Guest Lecture: Microservice Patterns

January 30 2023

Raoul (he/him)

- Staff Software Engineer @HubSpot
- When I’m not in front of my computer, I love to be in the nature 🏔️⛵🏄
- Fun fact: My dream job as a kid was to be an excavator operator 👷

About Us

HubSpot’s CRM platform has all the tools and integrations you need for marketing, sales, content management, operations, and customer service.

Each product in the platform is powerful alone, but they’re even better together.

Since 2006, HubSpot has helped over 128,000 companies in over 120 countries grow better.

To achieve that we have about 1,500 engineers globally.

HubSpot embraces a microservice architecture

19k

Deployables

2Mil

Requests / s

16Mil

Messages / s

At HubSpot, we are building **two** products:

One for our
customers.

One for our
employes.

We've created a company culture that embodies...

- Autonomy
- Trust
- Transparency
- Empathy
- Diversity Inclusion & Belonging

glassdoor

BEST PLACES TO WORK 2022

HubSpot
Best Career Growth 2022

HubSpot
Best Leadership Teams 2022

HubSpot
Best CEOs for Diversity 2022

HubSpot
Best CEOs for Women 2022

HubSpot
Best Operations Team 2022

HubSpot
Best HR Team 2022

HubSpot
Best Sales Team 2022

HubSpot
Best Marketing Team 2022

HubSpot
Best Product & Design Team 2022

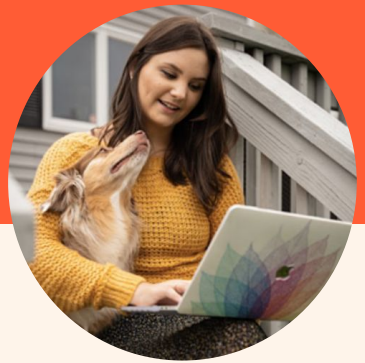
HubSpot
Best Engineering Team 2022

HubSpot
Best Places to Work in Boston 2022

HubSpot
Best Global Culture 2022

HubSpot
Best Company Outlook 2022

@flex



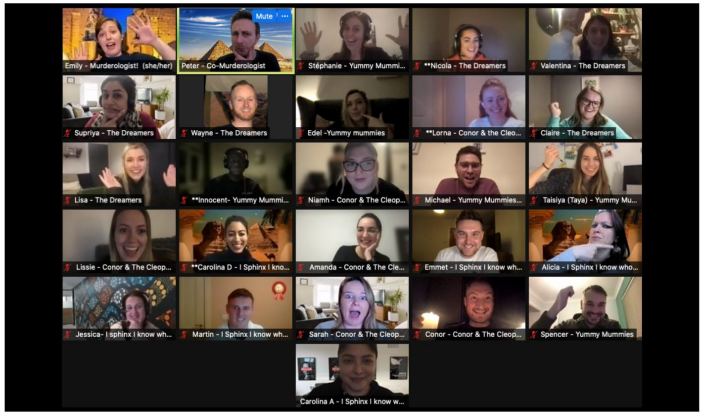
@home

@office

Work anywhere with HubSpot

In-person events

Online events



teambuilding™

How to stay in touch with us

- **Sign up** for the upcoming 'Coding Assessment' Workshop organized by HubSpot by submitting the interest form. We will reach out to you soon to provide further details and schedule.
- Entry-level roles and internships we offer you can find via this link
- LinkedIn and TikTok under **@Studentspot**
- Instagram and YouTube under **@hubspotlife**
- Connect with us on LinkedIn: Fahad Ani, Raoul Friedrich
- Or each out via email: fani@hubspot.com, rfriedrich@hubspot.com

Agenda

#0 Introduction to Microservices

#1 Strangler Pattern

#2 Decentralized ID Generation

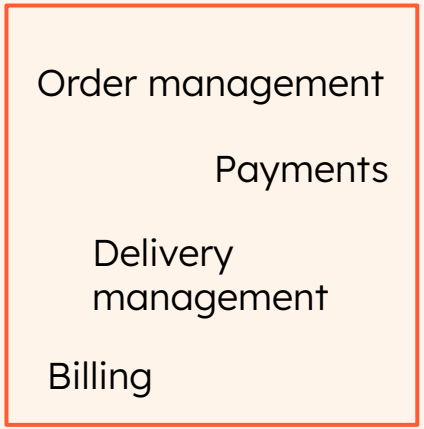
#3 Circuit Breaker

#4 Asynchronous Messaging

What are microservices?

The microservice architecture, is an architectural style that functionally decomposes an application into a set of services.

Monolithic architecture



Microservice architecture

[1] https://martinfowler.com/articles/microservices.html

Benefits of microservices

- Services are small and easily maintained
- Services are independently deployable
- Services are independently scalable
- They enable the continuous delivery and deployment of large, complex applications
- They enable teams to be autonomous
- Allows easy experimenting and adoption of new technologies
- Better fault isolation

Challenges of microservices

- Deciding when to adopt the microservice architecture is difficult
- Finding the right set of services is challenging
- Distributed systems are complex, which makes development, testing and deployment difficult
- Deploying features that span multiple services requires careful coordination

