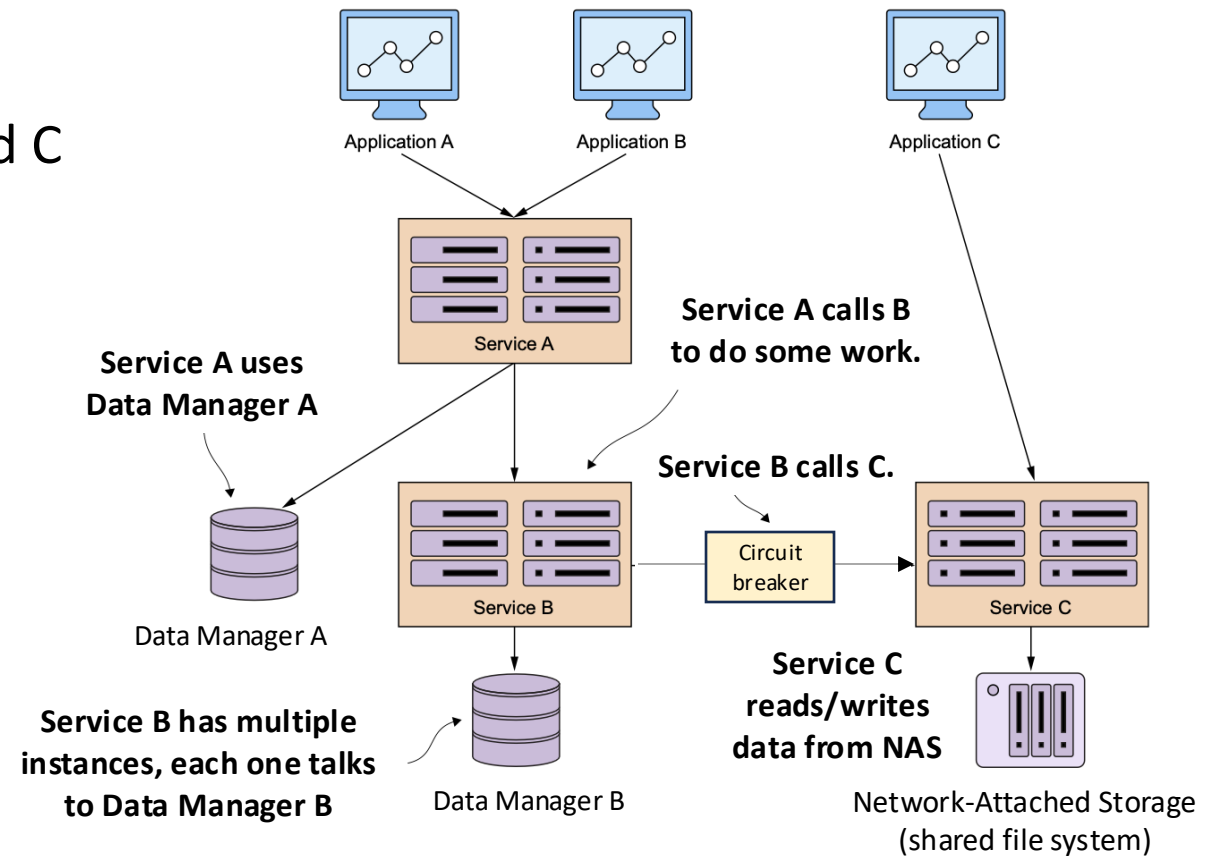- Example — circuit breaker in action
- **What happens? (ripple effect)**
  - C starts running slowly
  - Pending requests to C grow
  - Number of concurrent DB connections grow
  - Resources become exhausted because Service C never completes
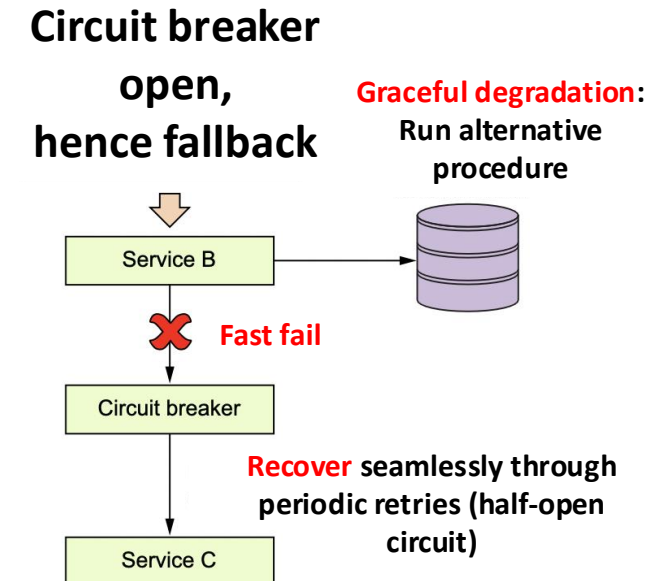 - Service A starts running out of resources because it calls B that is slow because of C

