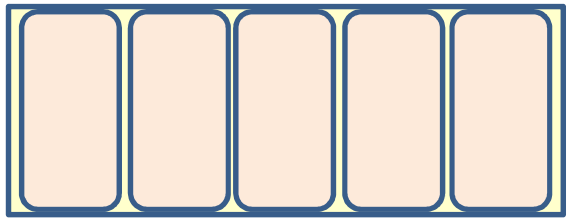
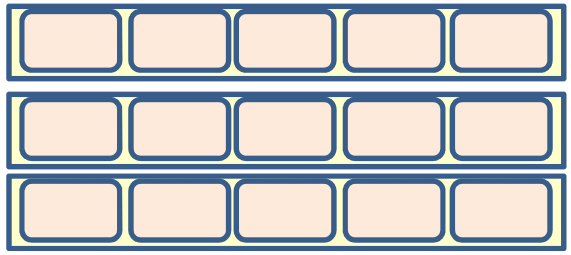
Final review

MICROSERVICES

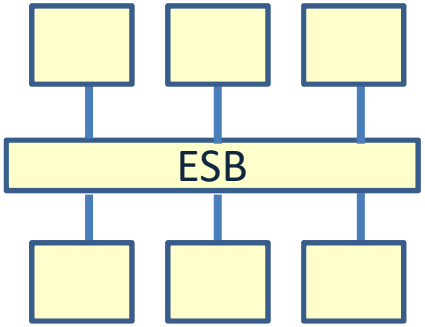
Architecture styles



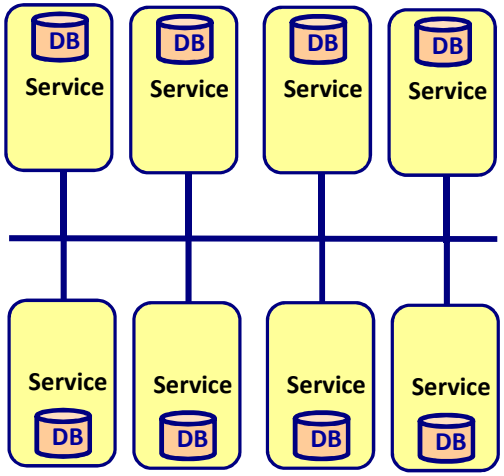
Layering



Components

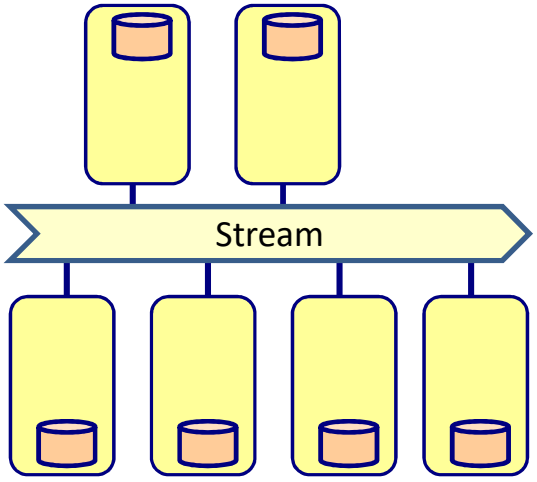


Service Oriented Architecture



Microservice architecture

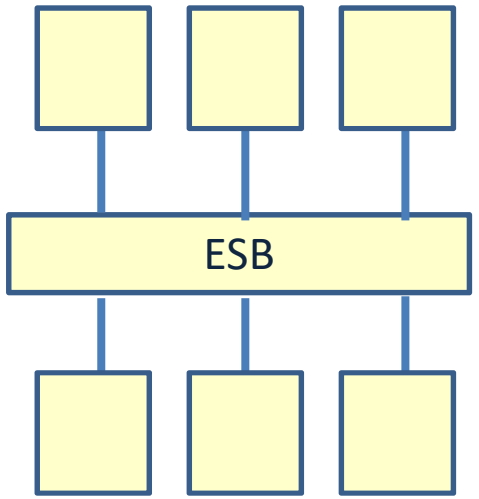
Event Driven Architecture



Stream based architecture

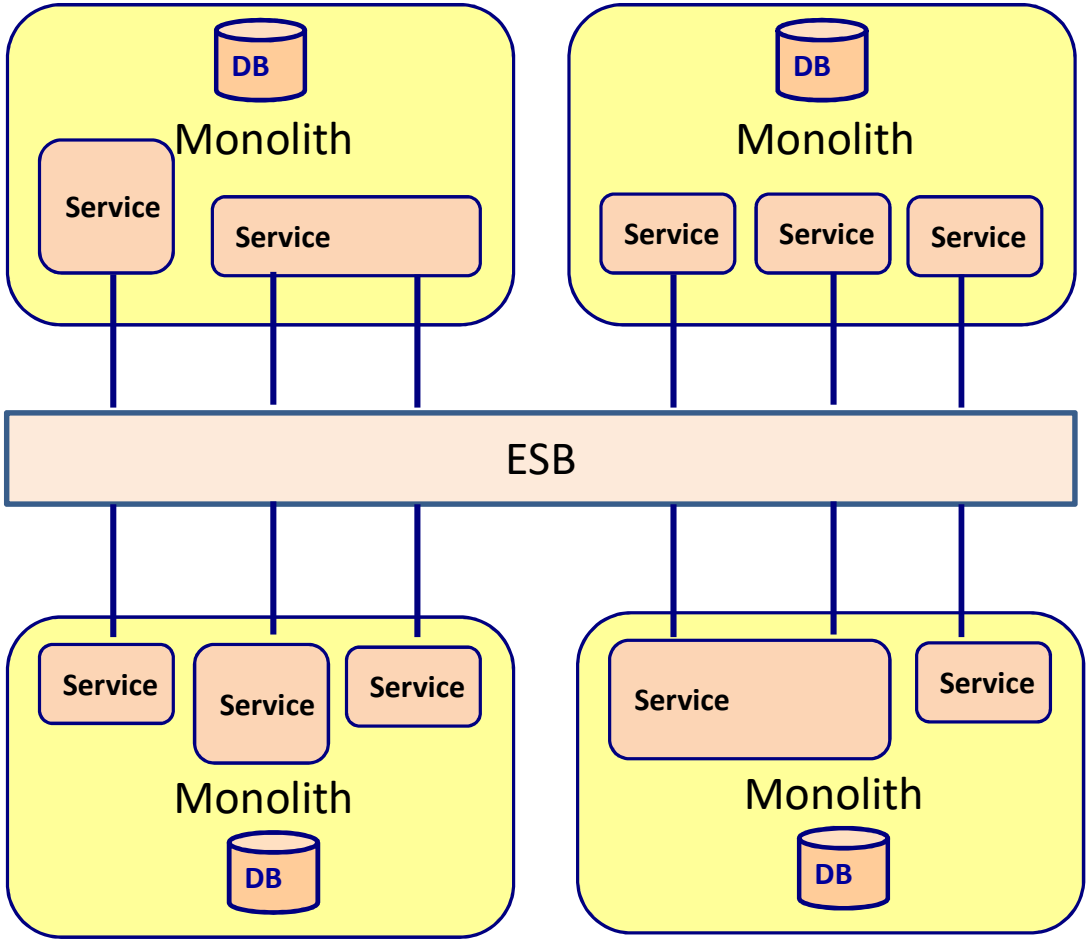
Service Oriented Architecture

- Disadvantages
  - Complex ESB
  - Changing the business process while still business processes are running is very difficult
  - Most SOA's are build on top of monoliths



Problem with SOA

- Most SOA's are build on top of monoliths

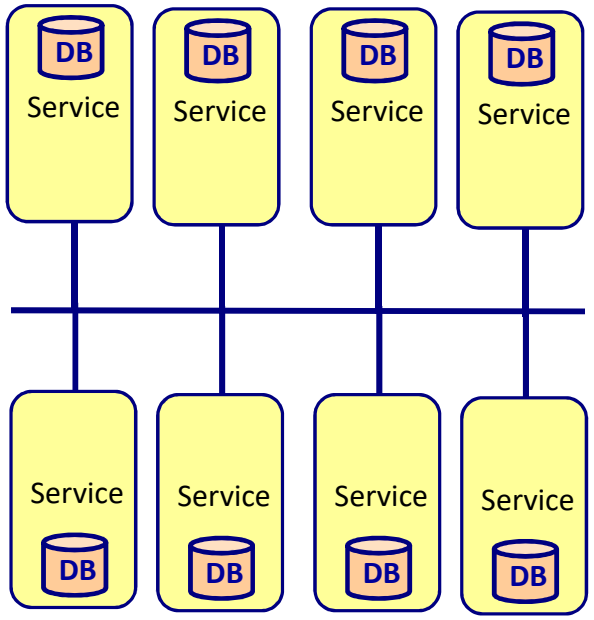


Problems with a monolith architecture

- Can evolve in a big ball of mud
- Limited re-use is realized across monolithic applications
- All or nothing scaling
- Single development stack
- Does no support small agile scrum teams
- Deploying a monolith takes a lot of ceremony

Microservices

- Small independent services
  - Simple and lightweight
  - Runs in an independent process
  - Language agnostic
  - Decoupled

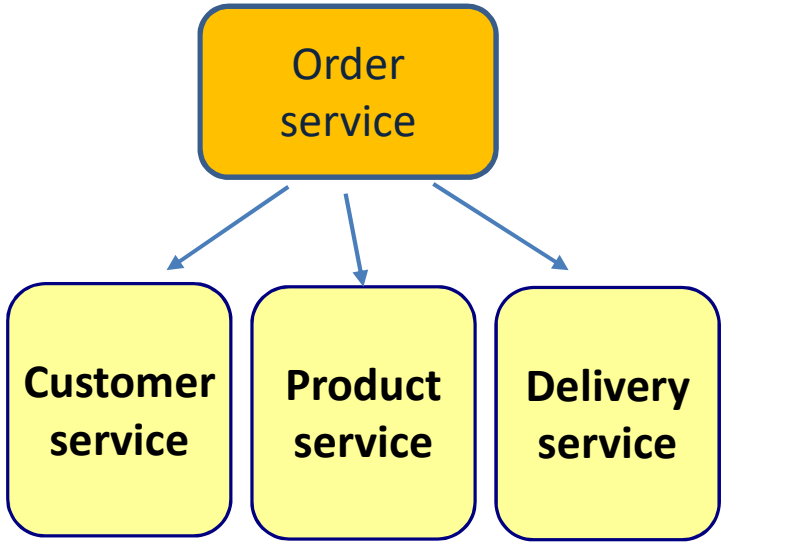


Appropriate boundaries

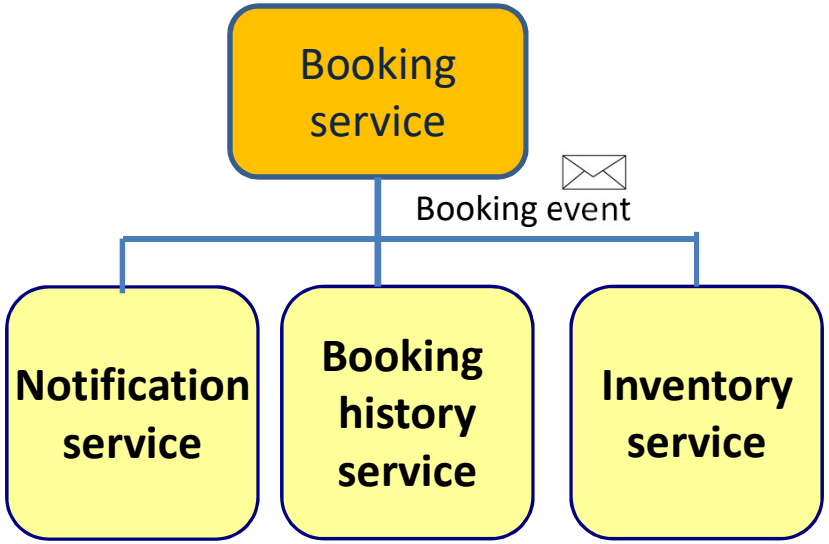
- DDD bounded context
- Autonomous functions
- Size of deployable unit
- Most appropriate function or subdomain
- Polyglot architecture
- Selective scaling
- Small agile teams
- Single responsibility
- Replicability or changeability
- Coupling and cohesion

Orchestration vs. choreography

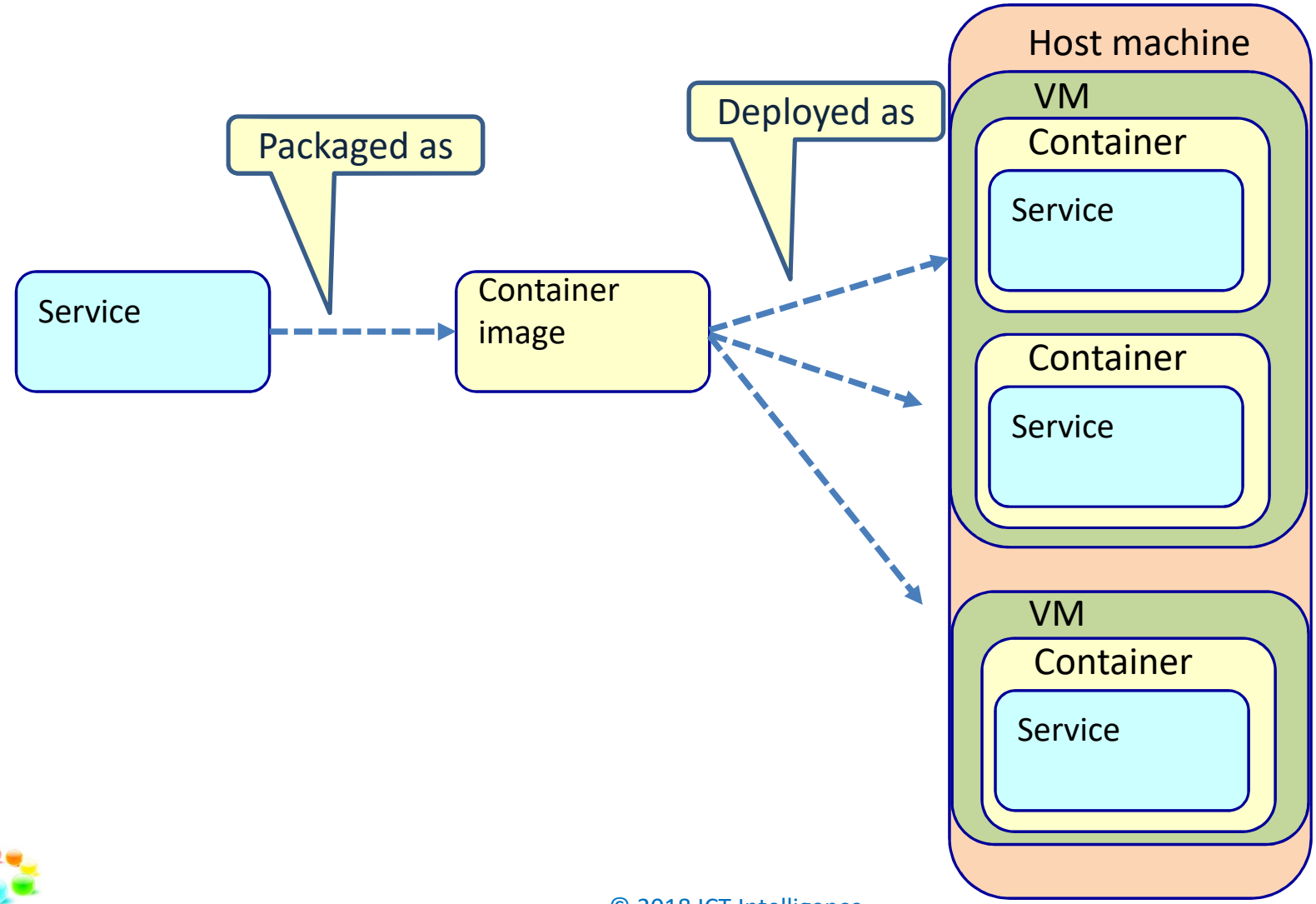
- Orchestration
  - One central brain



- Choreography
  - No central brain



Service per container



From monolith to microservice

- Not a big bang
- Strangler approach
- Separate front-end and back-end
- Extract a service