Agenda

#0 Introduction to Microservices

#1 Strangler Pattern

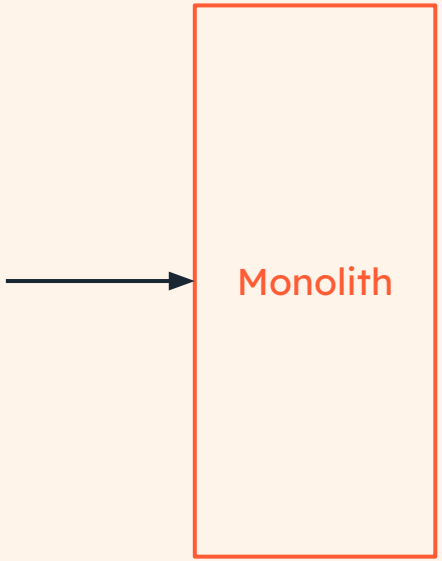
#2 Decentralized ID Generation

#3 Circuit Breaker

#4 Asynchronous Messaging

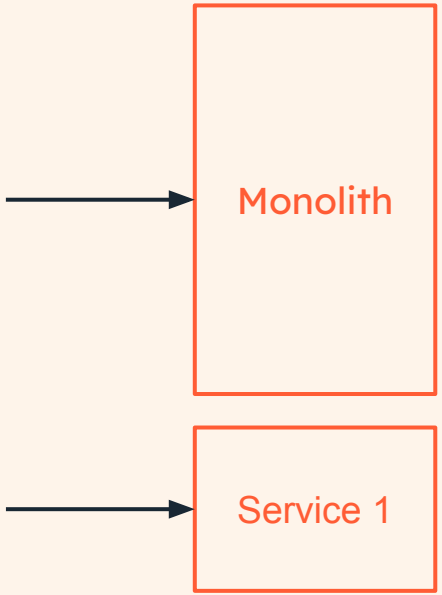
Strangler Pattern

Problem



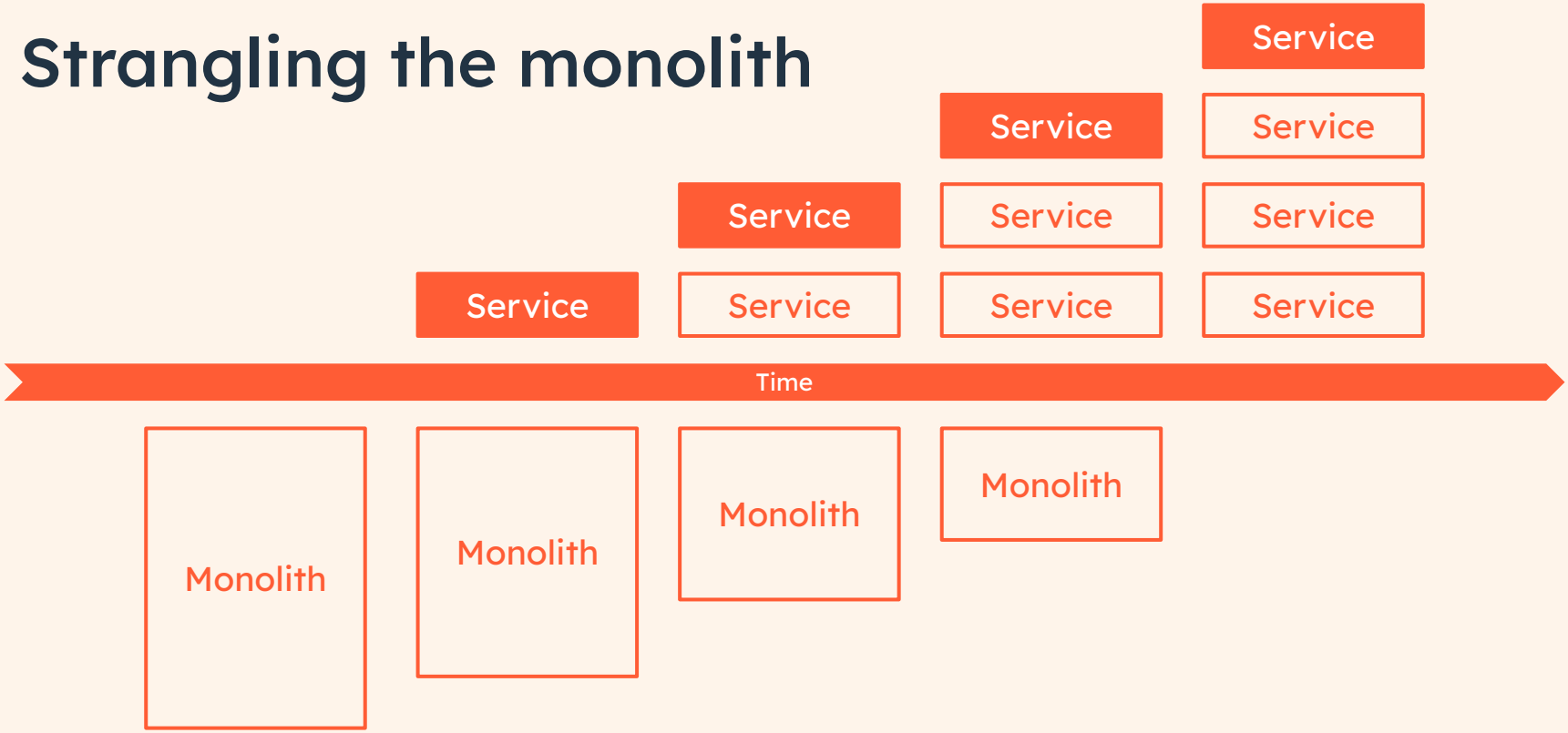
- Your application is growing (more traffic, more developers, ...)
- Your code-base grows and becomes complex
- Developers are stepping on each others' toes
- You once were in a lecture about microservices and you remember this could be a solution
- So we want to transfer to microservices, without stopping operations or development

Solution



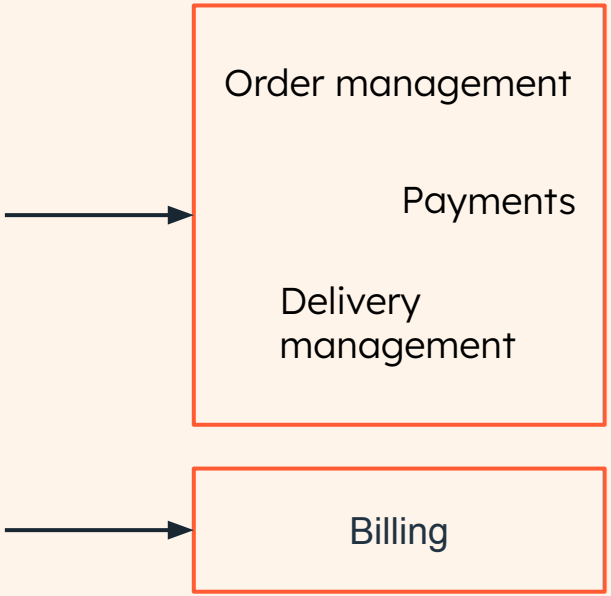
- We break out small flows of the monolith into a separate and new service
- Over time, there are more and more services and the monolith will shrink until it’s obsolete

Strangling the monolith



[2] https://microservices.io/patterns/refactoring/strangler-application.html

Example: Food Delivery



A long time ago, in a galaxy far,
far away....

Episode I

At the moment, Master Yoda is responsible for all important tasks within the Jedi Order. He finds new apprentice candidates, trains them, consults the rebellion, and much more.

The rebellion is concerned this is not scalable for the aging Yoda and also risky in case of an attack.

Help and make the Jedi Order more scalable and resilient by moving some responsibilities from Yoda to Obi-Wan Kenobi.

Agenda

#0 Introduction to Microservices

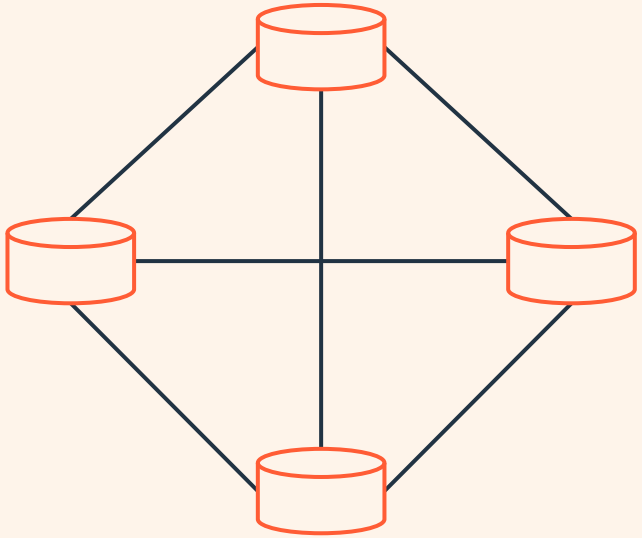
#1 Strangler Pattern

#2 Decentralized ID Generation

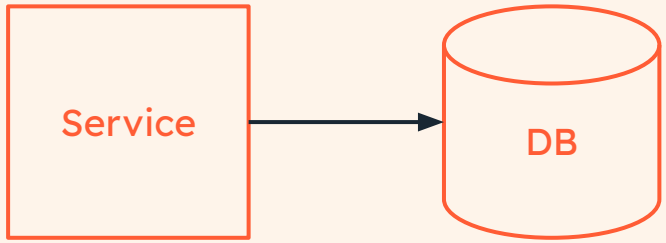
#3 Circuit Breaker

#4 Asynchronous Messaging

Decentralized ID Generation

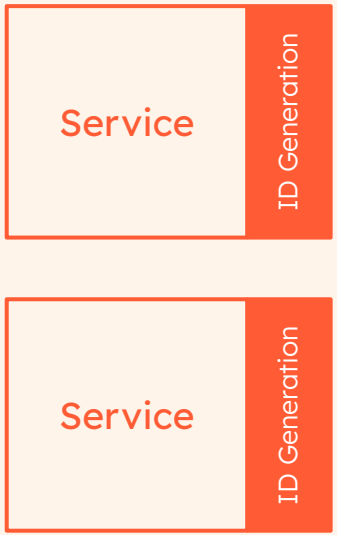


Problem



- ID generation is handled by our database's *auto_increment*
- We outgrew our single DB and now we need to split it up
- We still want to have unique IDs