Application A   Application B   Application C

Service A uses Data Manager A

Service A calls B to do some work.

Service A

Service B calls C.

Data Manager A

Service B

Service C

Service B has multiple instances, each one talks to Data Manager B

Service C reads/writes data from NAS

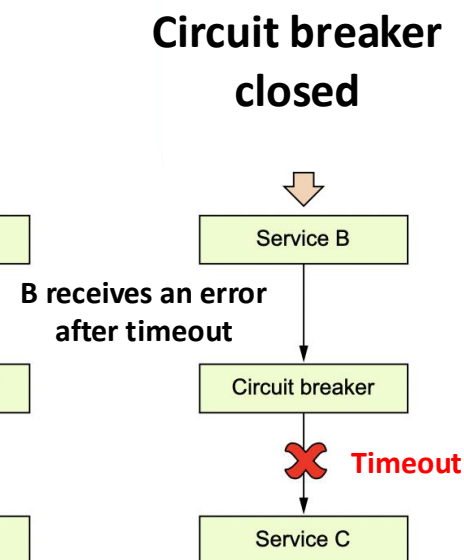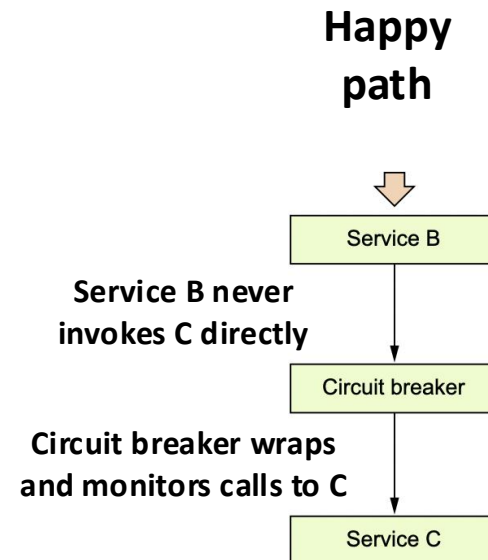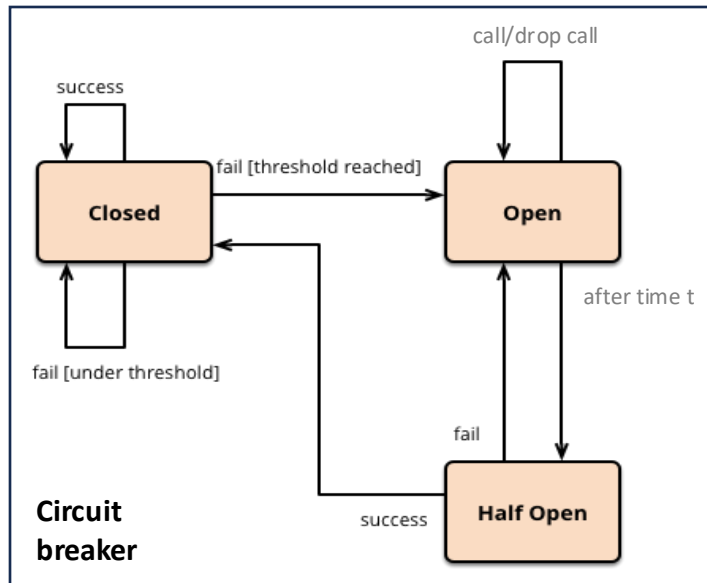Data Manager B

Network-Attached Storage (shared file system)

Microservices: resiliency patterns

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

Application A   Application B   Application C

Service A uses Data Manager A

Service A calls B to do some work.