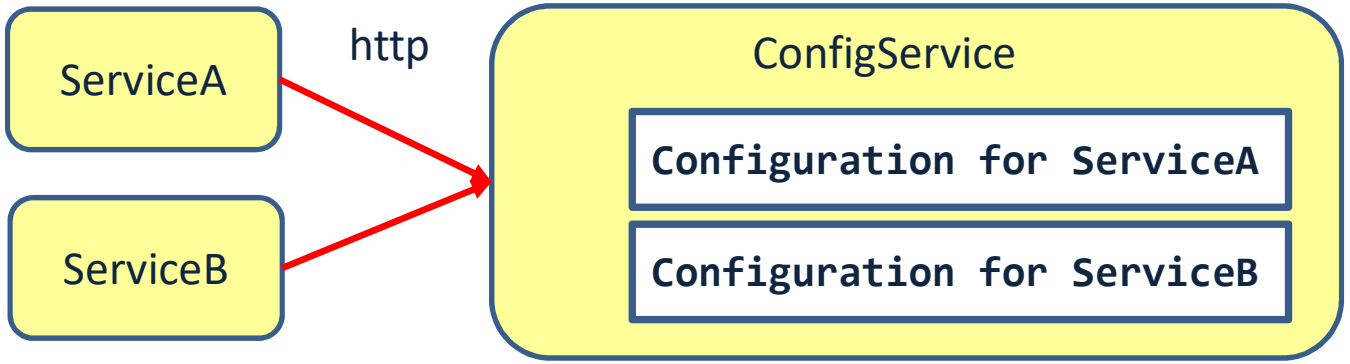
Why microservices?

- Agility
  - Much easier to respond to change
- Testability
  - Easier to test
  - Scope is smaller
- Deployability
  - Less ceremony
  - Less risk
- Scalability
- Availability
  - Fault tolerance