Solution 1 - UUID



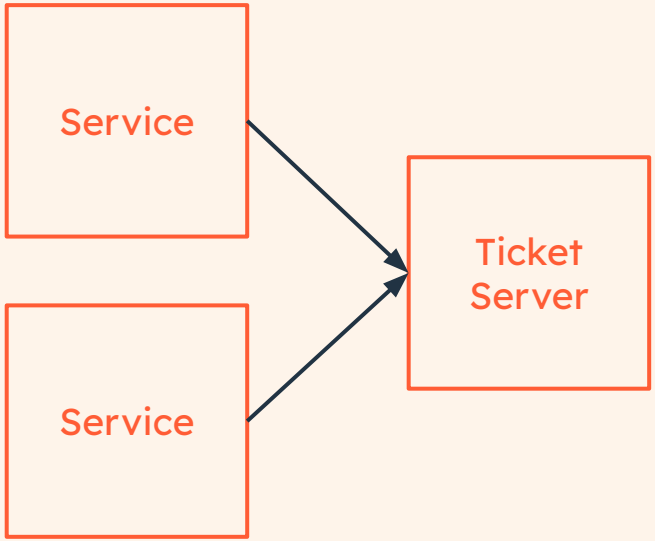
Pros:

- Generating UUID is simple. No coordination between servers is needed
- Fast

Cons:

- IDs are 128 bits long
- IDs do not go up with time
- IDs could be non-numeric

Solution 2 - Ticket Server



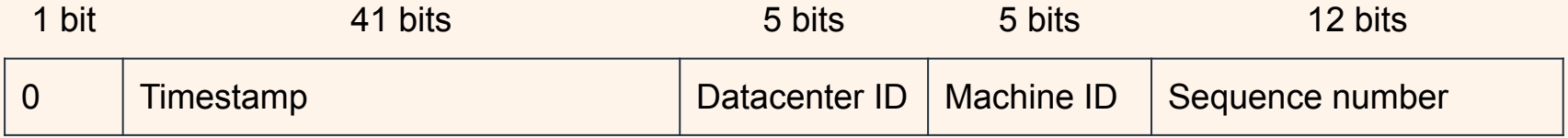
Pros:

- Numeric IDs
- Easy to implement
- Works for small to medium-scale applications

Cons:

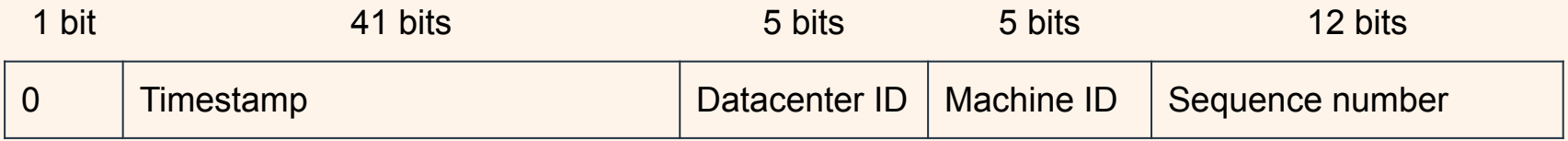
- Single point of failure. To avoid that you can setup multiple ticket servers, but that requires synchronization

Solution 3 - Twitter Snowflake



- Each ticket server (see Solution 2) can generate IDs independently
- Take the lowest 41 bits of a timestamp.
  Note: Timestamp may also start with a different epoch
- On startup, a ticket server receives a datacenter and machine ID
- Sequence number increments with every ID. It is reset every millisecond

Solution 3 - Twitter Snowflake



Pros:

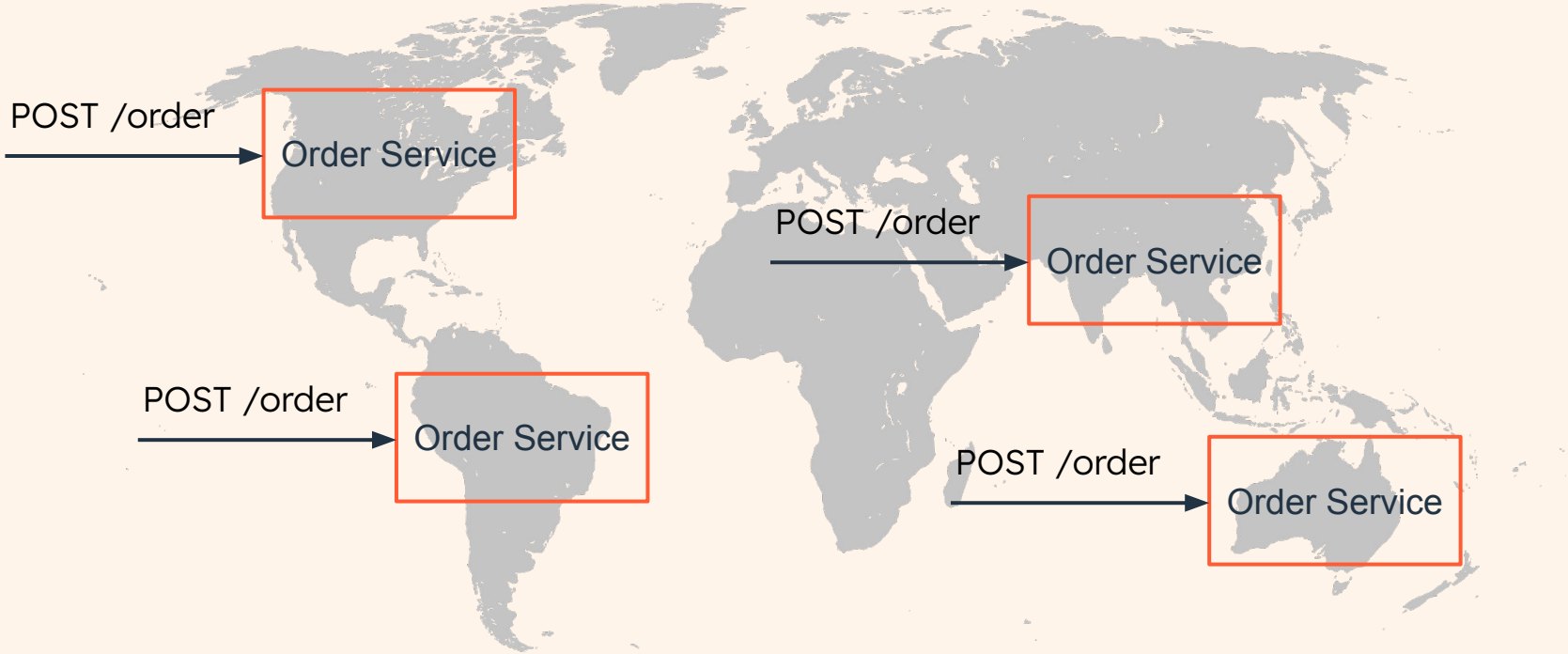
- Numeric IDs
- No single point of failure
- No synchronization

Cons:

- Harder to setup
- Timestamp only has 41 bits

[3] Alex Xu, System Design Interview

Example: Food Delivery



Episode II

The rebellion is building secret production facilities across the galaxy to produce droids. Voice commands for droids are based on their ID. This means they must be unique and as short as possible.