Service A

Data Manager A

Service B calls C.

Circuit breaker

Service B

Service C

Service B has multiple instances, each one talks to Data Manager B

Data Manager B

Service C reads/writes data from NAS
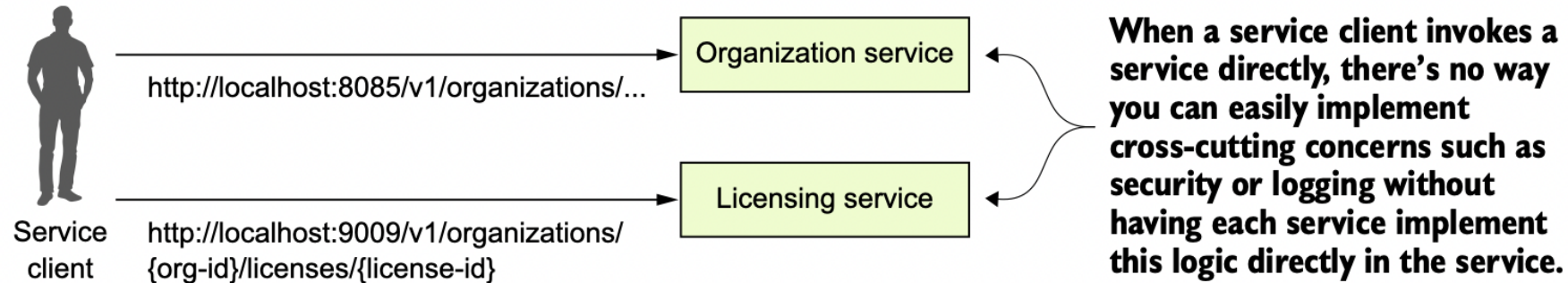
Network-Attached Storage (shared file system)

Microservices: resiliency patterns

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

Circuit breaker

Happy path

Service B never invokes C directly

Circuit breaker wraps and monitors calls to C

Circuit breaker closed

B receives an error after timeout

Timeout

Circuit breaker open, hence fallback

Fast fail

Graceful degradation: Run alternative procedure

Recover seamlessly through periodic retries (half-open circuit)