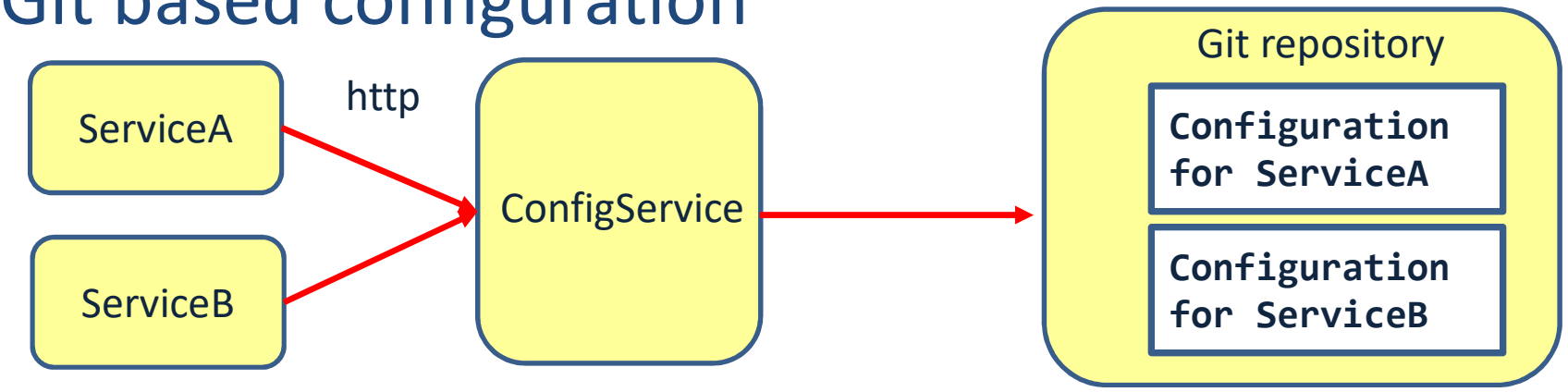
CONFIG SERVICE

Spring cloud config

- File based configuration



- Git based configuration

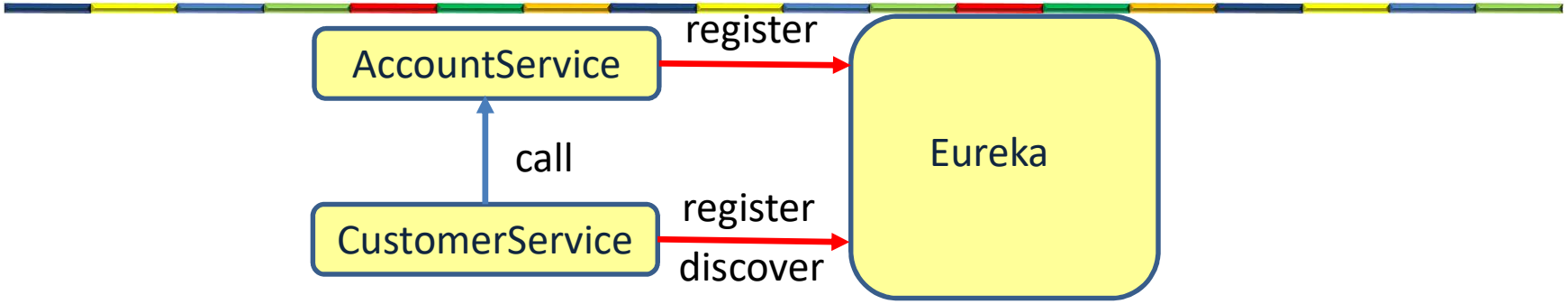


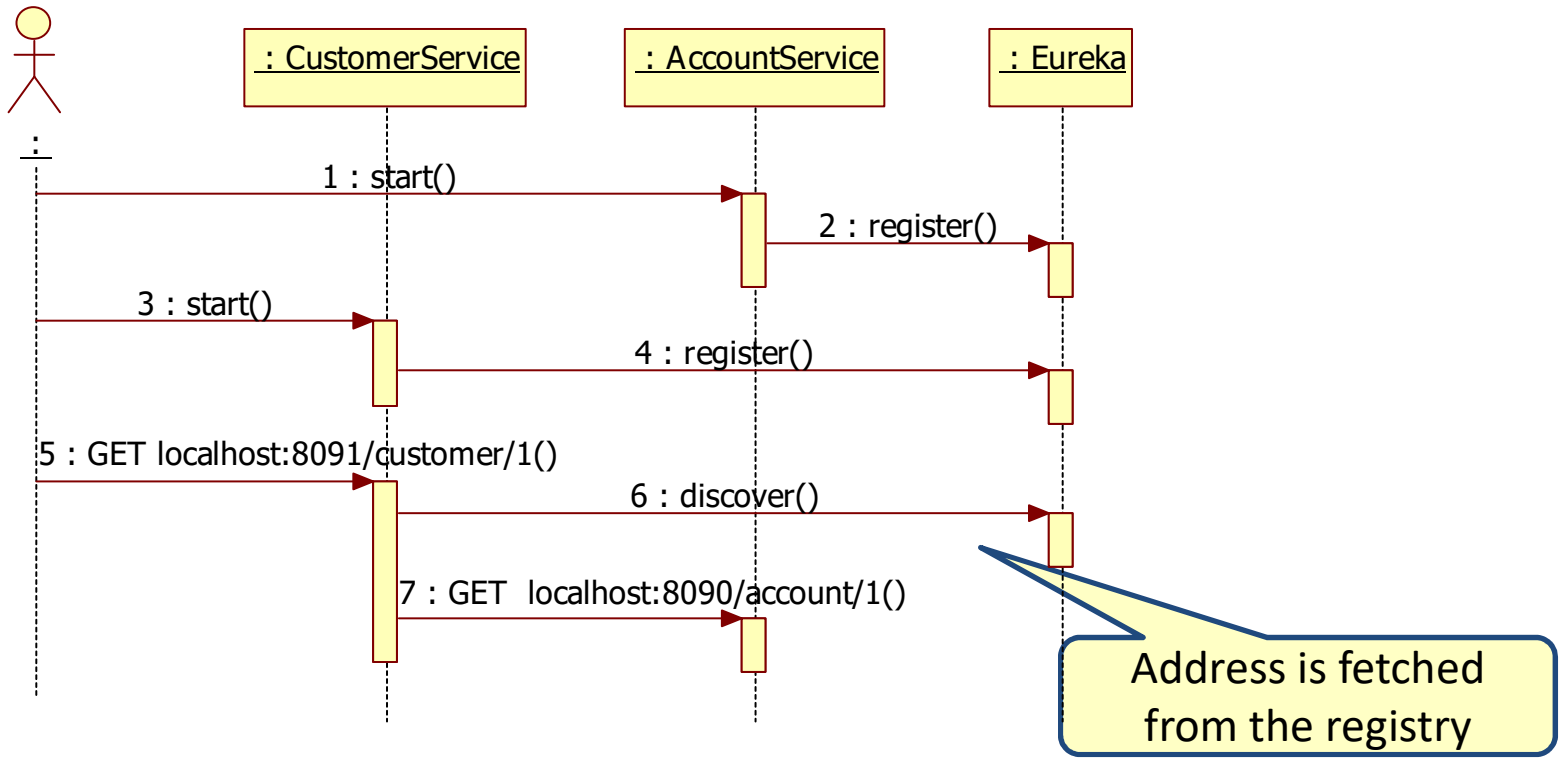
NETFLIX

EUREKA

SERVICE REGISTRY: EUREKA

Using Eureka



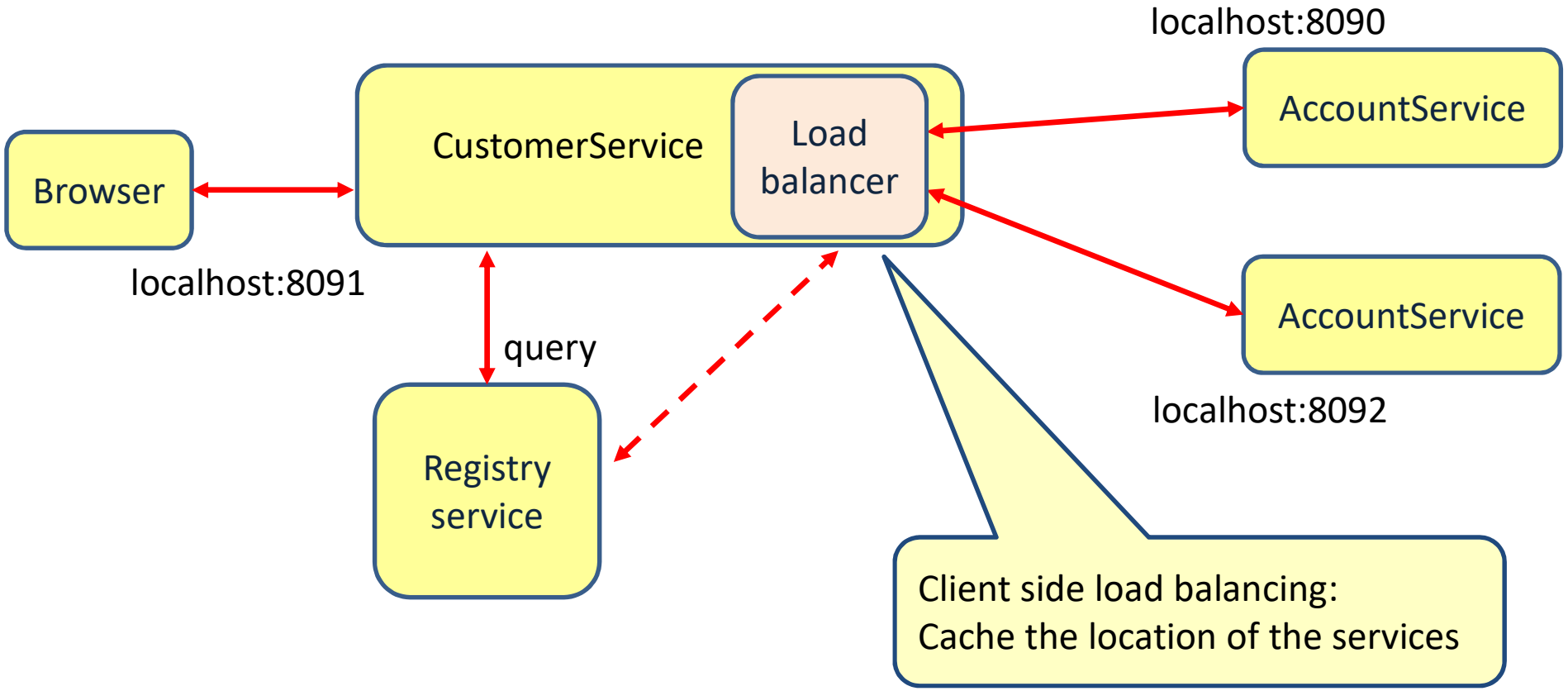


NETFLIX

RIBBON

LOAD BALANCING: RIBBON

Load balancer

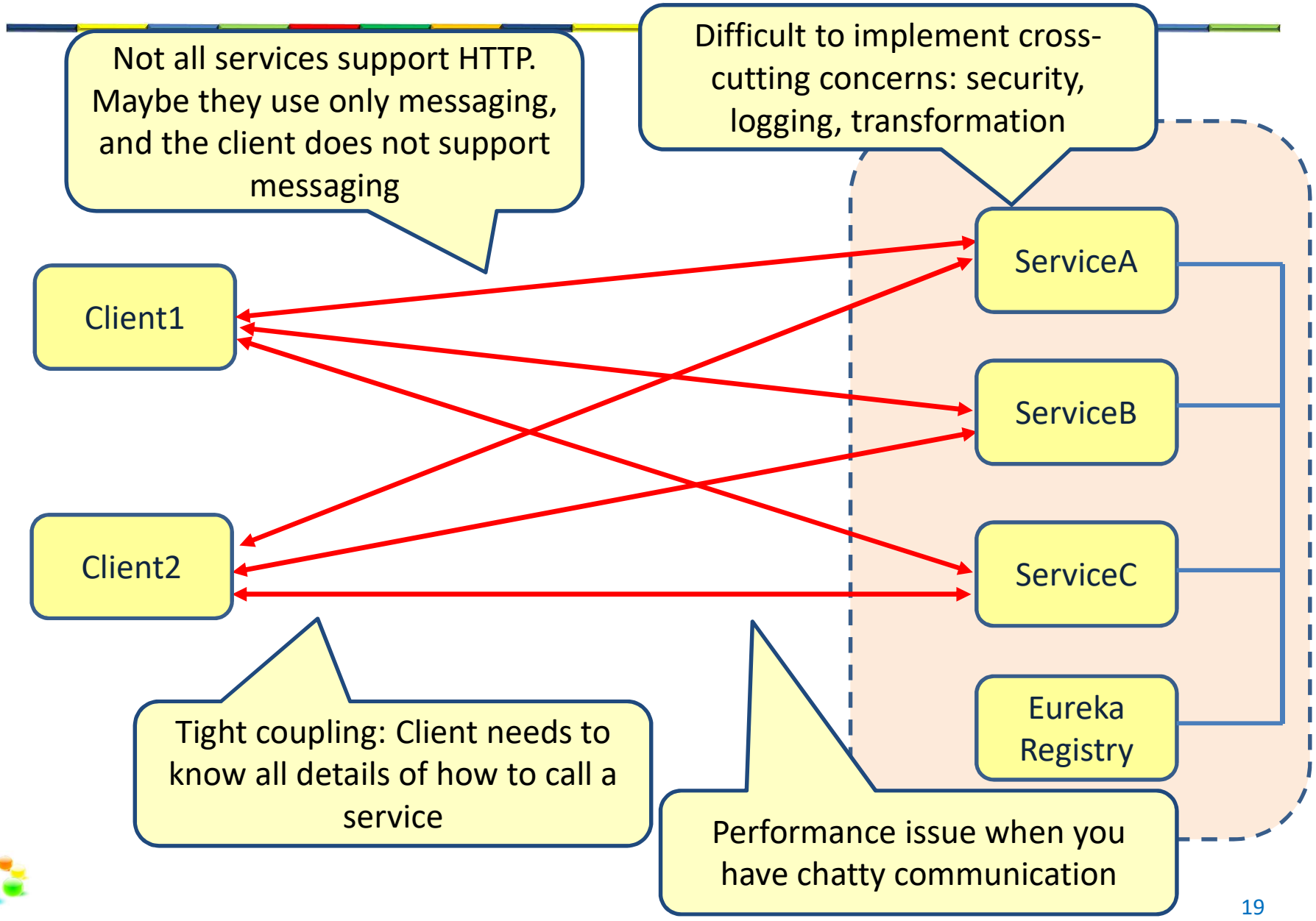


NETFLIX

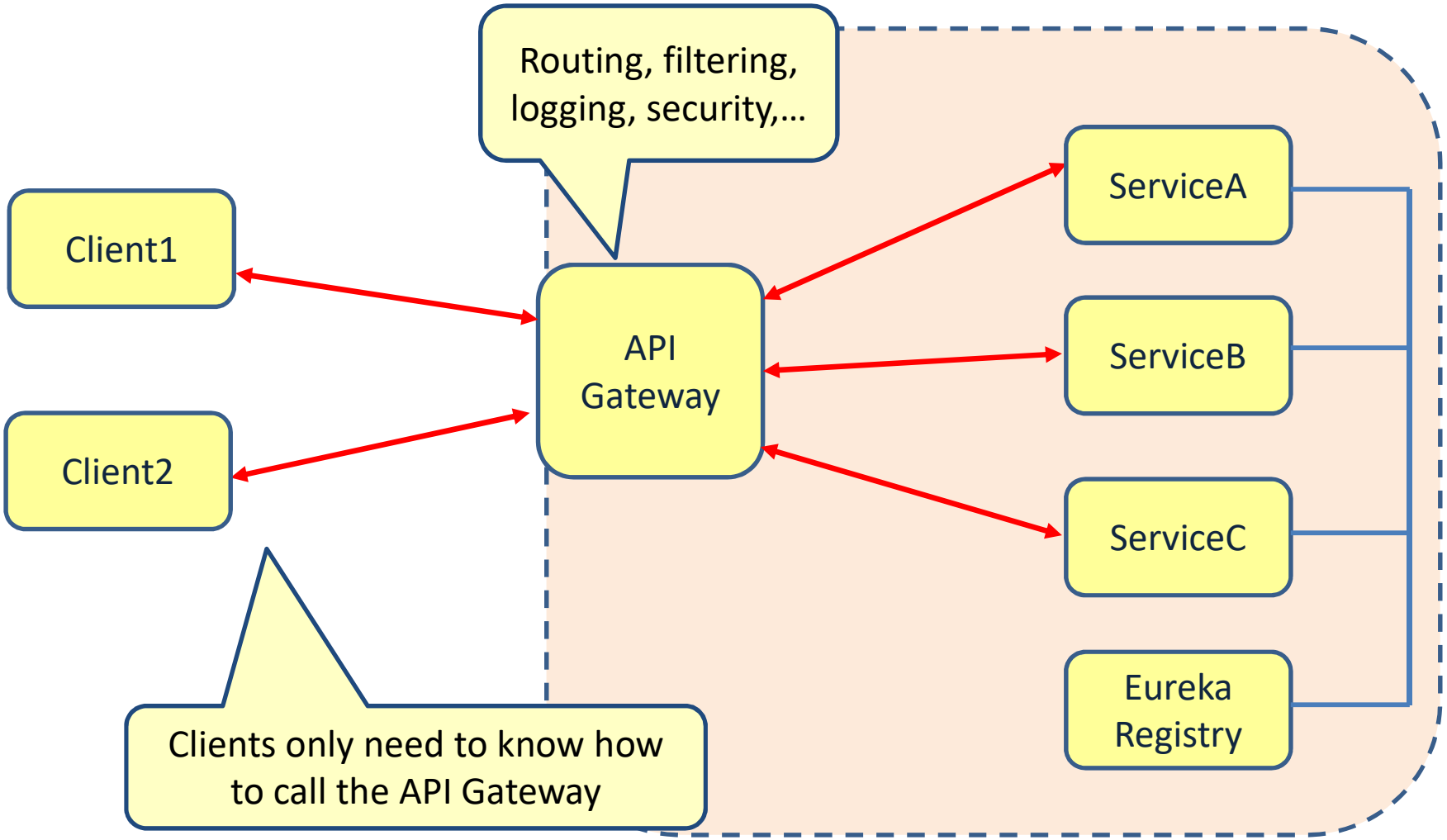
ZUUL

API GATEWAY: ZUUL

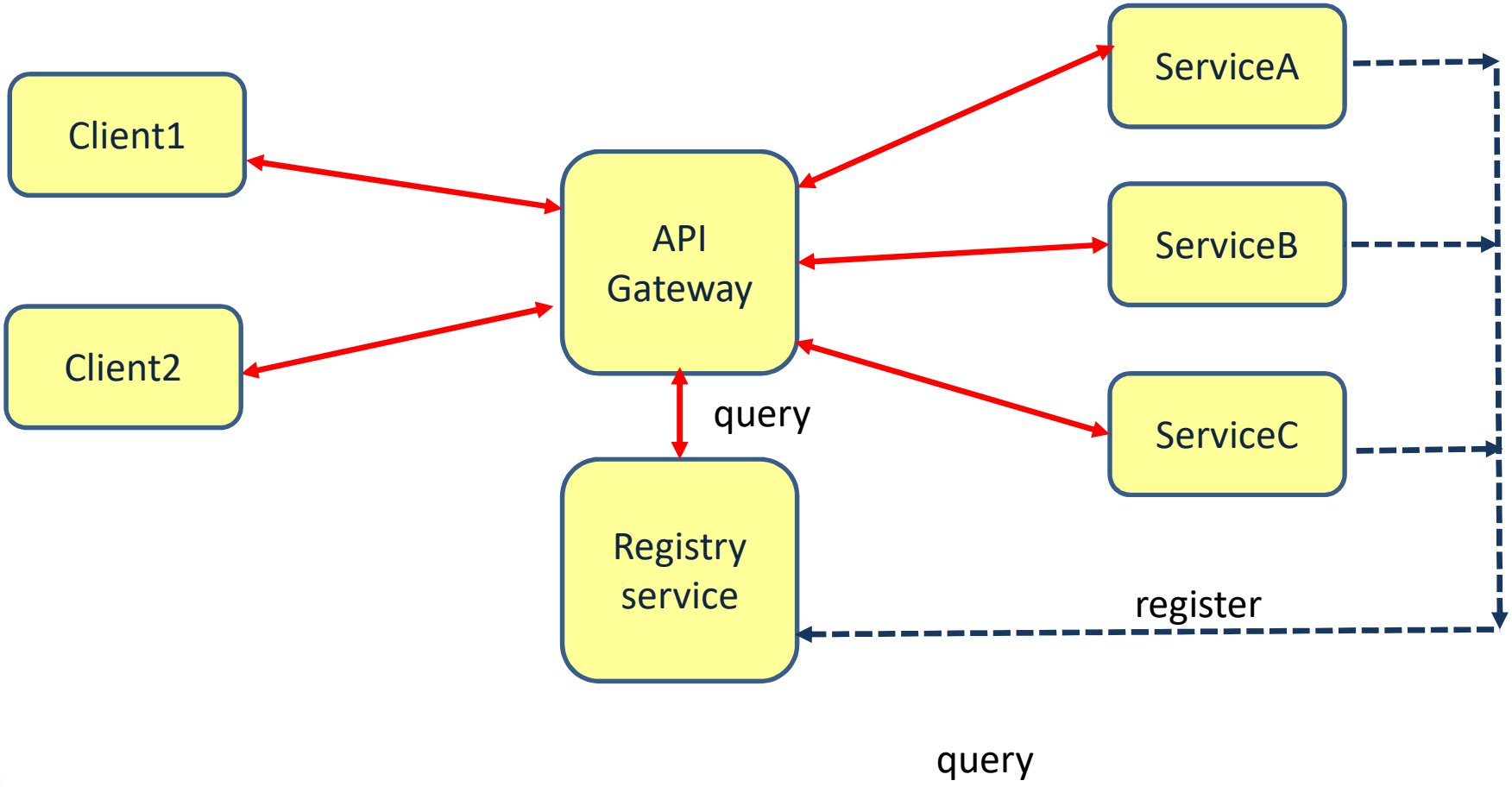
Adding clients



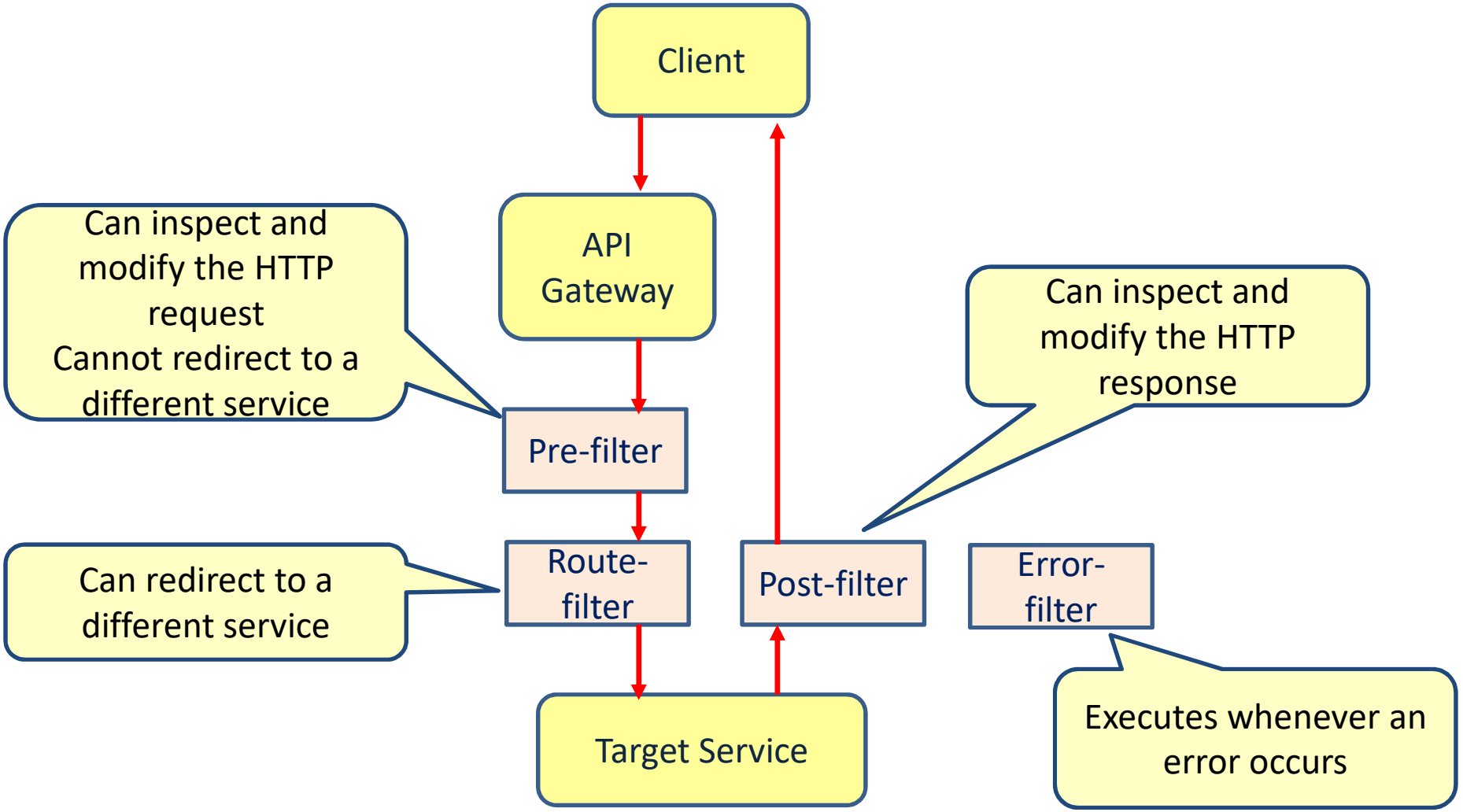
Api Gateway



Api Gateway and registry service

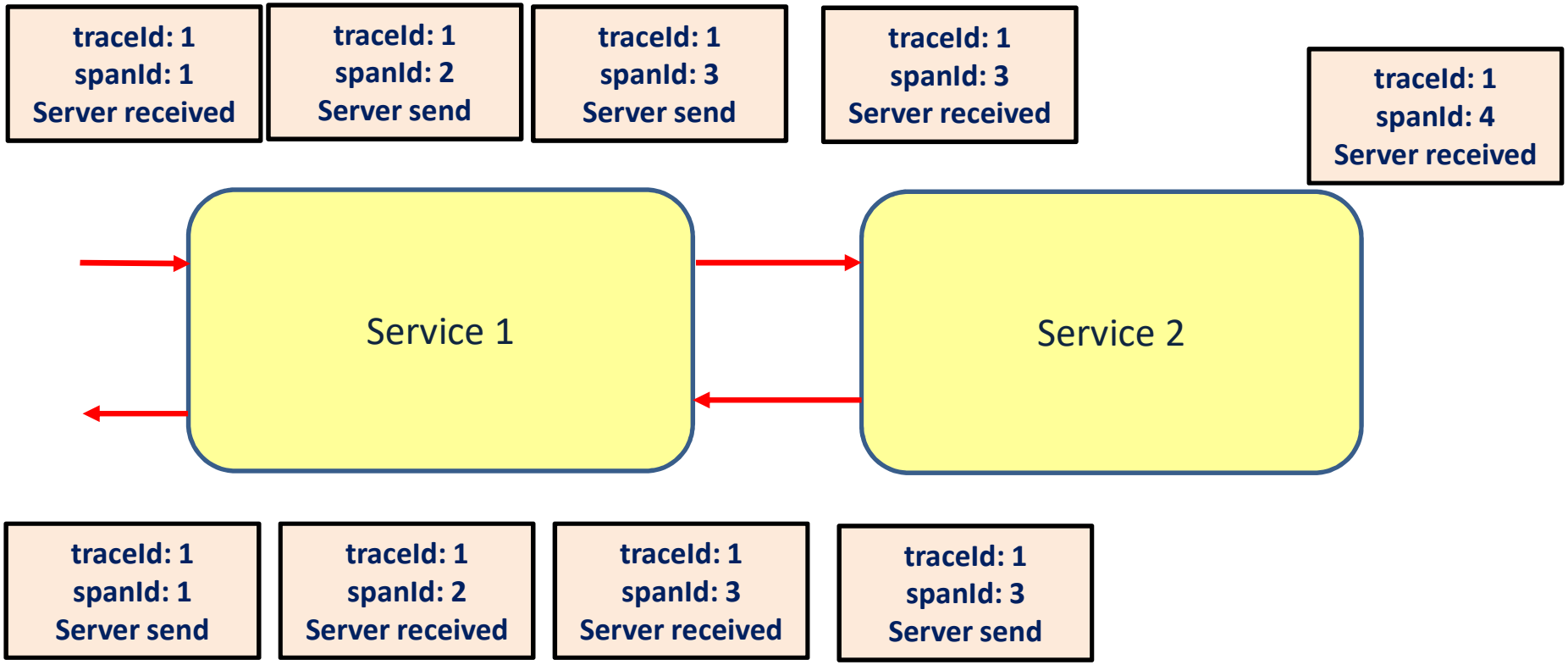


API Gateway Filters



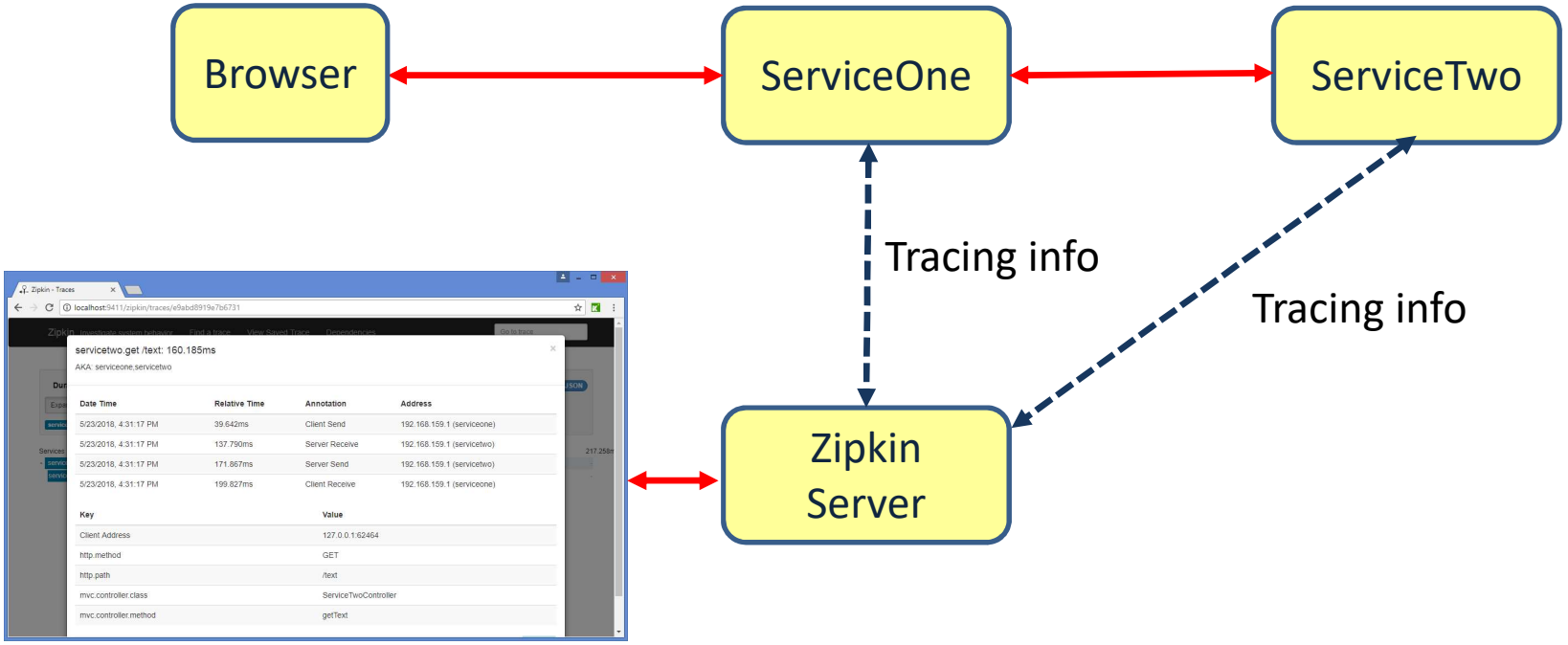