The rebellion is afraid that the Empire will intercept communication and detect our facilities. Therefore, we must generate IDs offline. For that purpose a production facility gets a 4 digit unique facility ID when it's built.

Implement the ID generation mechanism for our factories.

Agenda

#0 Introduction to Microservices

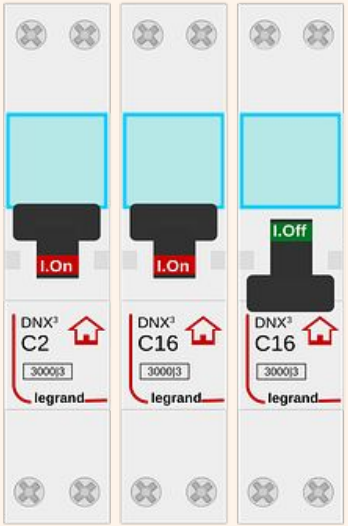
#1 Strangler Pattern

#2 Decentralized ID Generation

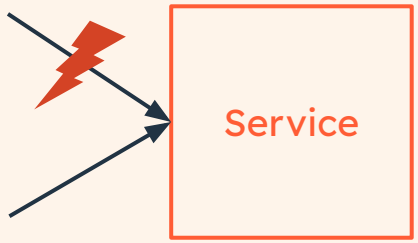
#3 Circuit Breaker

#4 Asynchronous Messaging

Circuit Breaker

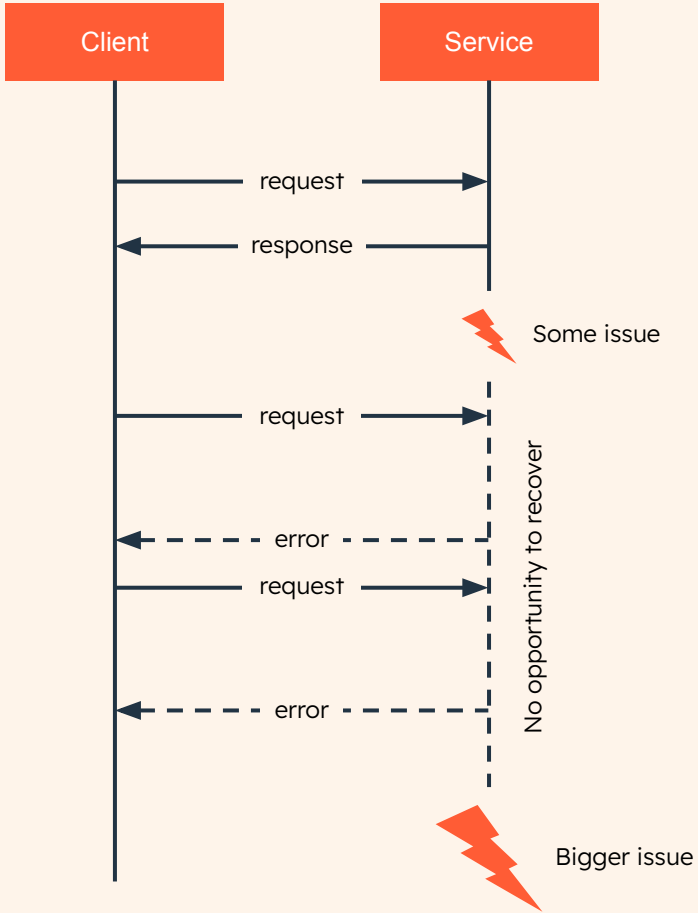


Problem

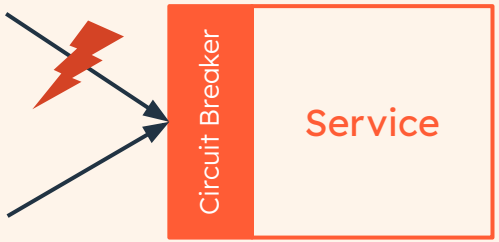


- Our service returns an error for 1 endpoint (e.g. we introduced a bug)
- There is a danger that consumers of our service drain our resources due to retries
- Other endpoints work just fine and they shouldn't get affected or crash