Microservices: security patterns

When a service client invokes a service directly, there's no way you can easily implement cross-cutting concerns such as security or logging without having each service implement this logic directly in the service.
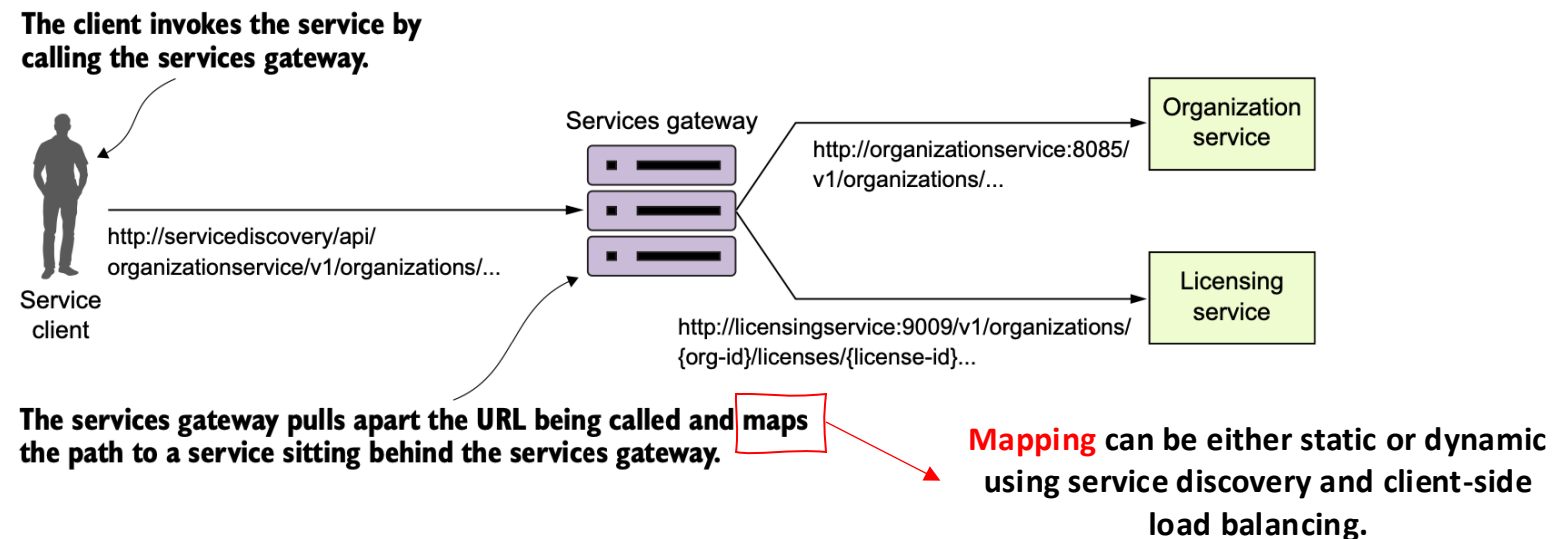
- **Problem**: what about **cross-cutting** concerns? (e.g., security, logging)
  - Service (or API) gateway

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

The client invokes the service by calling the services gateway.

http://servicediscovery/api/
organizationservice/v1/organizations/...

Service client

Services gateway

http://organizationservice:8085/
v1/organizations/...

Organization service

http://licensingservice:9009/v1/organizations/
{org-id}/licenses/{license-id}...

Licensing service

The services gateway pulls apart the URL being called and maps the path to a service sitting behind the services gateway.

Mapping can be either static or dynamic using service discovery and client-side load balancing.