Spring cloud Sleuth

- Span: an individual operation
- Trace: a set of spans



Zipkin

- Centralized tracing server
  - Collects tracing information
- Zipkin console shows the data



ELK stack

Micro Service

Micro Service

Micro Service

Collect the log data

Make logs searchable

Visualize the data

logstash

elasticsearch

kibana

Collect logs
Transform
Time normalization

Schema less search DB
Highly scalable
Fast searching

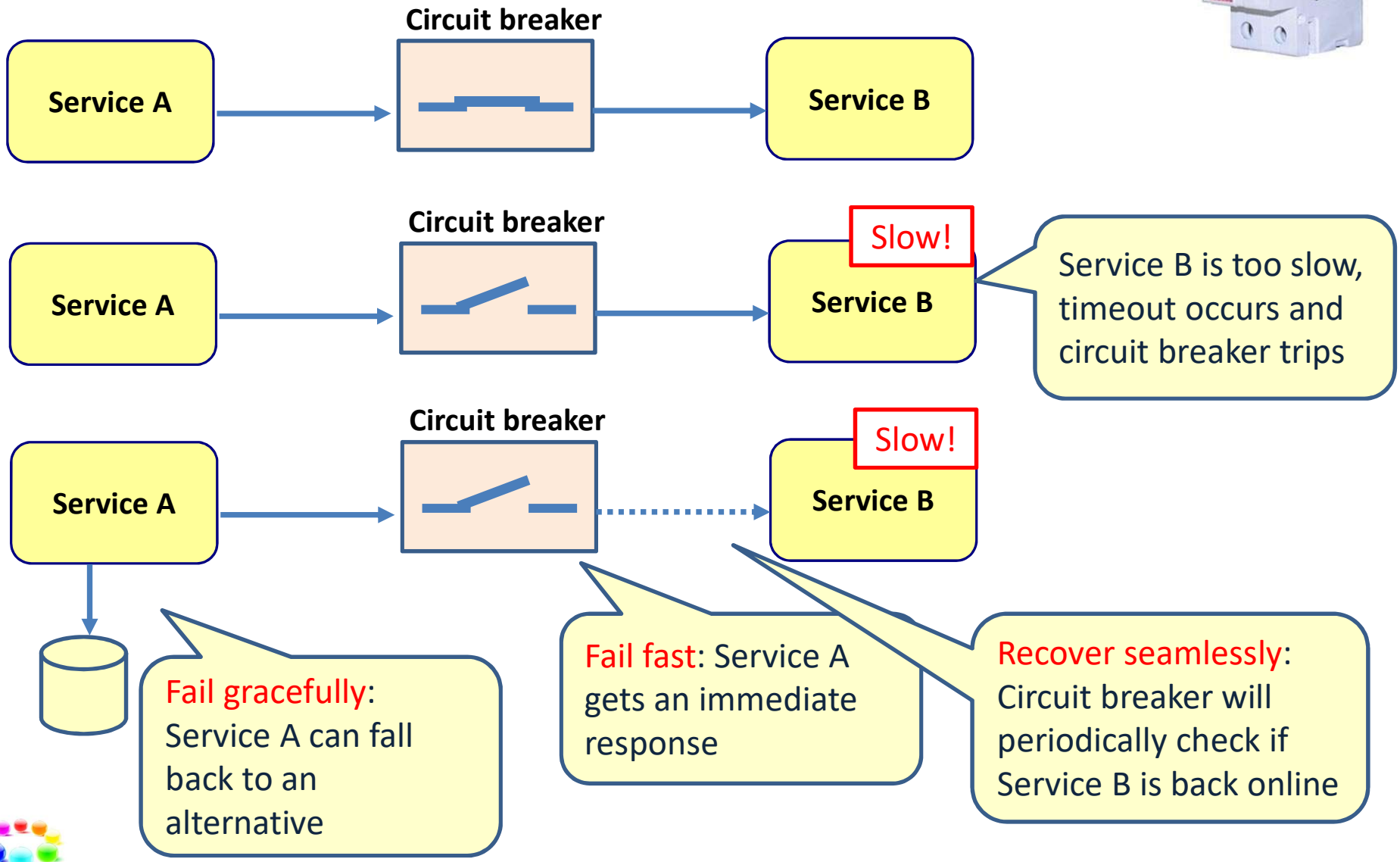
NETFLIX

RESILLIENCE: HYSTRIX

Timeouts

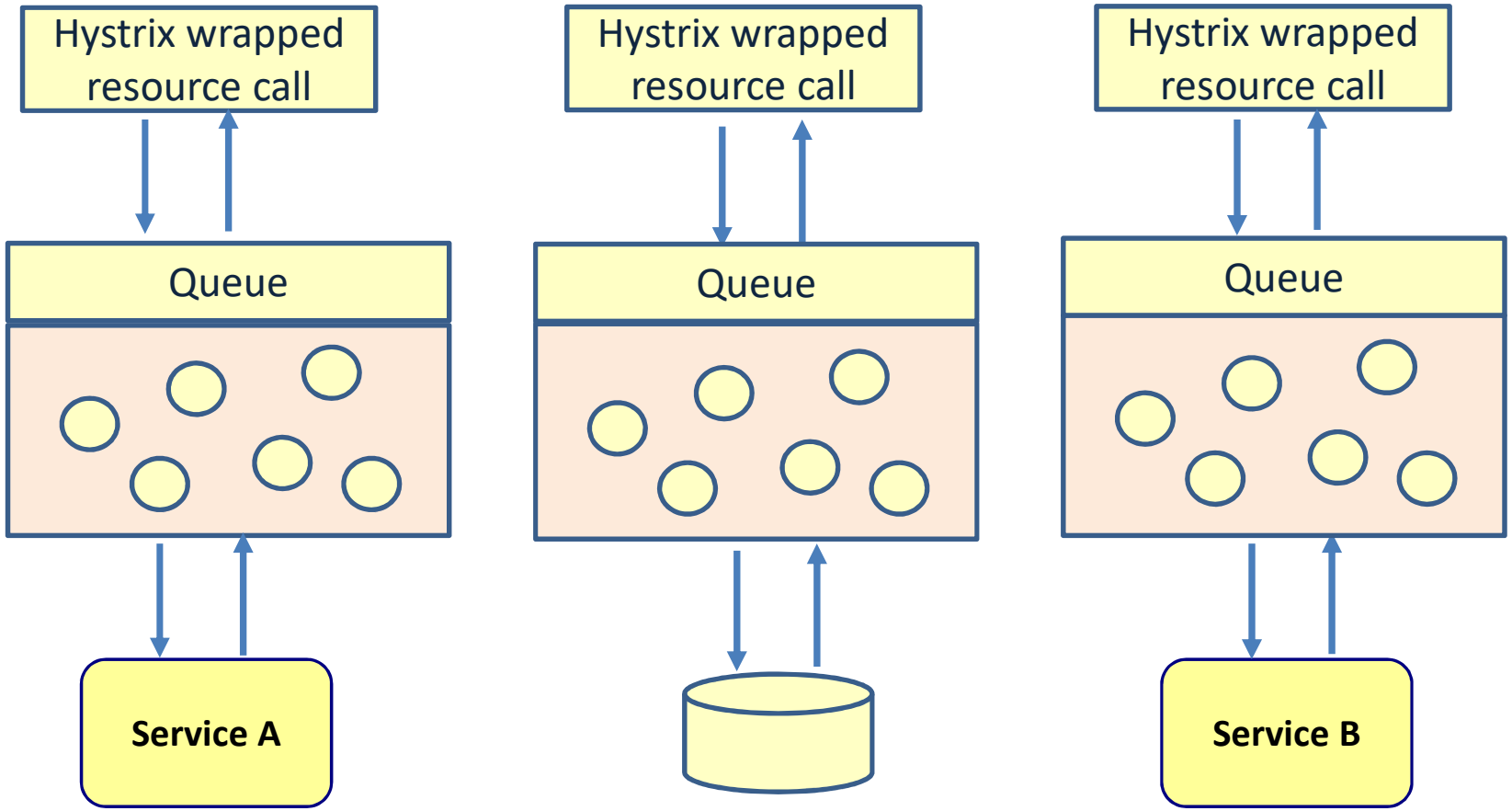
- Put timeouts on all out-of-process calls.
  - Other services
  - Database
  - File system
- Log when timeouts occur
  1. Pick a default timeout
  2. Monitor
  3. Adjust