Circuit Breaker

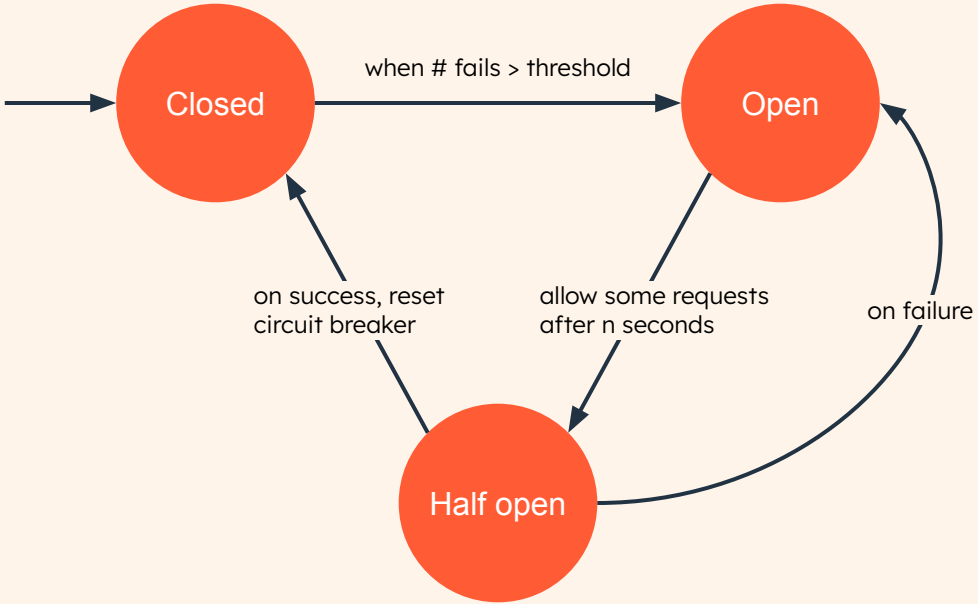


Solution

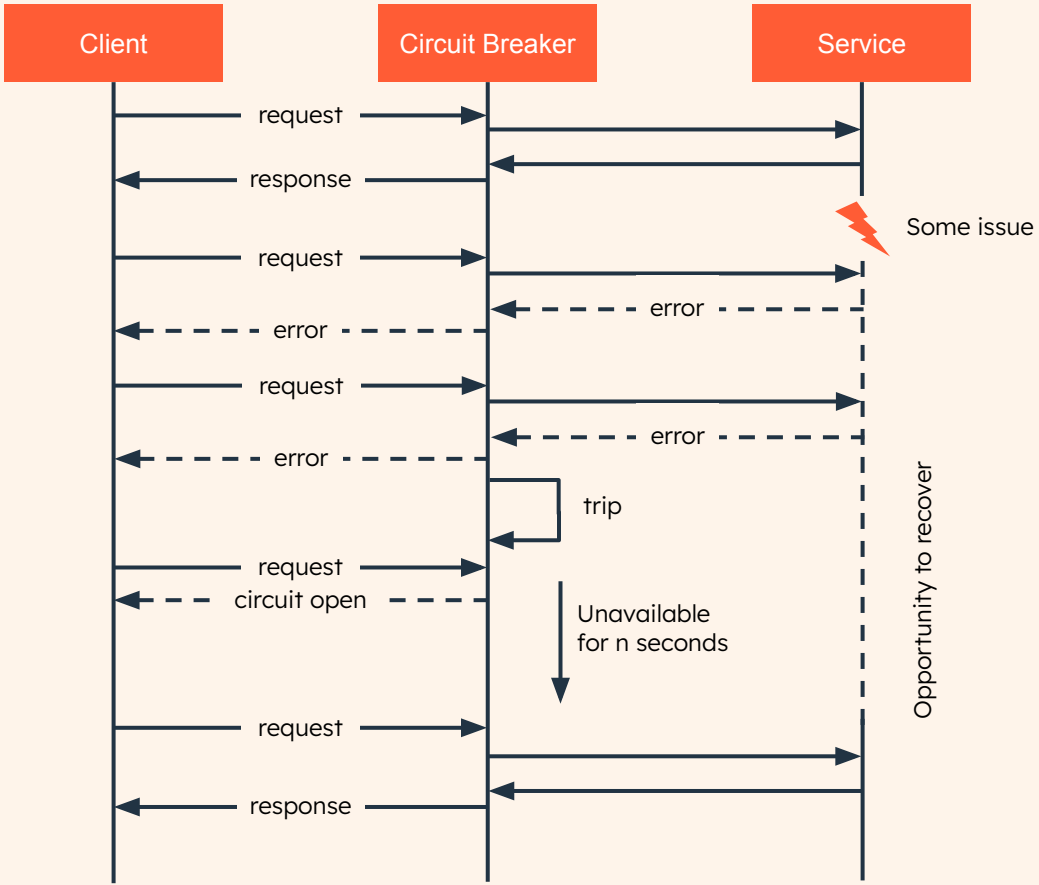


- Proxy/monitor HTTP responses
- If to many errors appear consecutively, the circuit breaker trips and all subsequent requests will automatically fail for a period of time
- After the period, we retry if the service recovered again

You can model the internals as states (State Pattern)

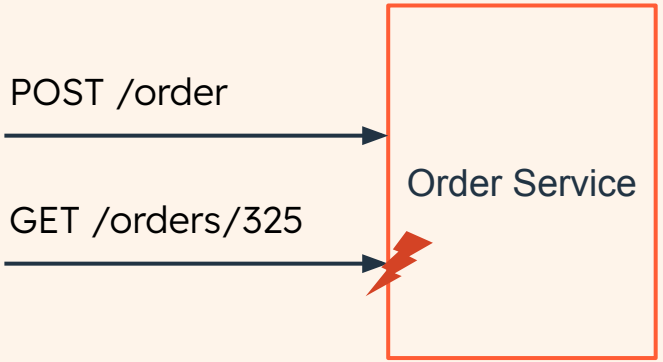


Circuit Breaker



[4] https://microservices.io/patterns/reliability/circuit-breaker.html

Example: Food Delivery



Episode III

The rebellion is very satisfied. Our droid factories successfully produce different types of droids (R2-series, 3PO-series, etc.). We have an operator to manage the production line and coordinate which droids to produce.

Unfortunately, a traitor managed to sabotage the R2 production line and now it's broken. With the operator still re-trying to produce new R2 droids, the factory doesn't have the breathing room to recover. We risk that the error spreads to other production lines and the factory stops operating.

Implement a circuit breaker so that our factory continues to function.