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

Microservices: security patterns

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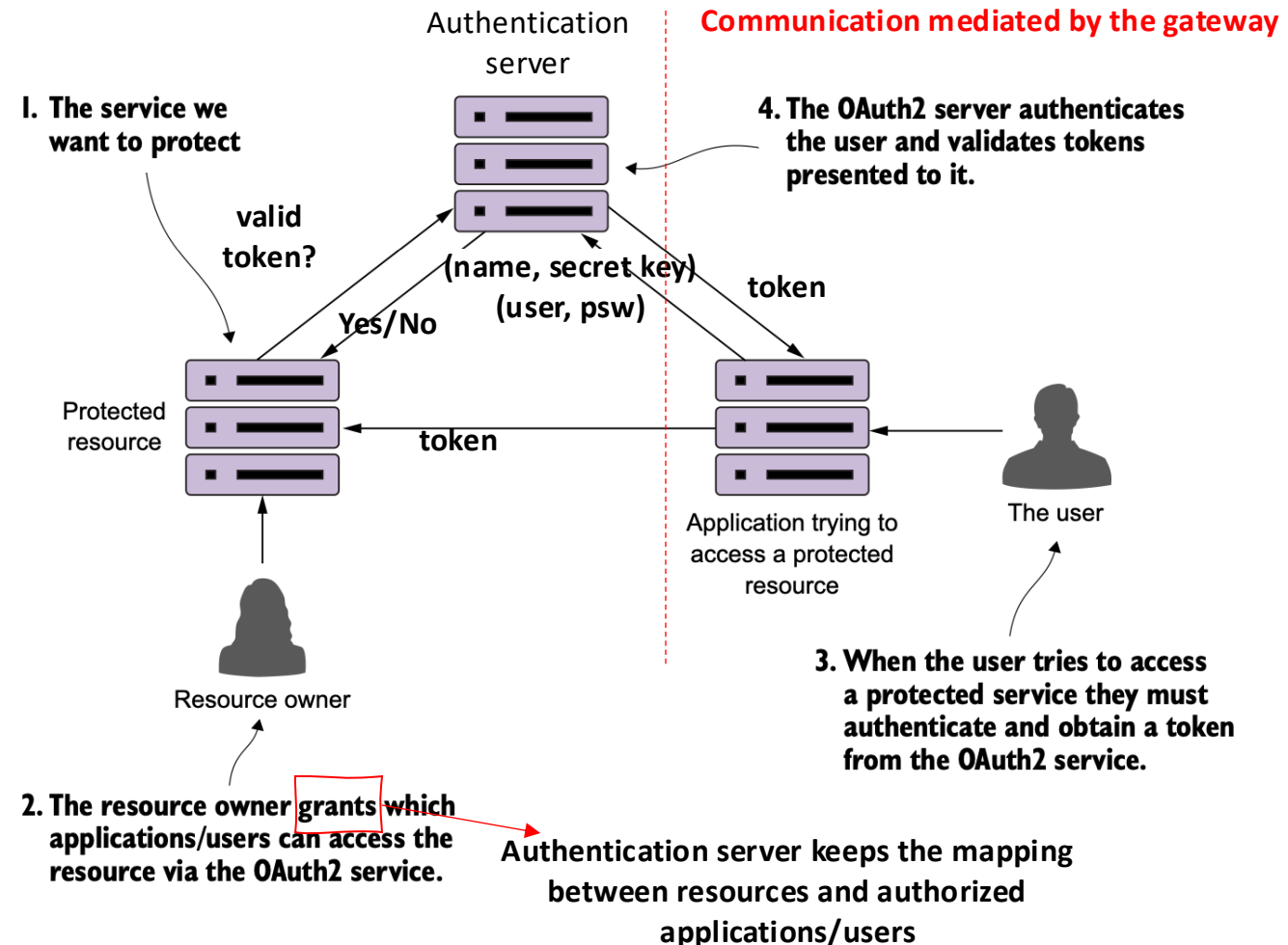
Microservices: security patterns

- ## 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

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

Authentication server

Communication mediated by the gateway

I. **The service we want to protect**

valid token?

4. **The OAuth2 server authenticates the user and validates tokens presented to it.**

(name, secret key)
(user, psw)

token

Yes/No

Protected resource

token

Application trying to access a protected resource

The user

3. **When the user tries to access a protected service they must authenticate and obtain a token from the OAuth2 service.**

Resource owner

2. **The resource owner grants which applications/users can access the resource via the OAuth2 service.**

Authentication server keeps the mapping between resources and authorized applications/users

Microservices: communication patterns

- 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
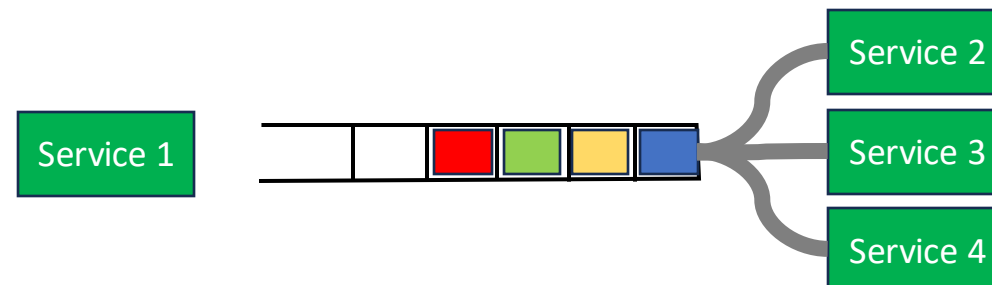

- Can support multiple communication styles
  - **Notification**: one way, see above

  - **Request/response**: two ways

  - **Publish/subscribe**: multicast

Microservices: communication patterns

- ## 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/