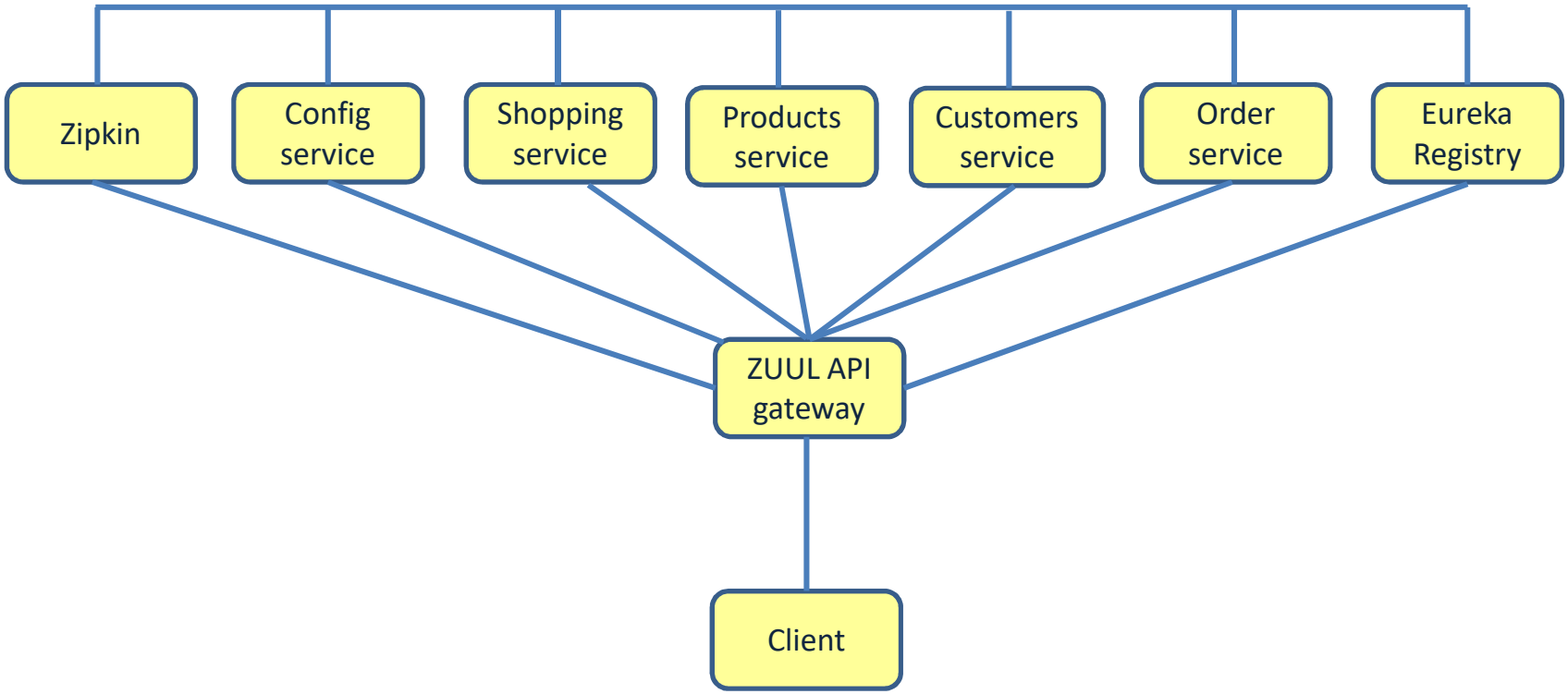
Circuit breaker

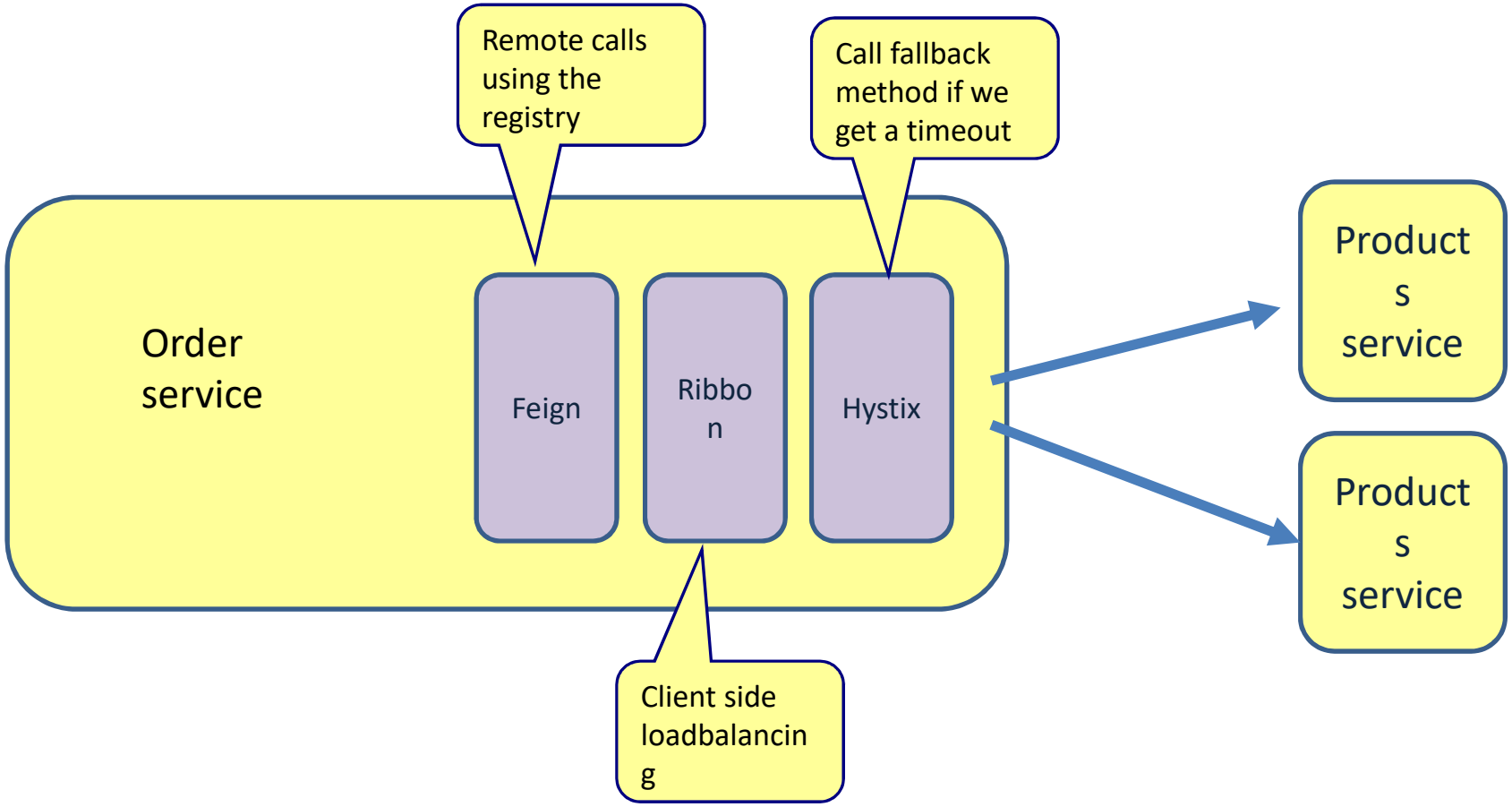


Hystrix bulkheads

- Hystrix uses a common thread pool for all remote calls



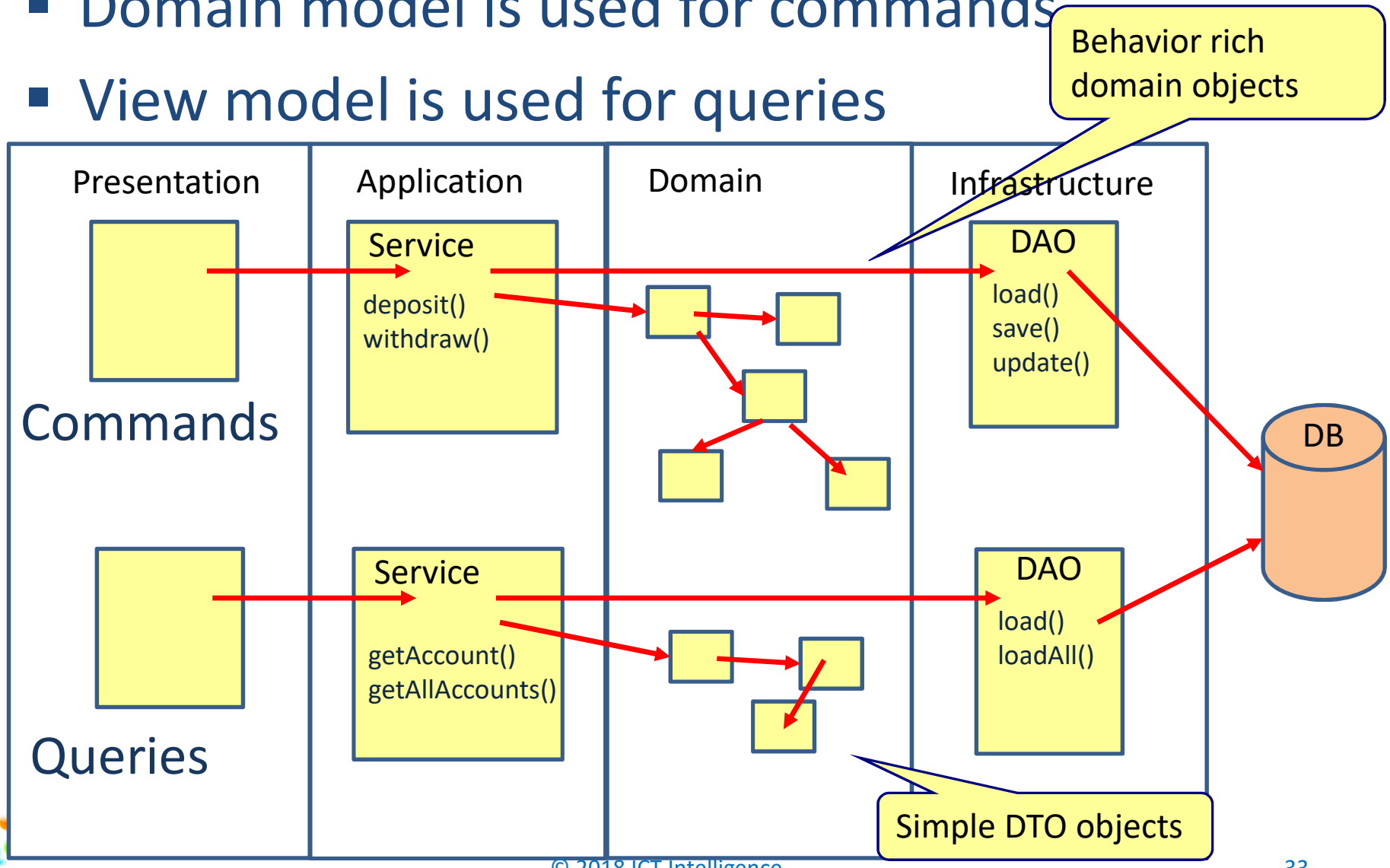

Zipkin
Config service
Shopping service
Products service
Customers service
Order service
Eureka Registry
ZUUL API gateway
Client


Remote calls using the registry
Call fallback method if we get a timeout
Order service
Feign
Ribbon
Hystix
Products service
Products service
Client side loadbalancing