Agenda

#0 Introduction to Microservices

#1 Strangler Pattern

#2 Decentralized ID Generation

#3 Circuit Breaker

#4 Asynchronous Messaging

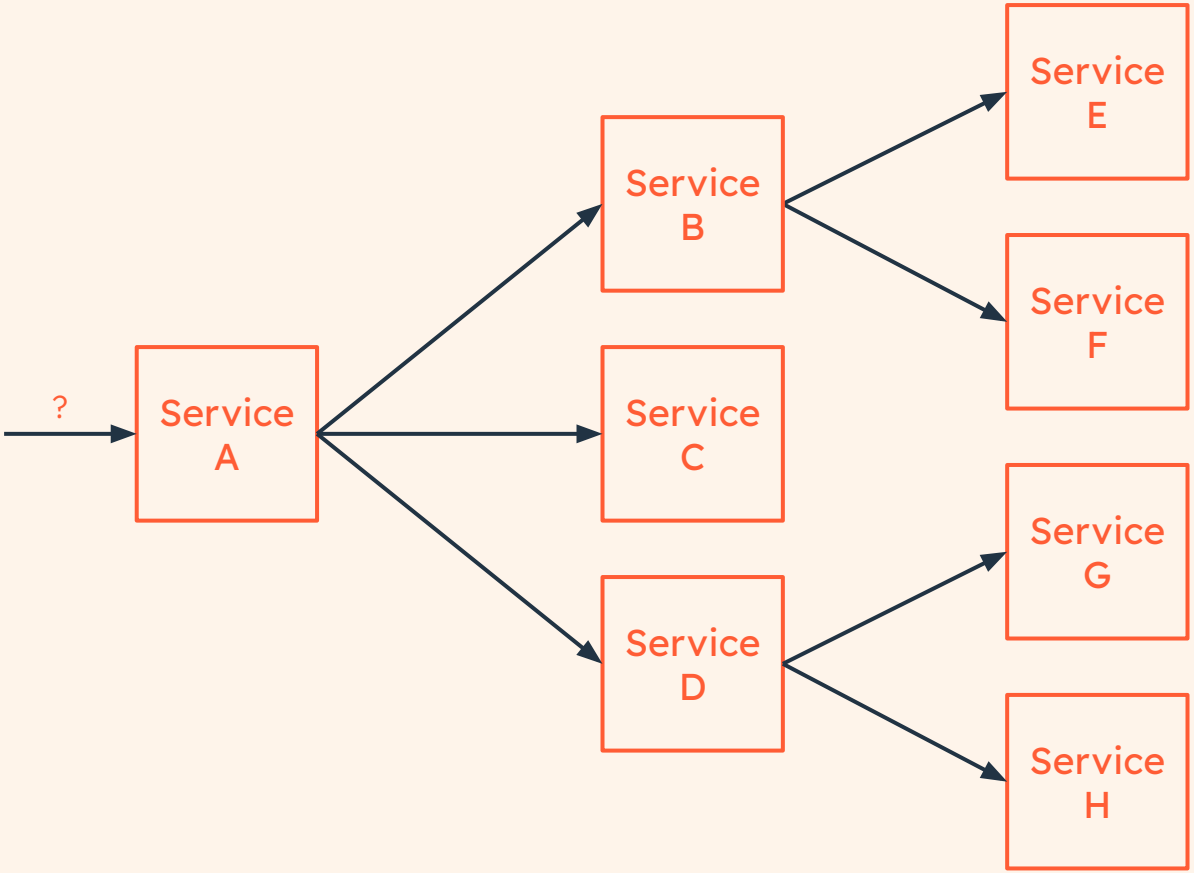
Asynchronous Messaging

Problem

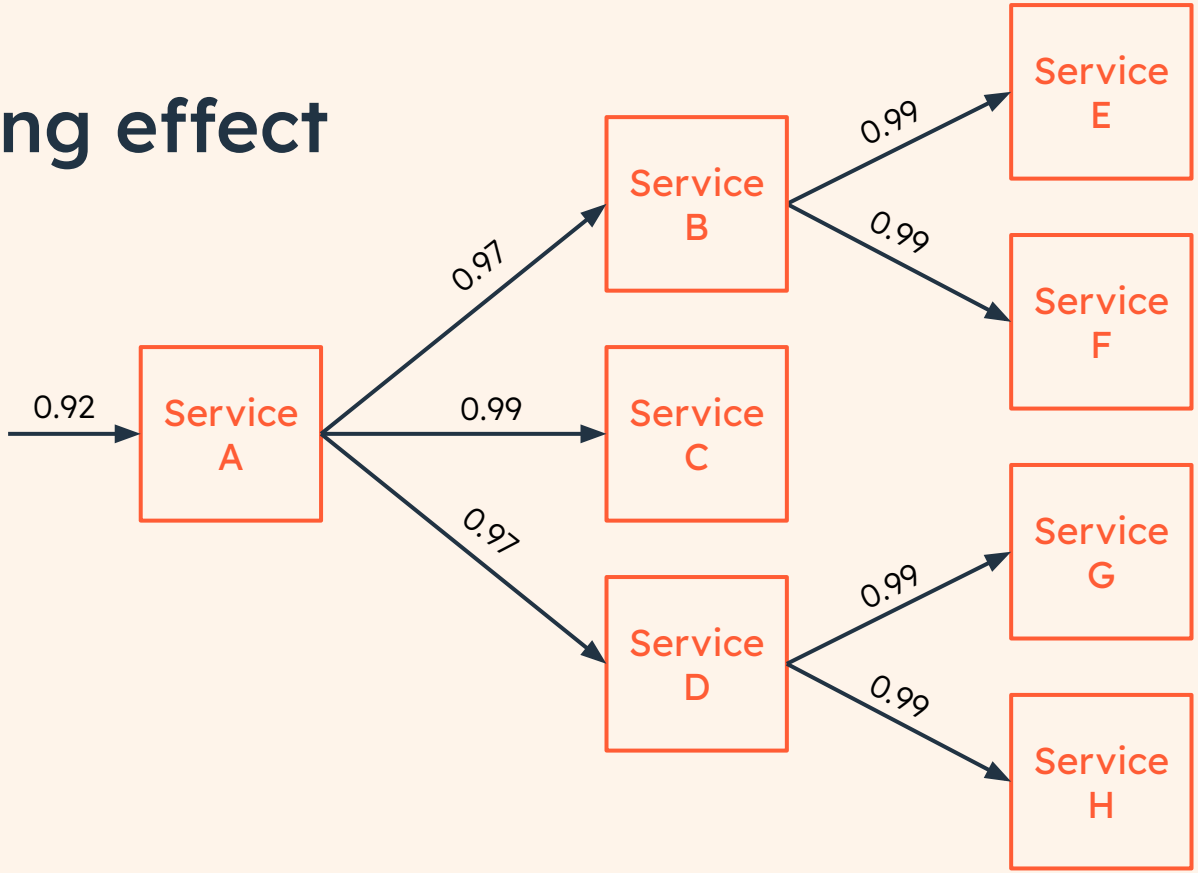
Synchronous communication leads to tight runtime coupling.

- Both services must be available for the duration of the request
- Services that have a high latency by design (e.g. neural networks) negatively impact your latency
- Peaks with high load saturate your services and threads

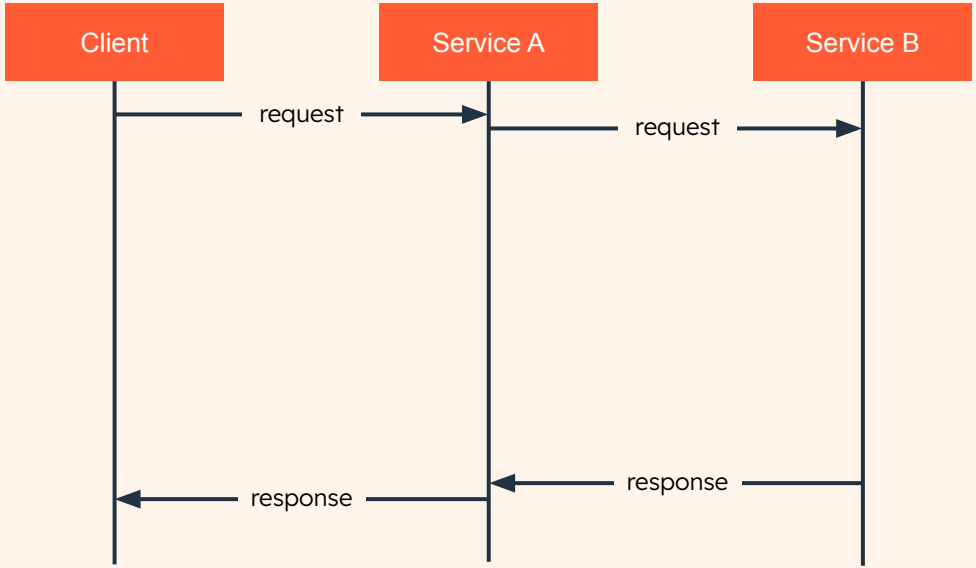
What success rate will we have, if every service has a guaranteed uptime of 0.99?