CQRS

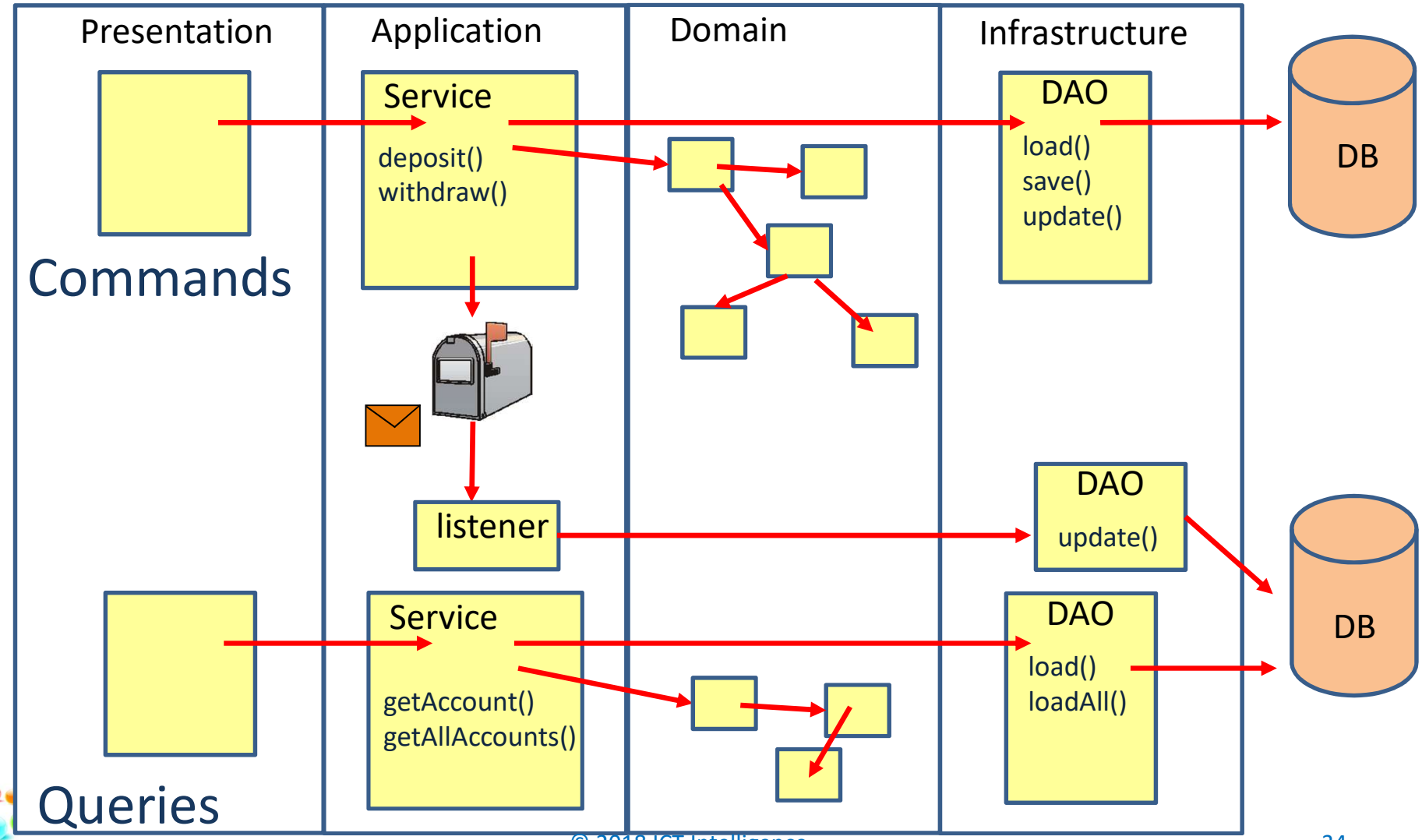
CQRS

- Domain model is used for commands
- View model is used for queries



Eventual consistency

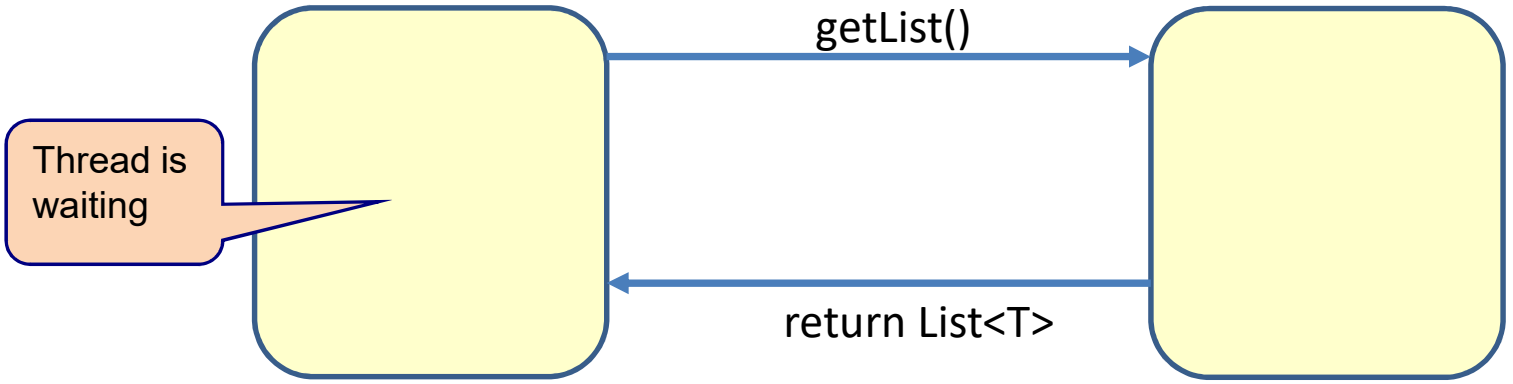
- Views will become eventual consistent



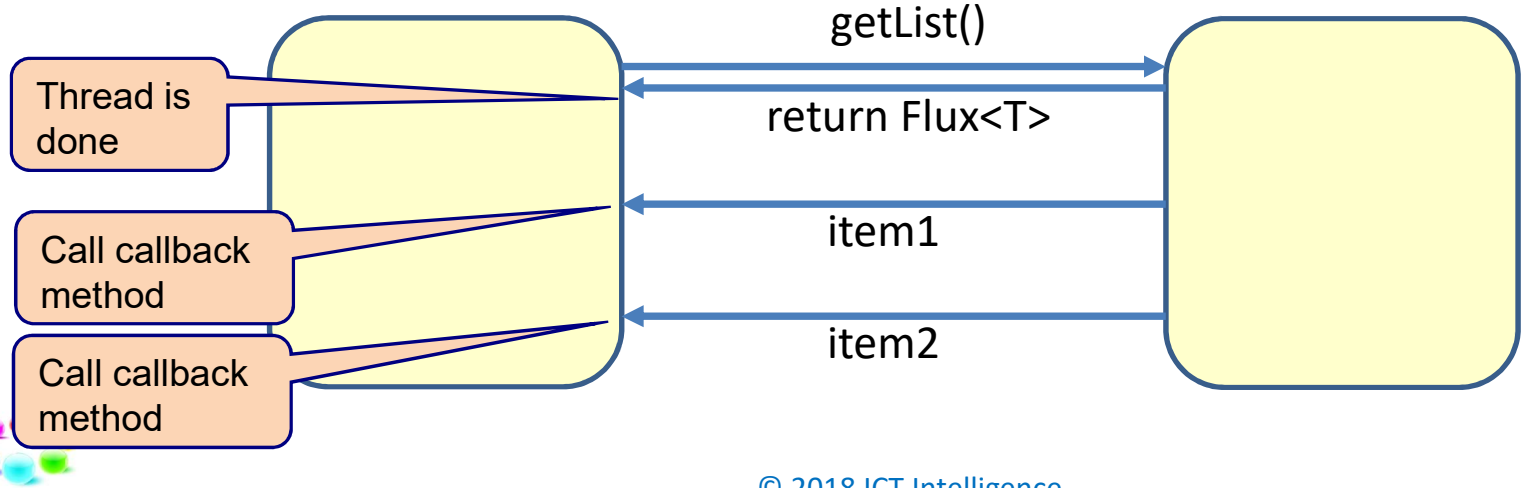
REACTIVE REST WITH SPRING WEBFLUX

Imperative versus reactive

- Synchronous, blocking

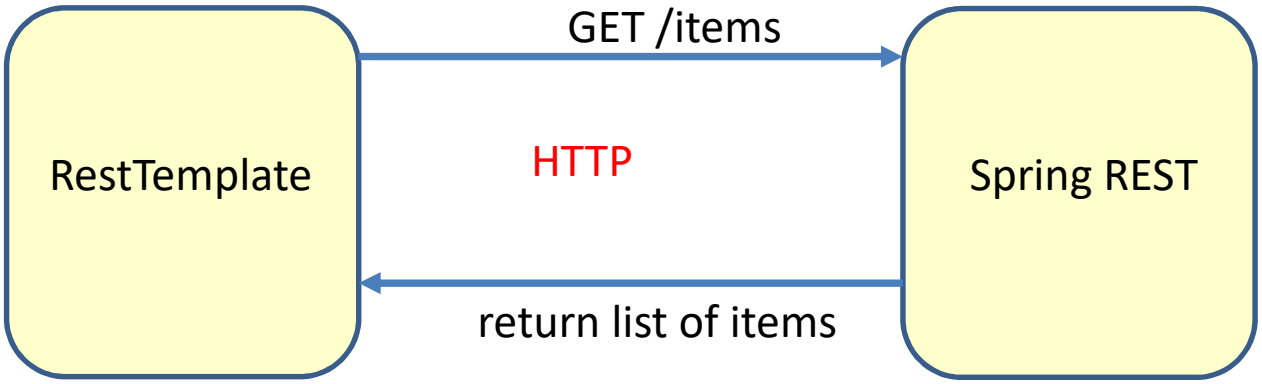


- Asynchronous, non-blocking

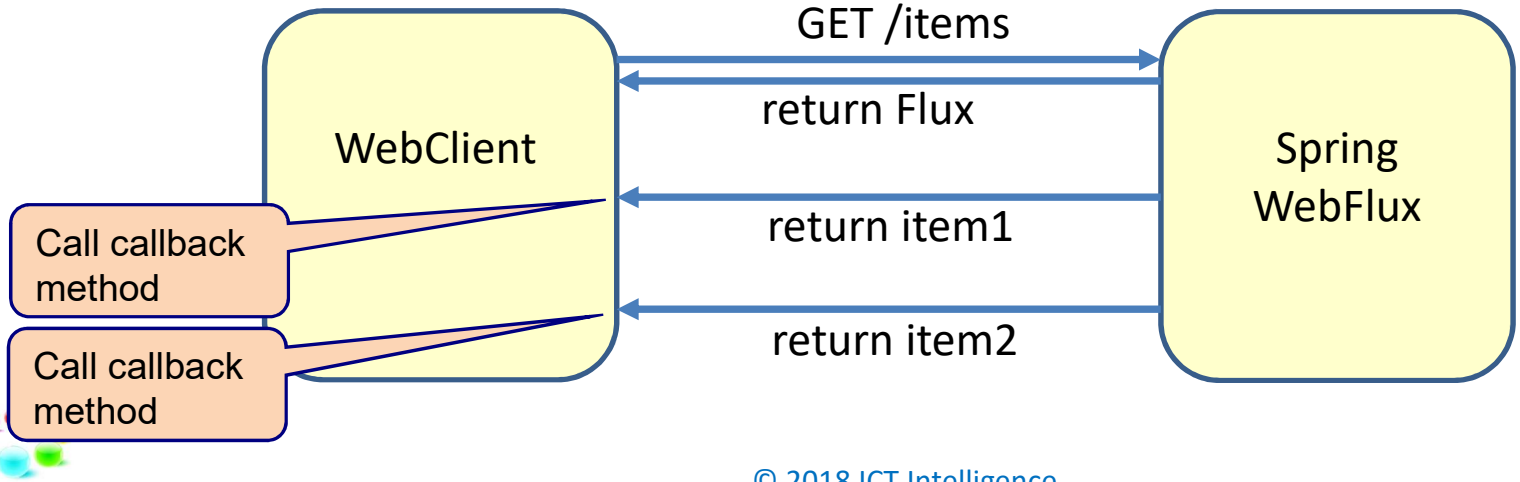


Reactive Web

- Synchronous, blocking



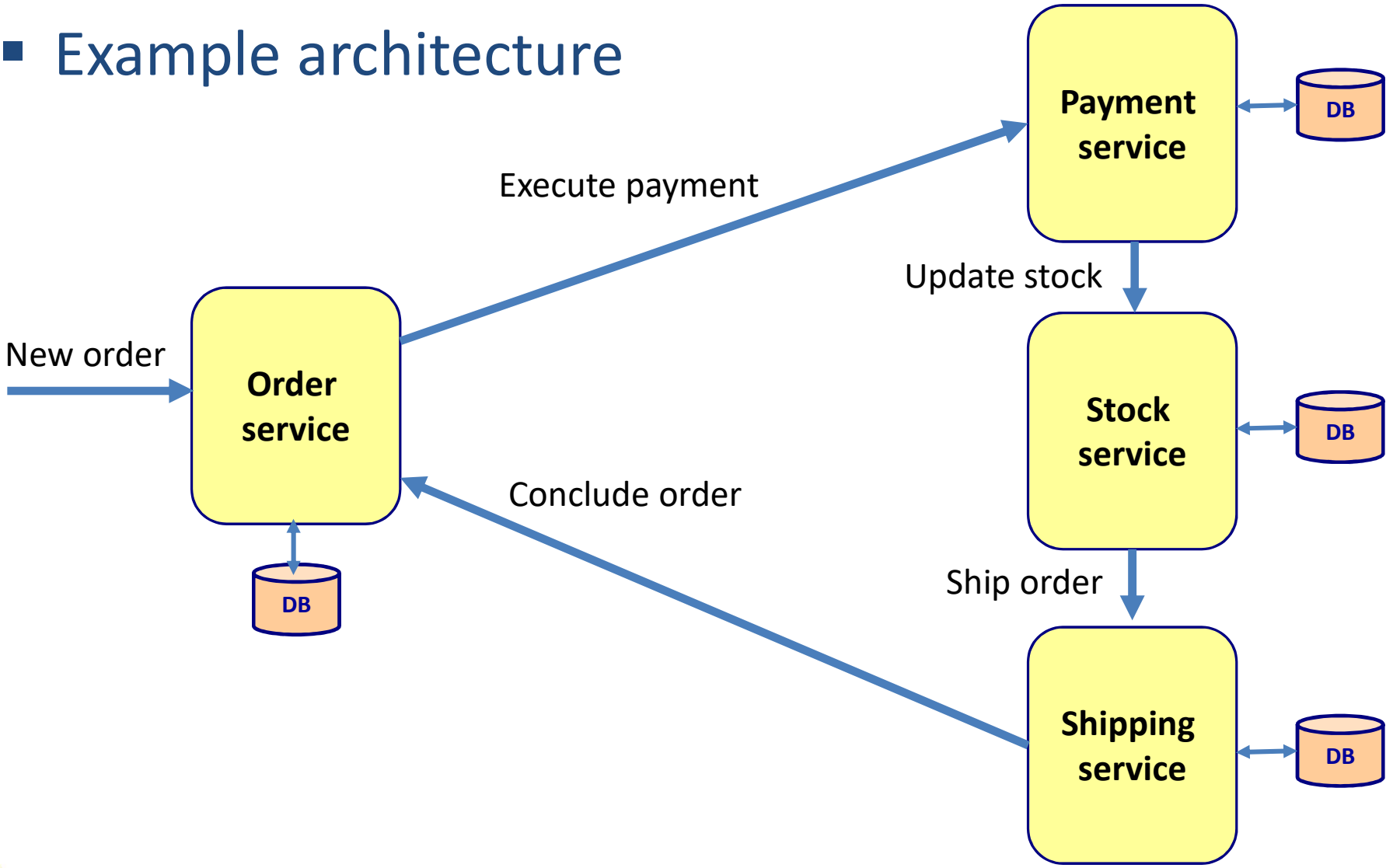
- Asynchronous, non-blocking



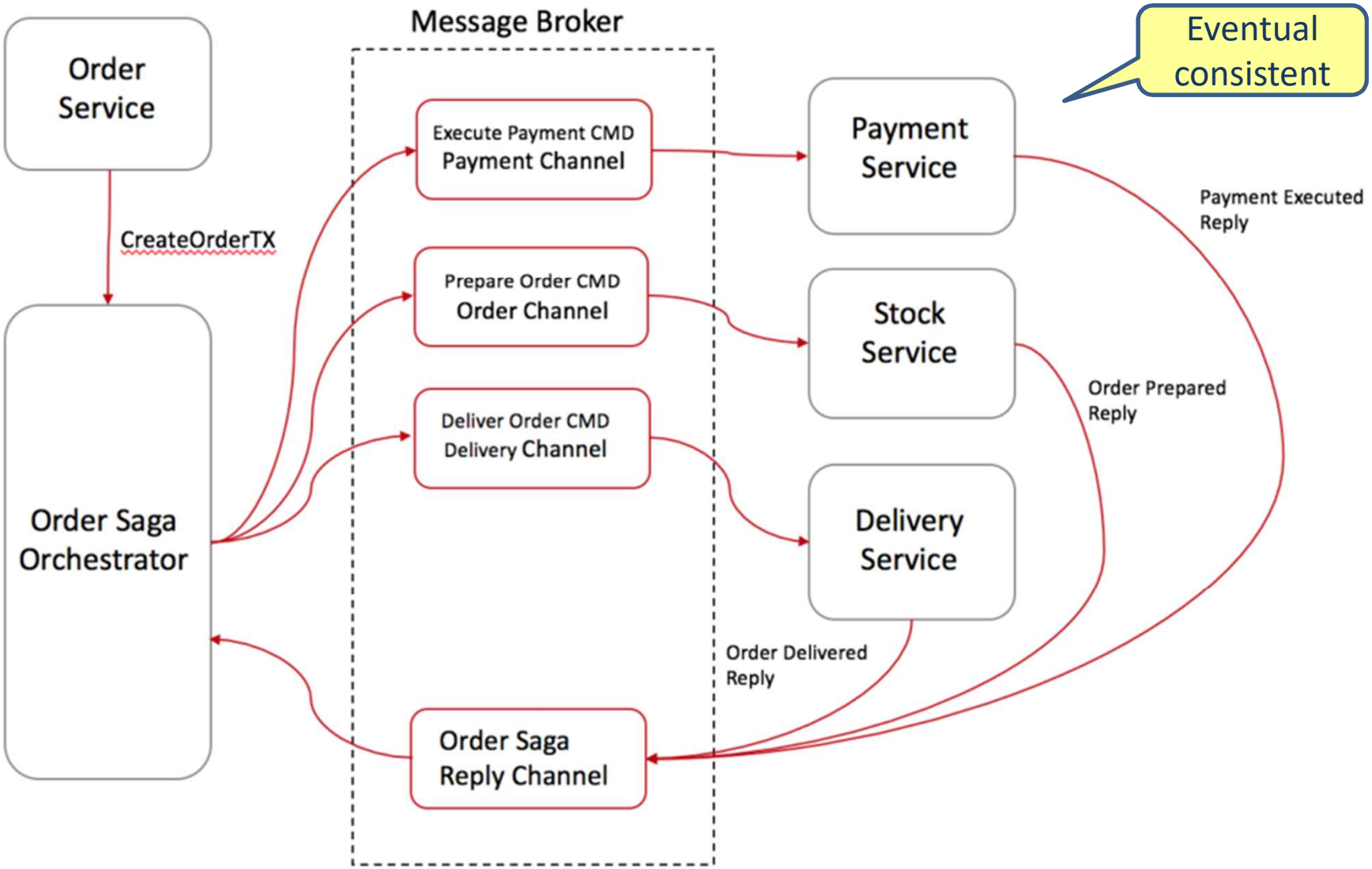
TRANSACTIONS

Saga pattern

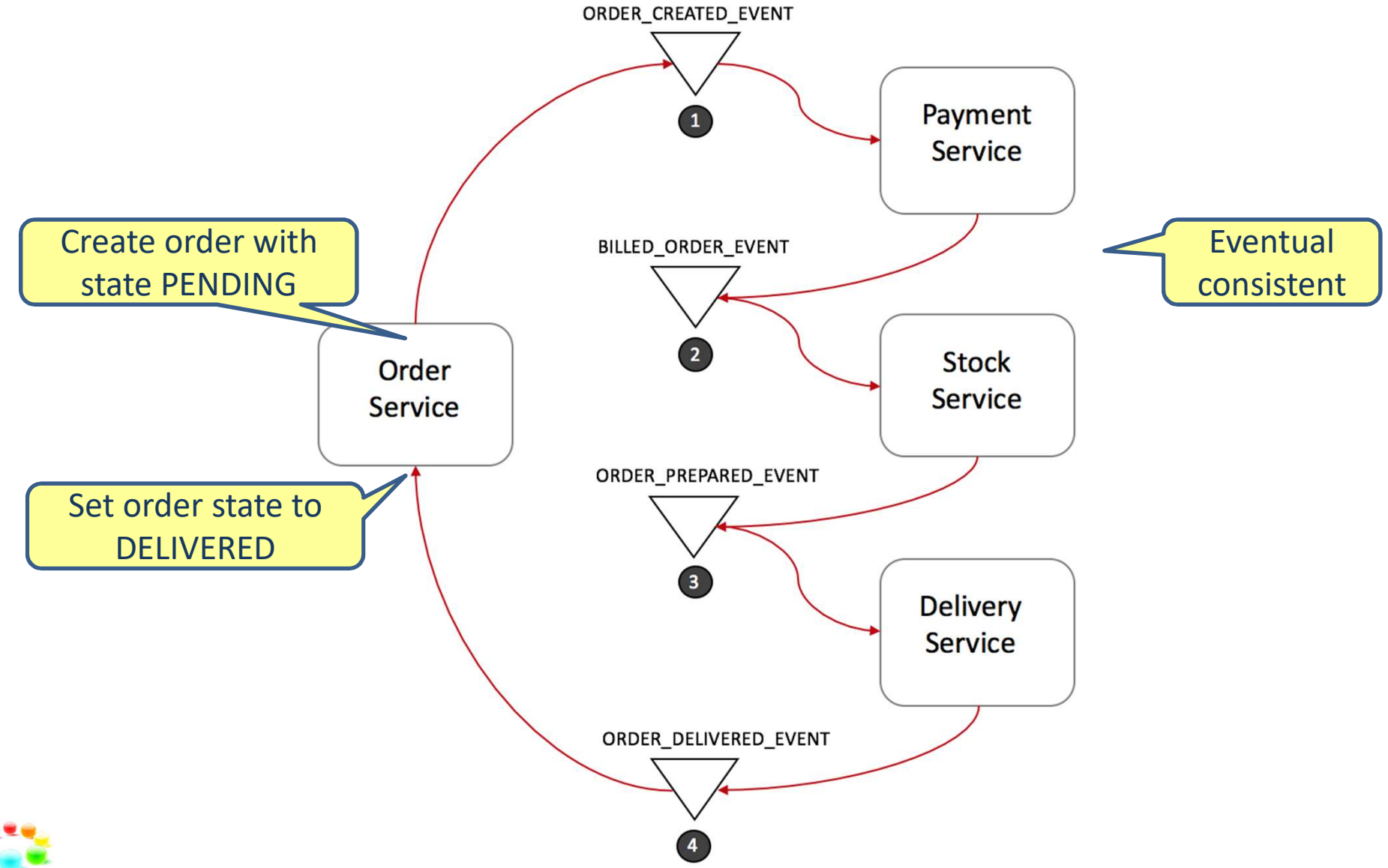
- Example architecture



Saga with command/orchestration

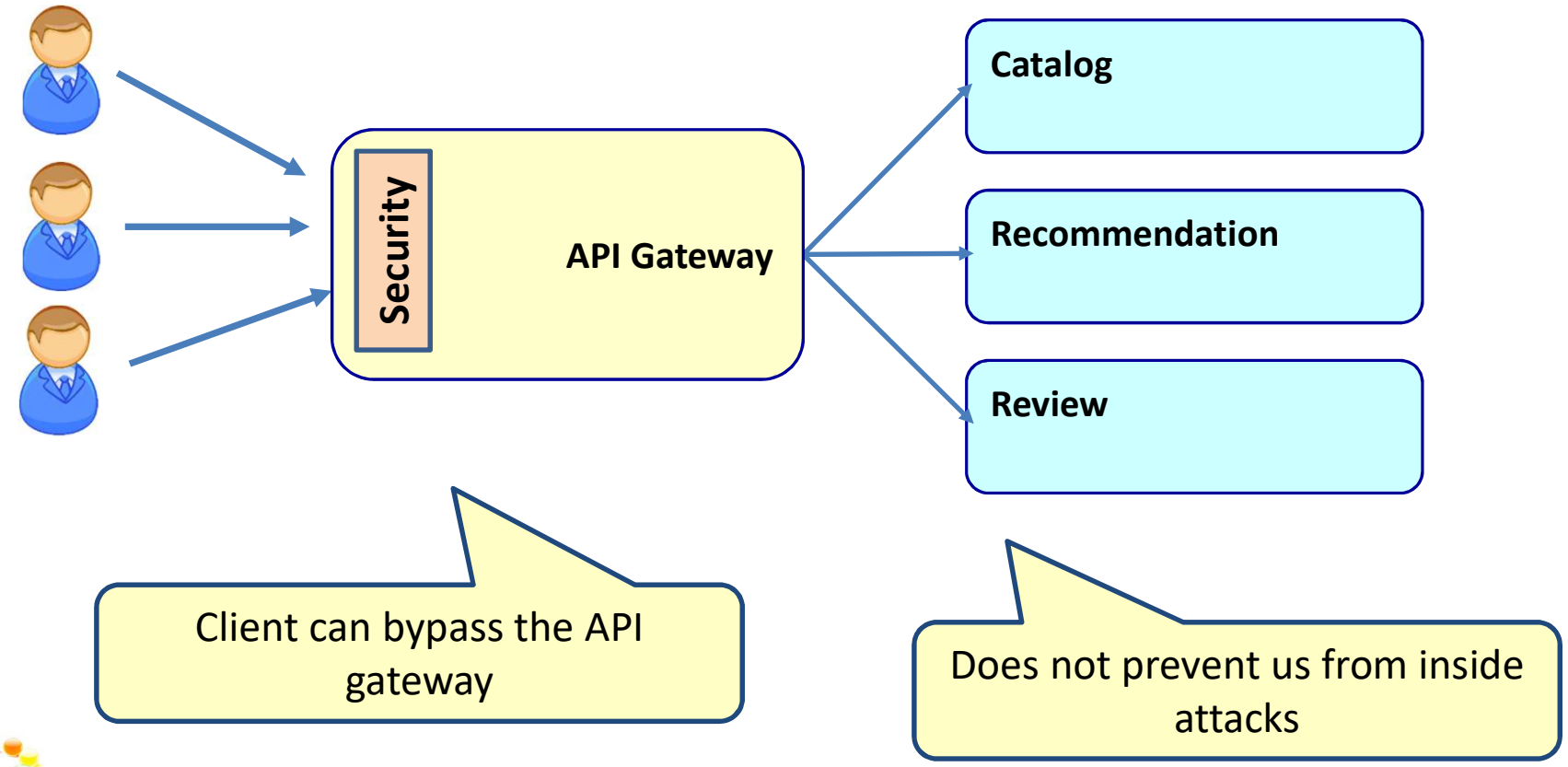


Saga with events/choreography

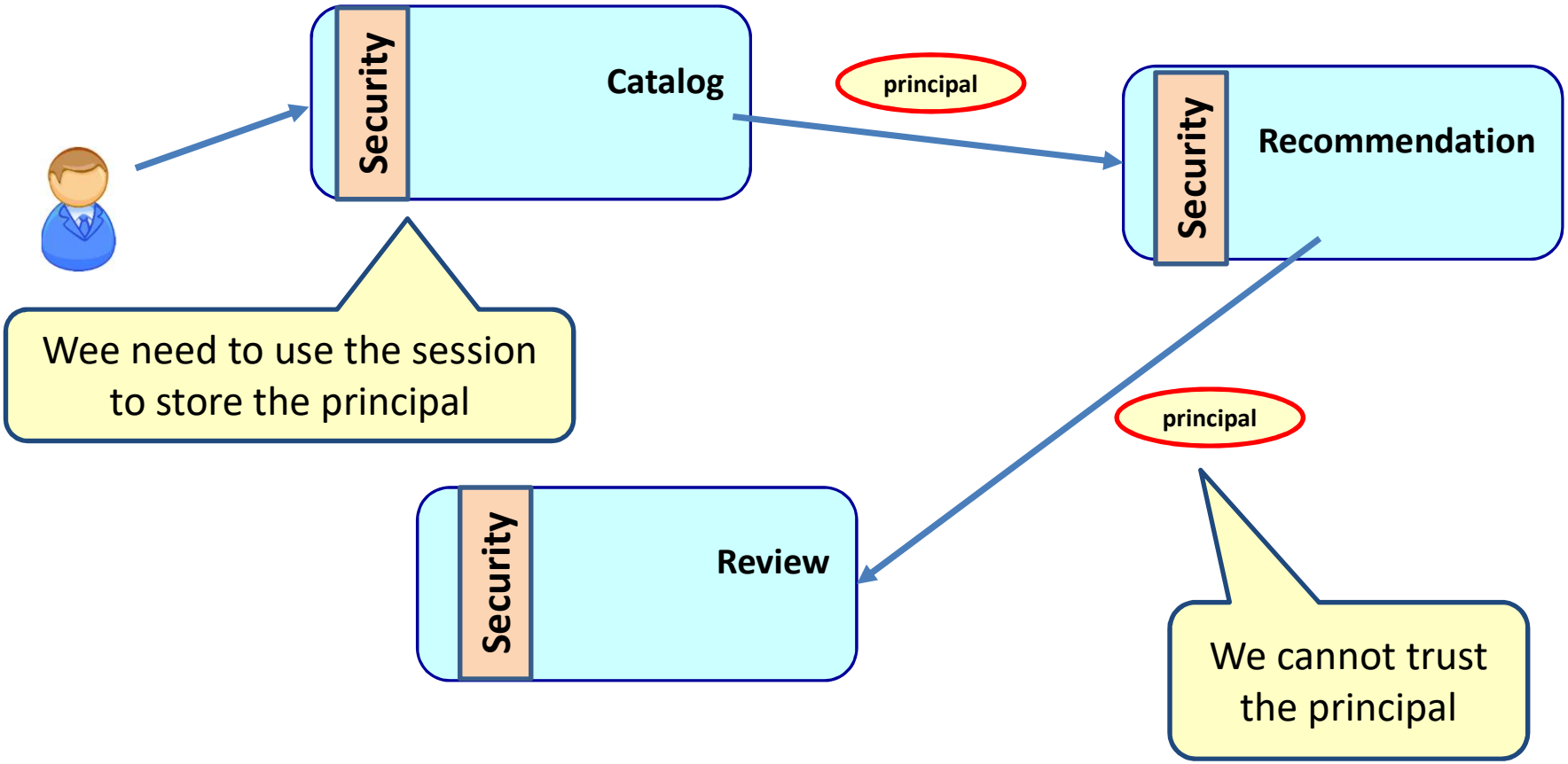


SECURE THE MICROSERVICE ARCHITECTURE

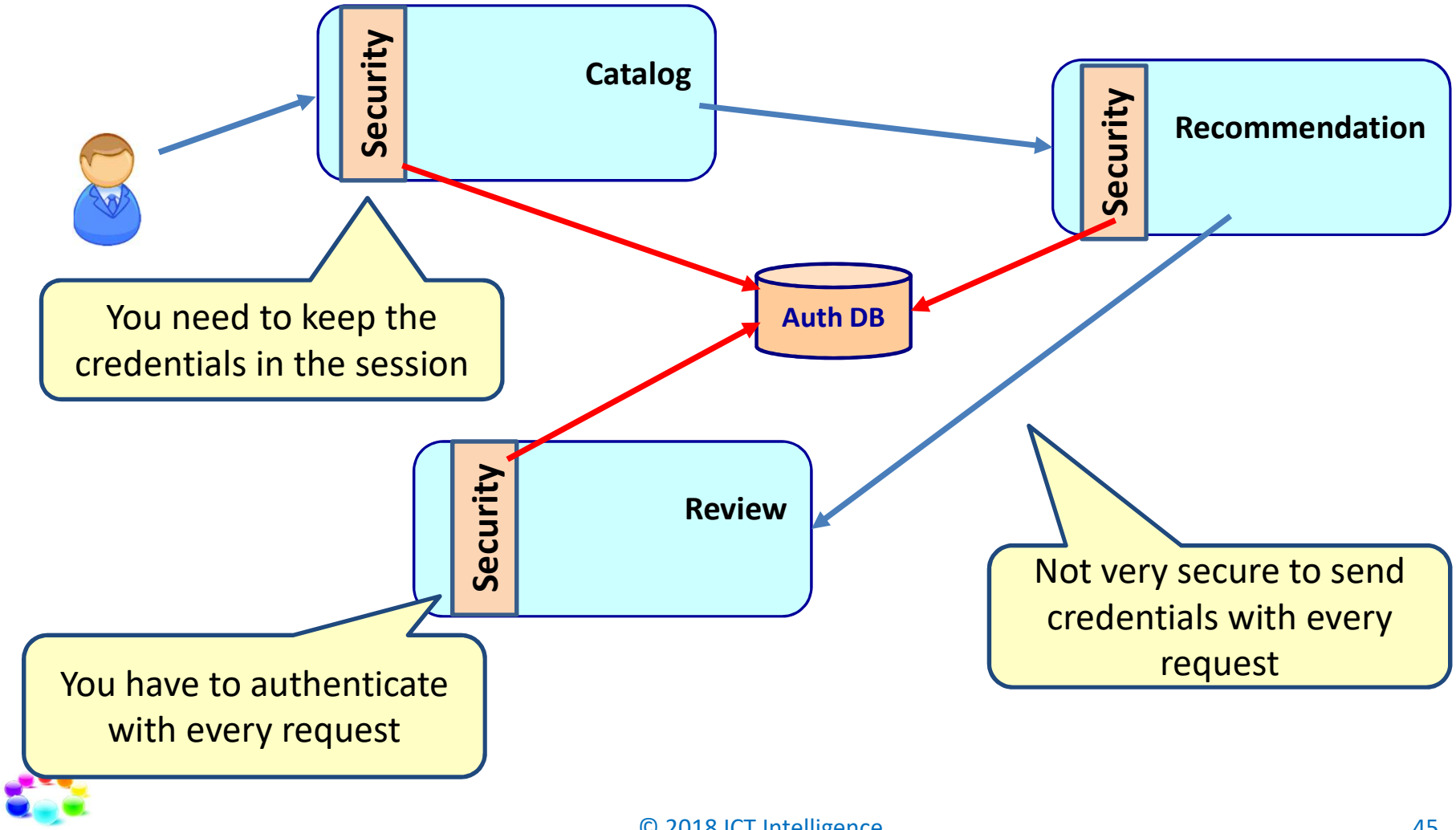
Secure the API gateway



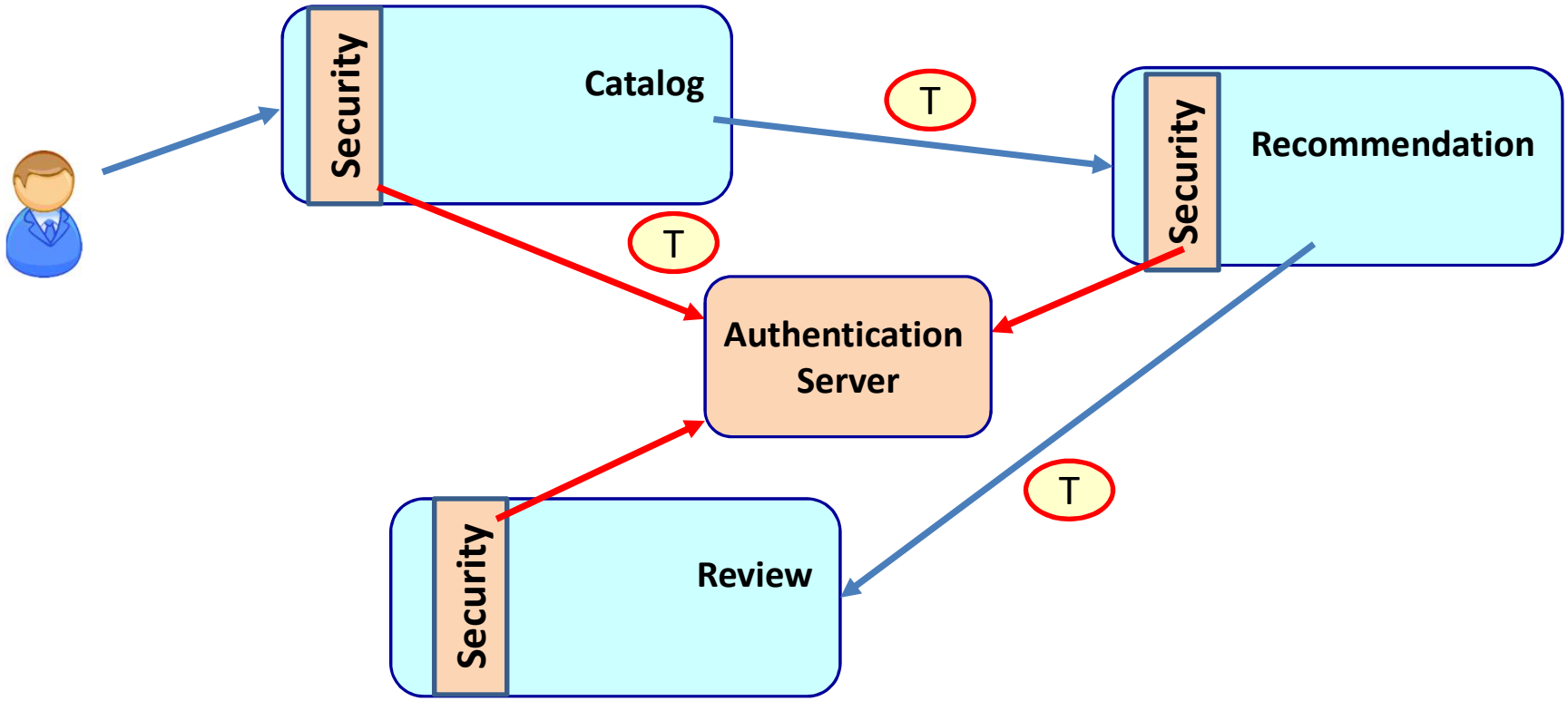