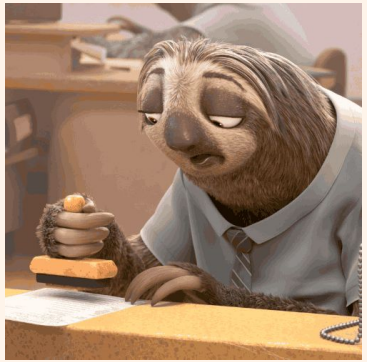


Compounding effect

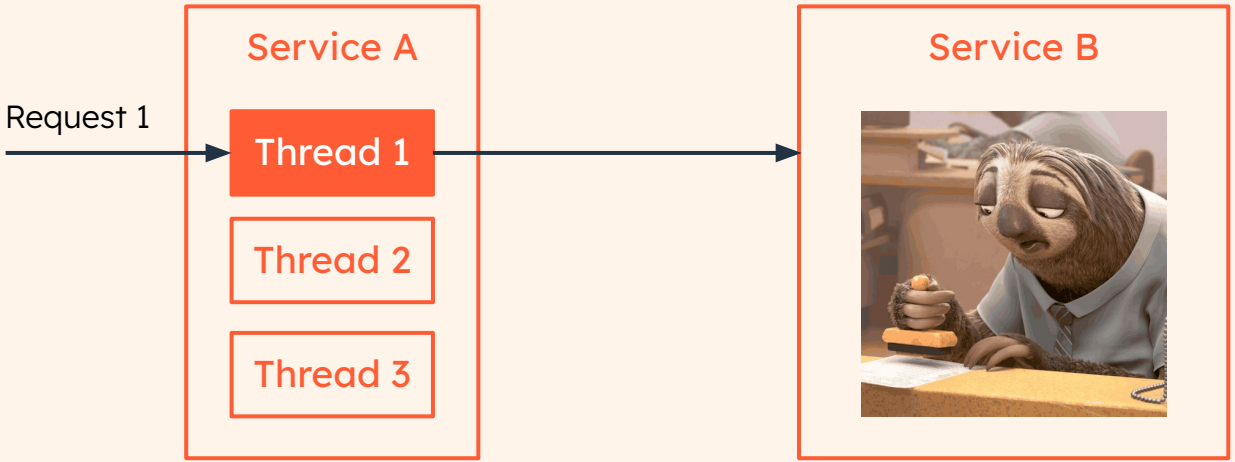


The client waits until everything is complete

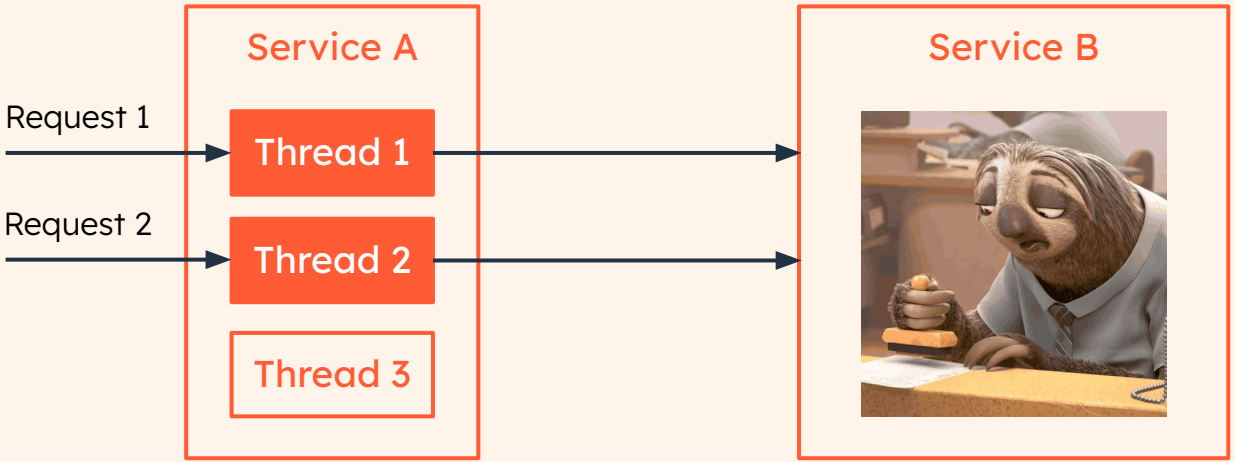




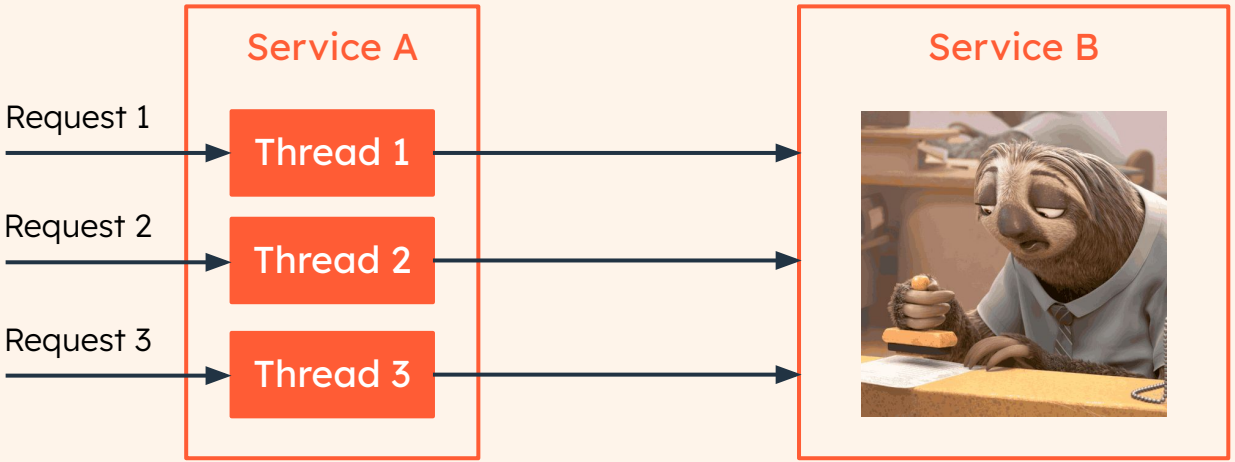
Peaks with high load saturate your threads



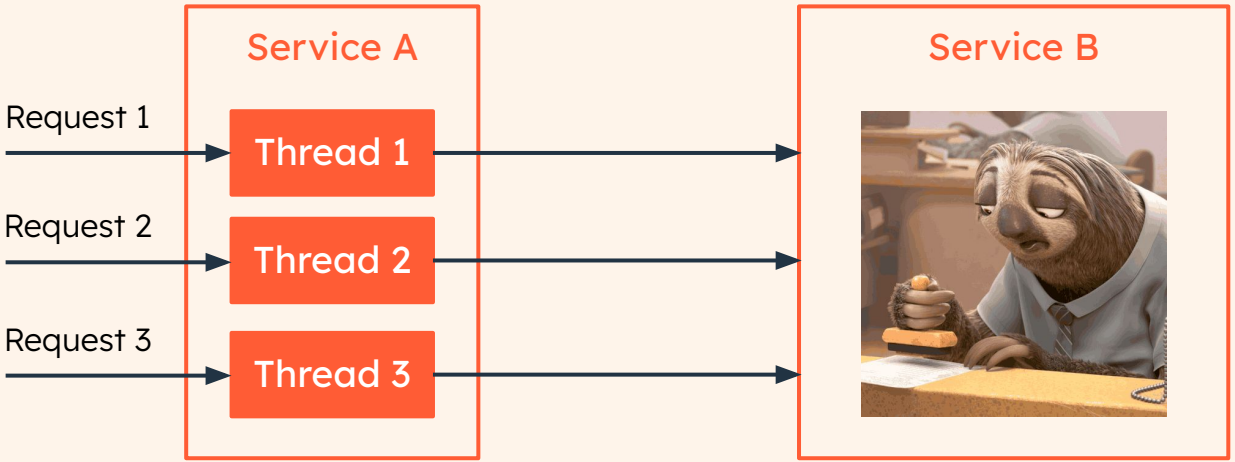
Peaks with high load saturate your threads



Peaks with high load saturate your threads



Peaks with high load saturate your threads

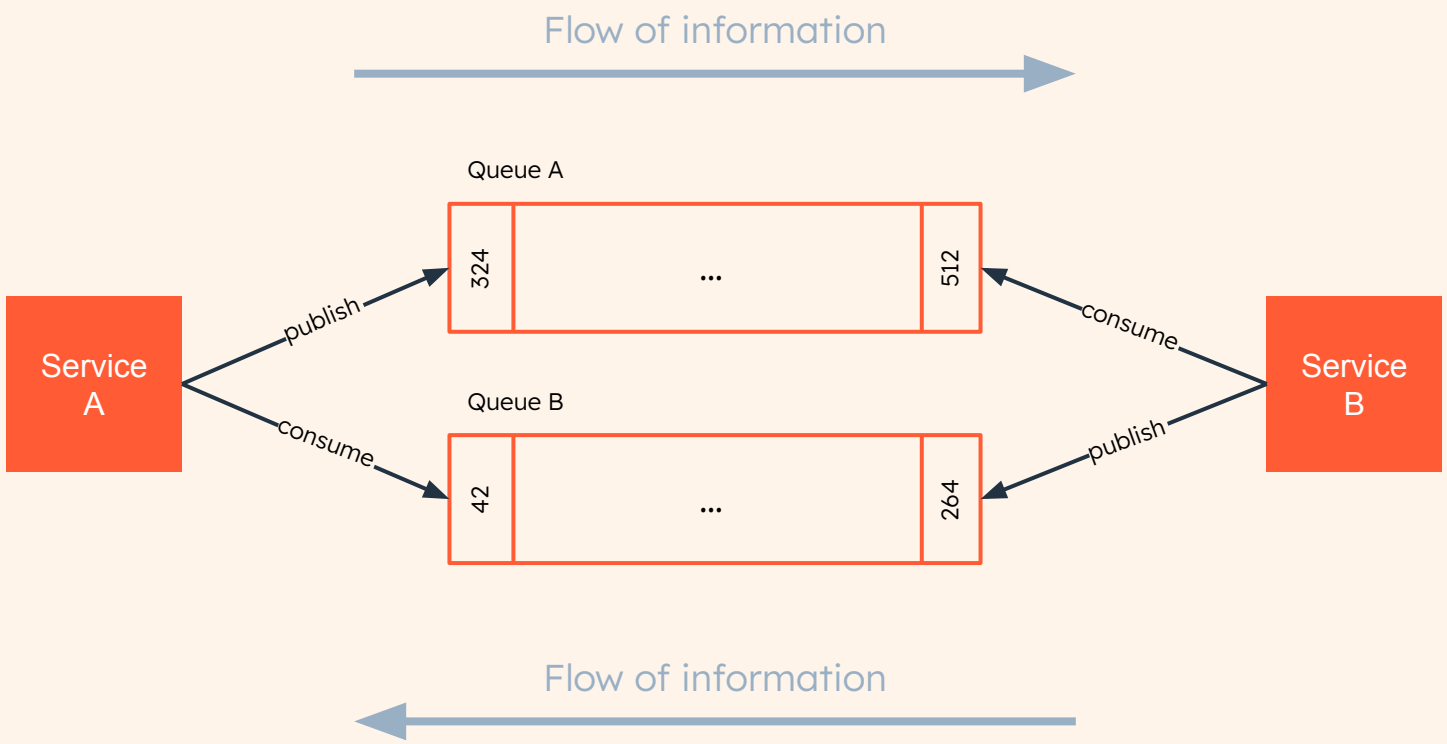


Request 4?

Solution

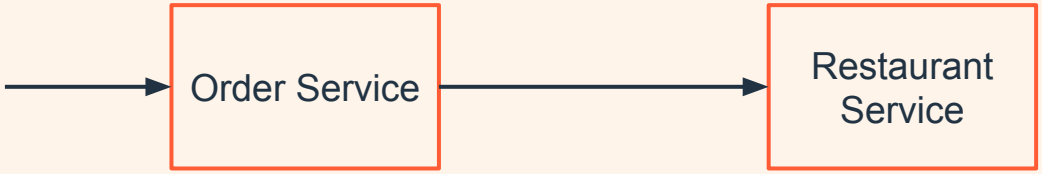
- Asynchronous communication
- Usually achieved via message queues (e.g. Kafka, RabbitMQ) aka Observer Pattern
- You can also use a database

Asynchronous Messaging



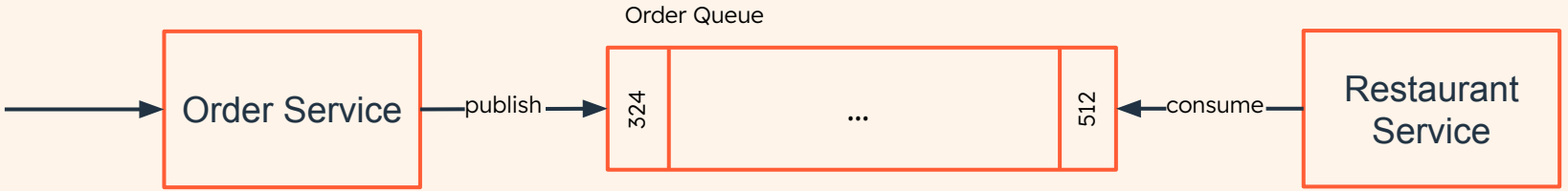
[5] https://microservices.io/patterns/communication-style/messaging.html

Example: Food Delivery



Scenarios:

1. Restaurant Service is unavailable
2. Restaurant Service has high latency
3. High volume of orders



Episode IV

In our droid production facilities it can take quite long until a droid is assembled. While that happens, the operator sits idle until the droid is finished. Unfortunately, we faced occasions when the operator got mad and didn't want to wait any longer. This ment we lost the produced droid (HTTP timeout).

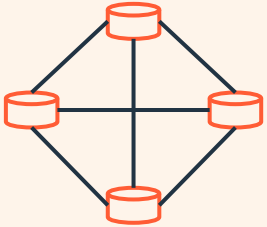
The operator came up with a new idea and created a new inbox where she wants to get notified when a new droid is ready.

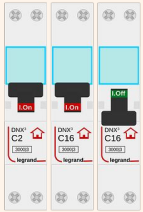
Implement the async messaging pattern for our production facilities.

Summary

The strangler pattern helps us to transition from a monolithic architecture to a microservice architecture.

Decentralized ID generation allows us to distribute storage and increase the throughput of our system.



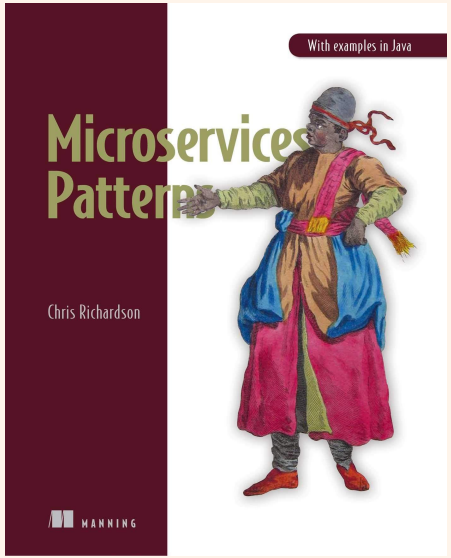


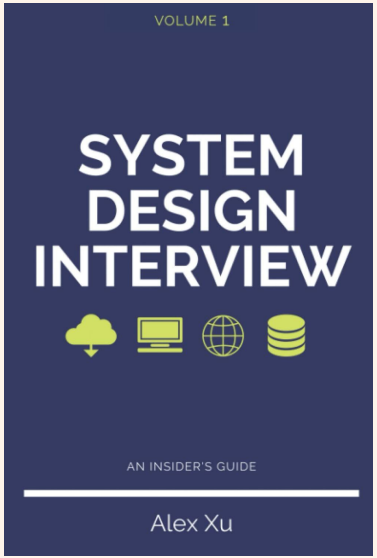
The circuit breaker pattern helps us to keep issues contained and give a service the opportunity to recover.

The asynchronous messaging pattern helps us to decouple tight runtime constraints.

Resources

[1] https://martinfowler.com/articles/microservices.html
[2] https://microservices.io/patterns/refactoring/strangler-application.html
[4] https://microservices.io/patterns/reliability/circuit-breaker.html
[5] https://microservices.io/patterns/communication-style/messaging.html





How to stay in touch with us

- **Sign up** for the upcoming 'Coding Assessment' Workshop organized by HubSpot by submitting the interest form. We will reach out to you soon to provide further details and schedule.
- Entry-level roles and internships we offer you can find via this link
- LinkedIn and TikTok under **@Studentspot**
- Instagram and YouTube under **@hubspotlife**
- Connect with us on LinkedIn: Fahad Ani, Raoul Friedrich
- Or each out via email: fani@hubspot.com, rfriedrich@hubspot.com