Send principal with every request



Send userid/password with every request

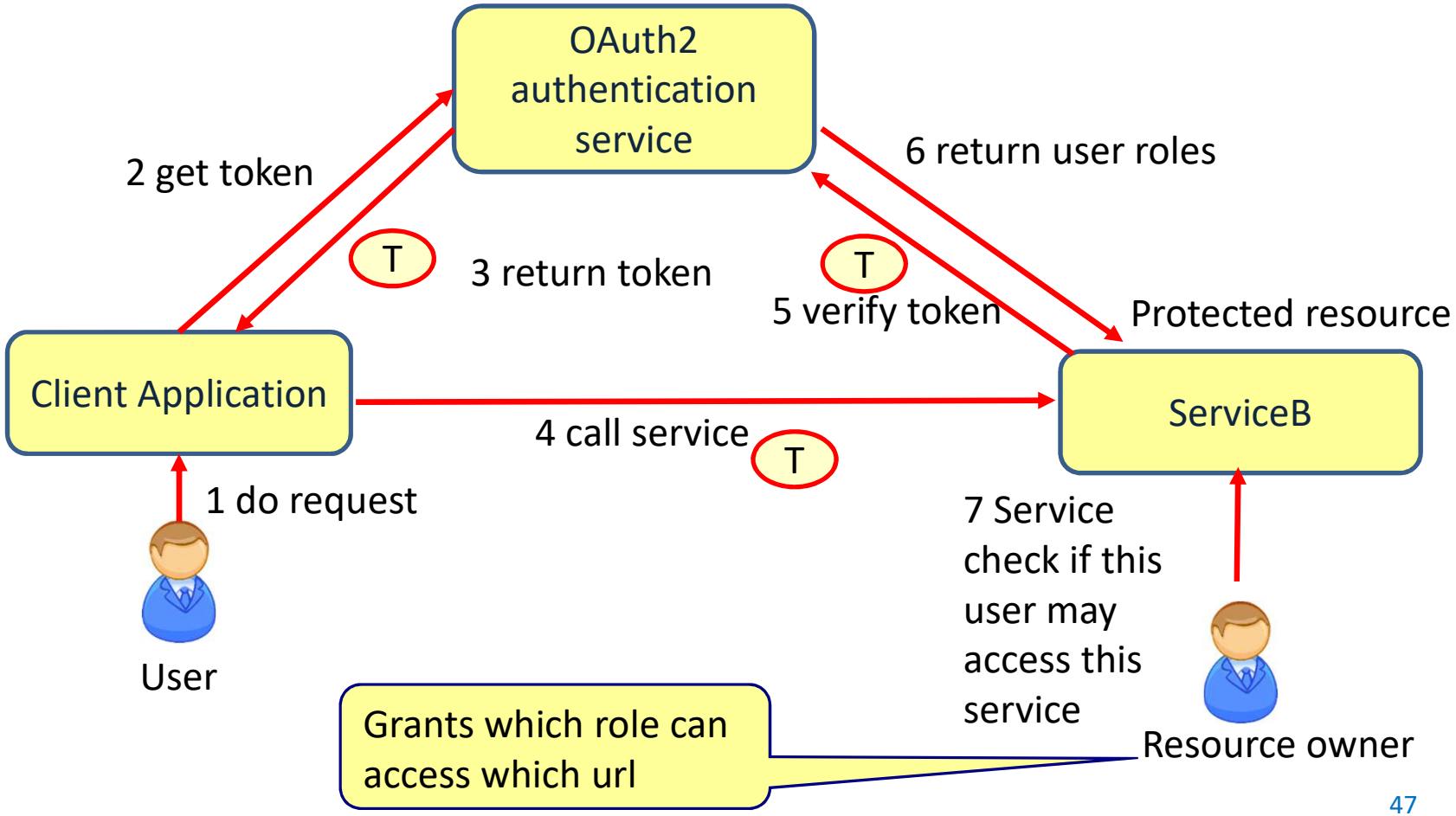


OAuth2: Token based security

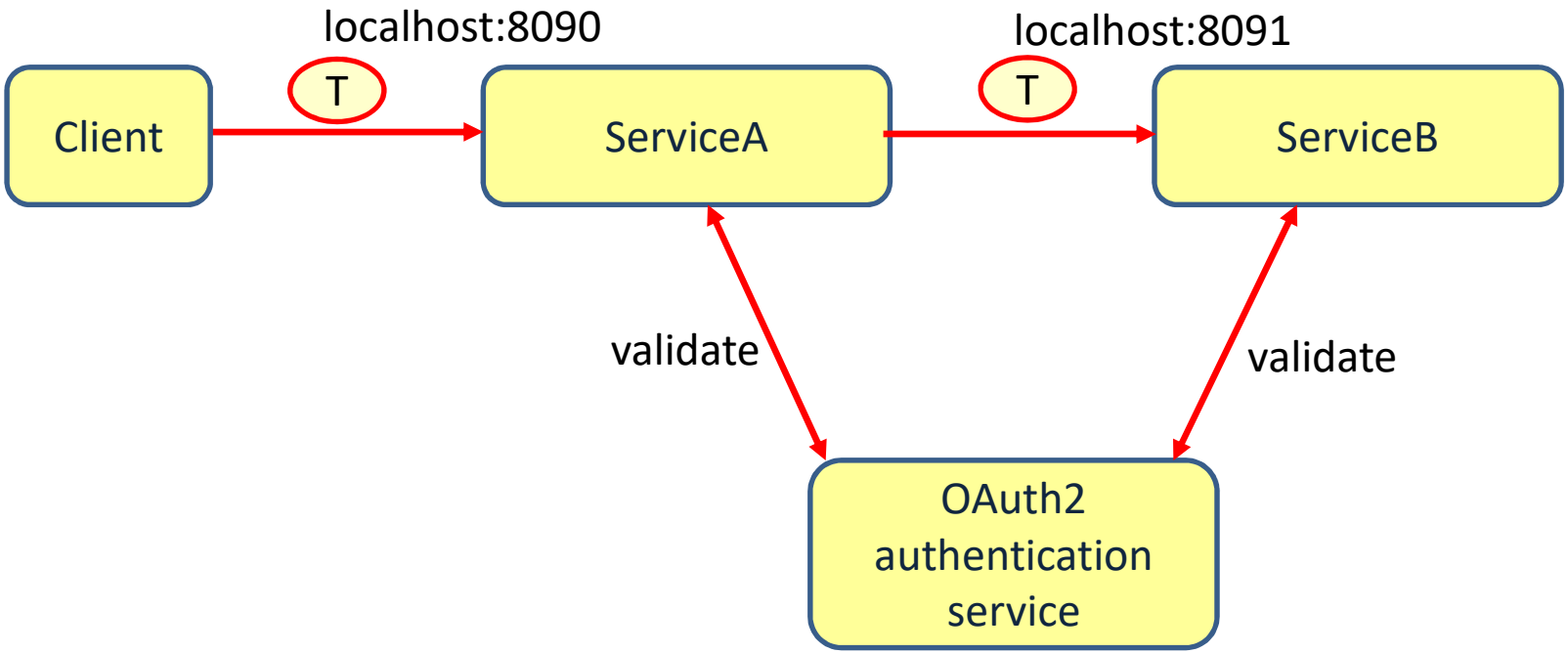


How does OAuth2 work

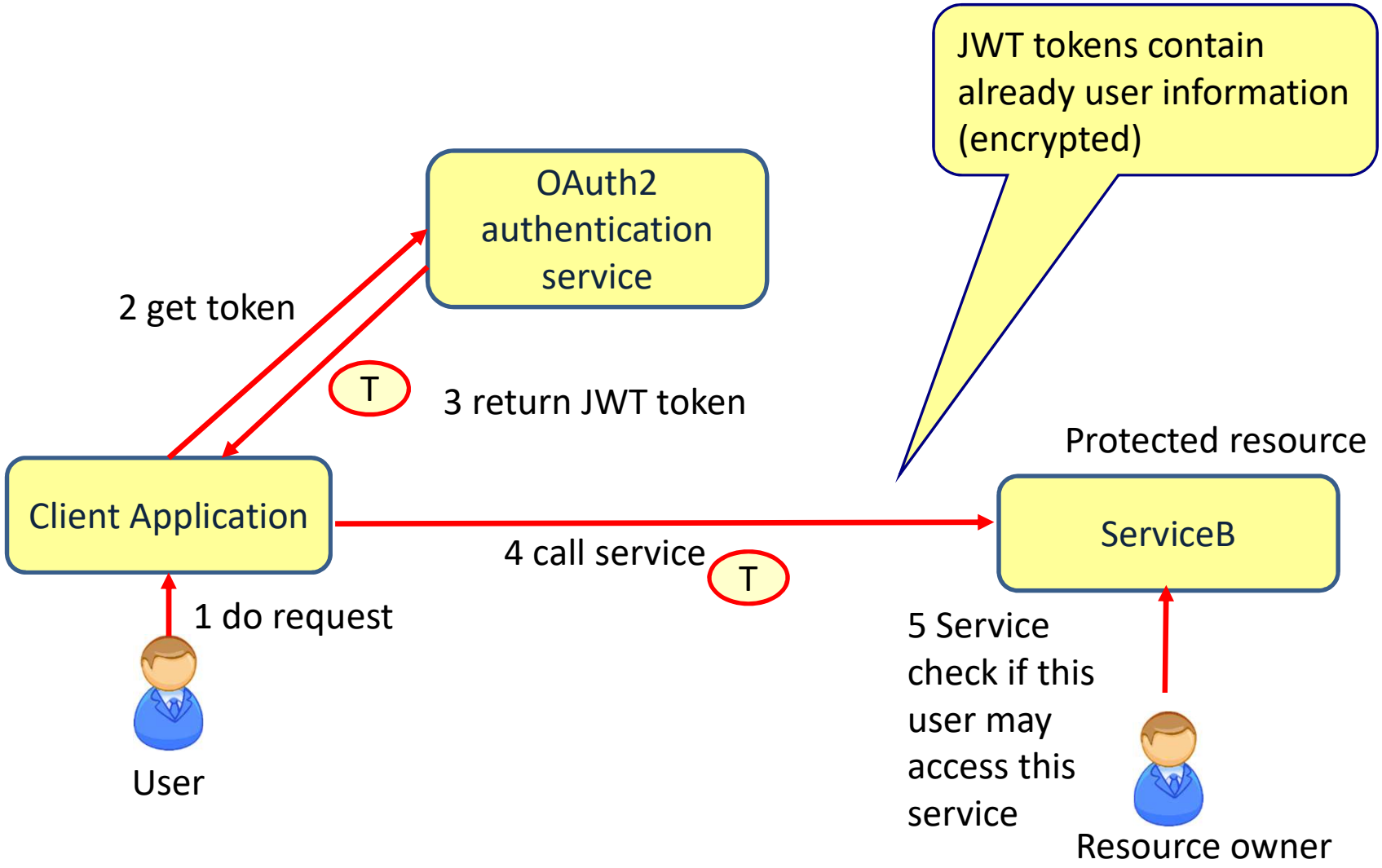
- Token based authentication and authorization framework



Propagate the token



JWT tokens

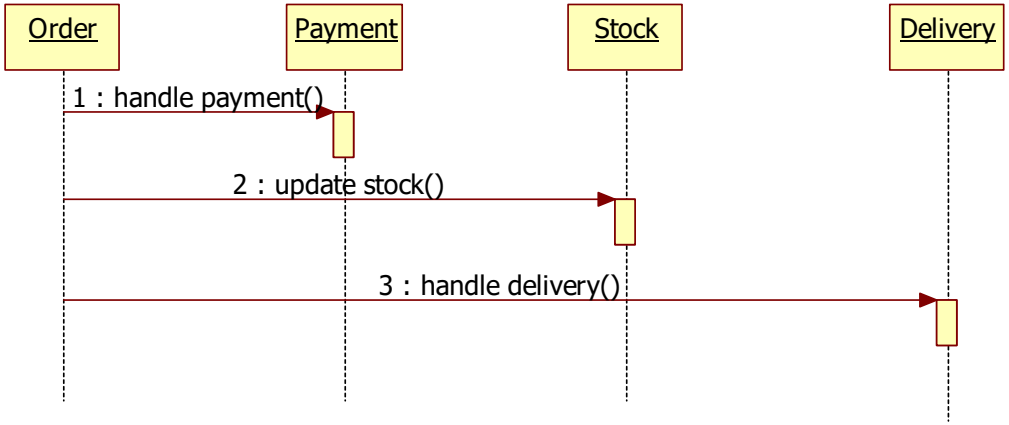


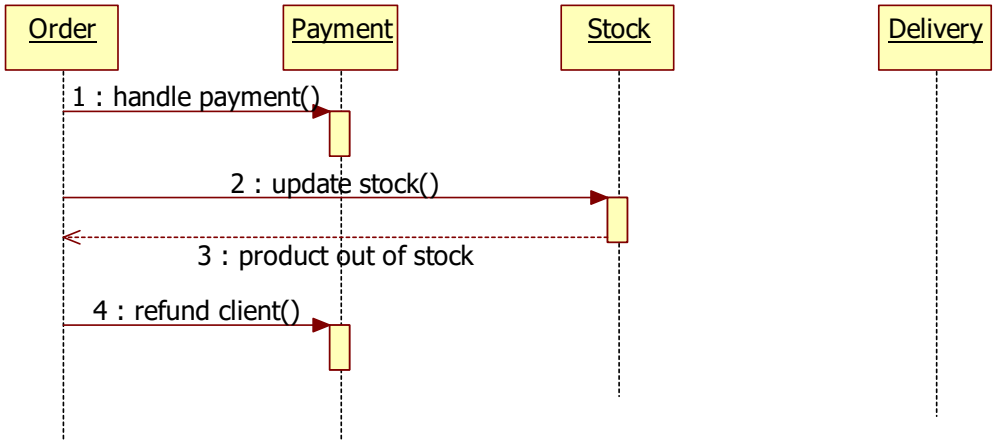
EVENT DRIVEN ARCHITECTURE

Event driven architecture

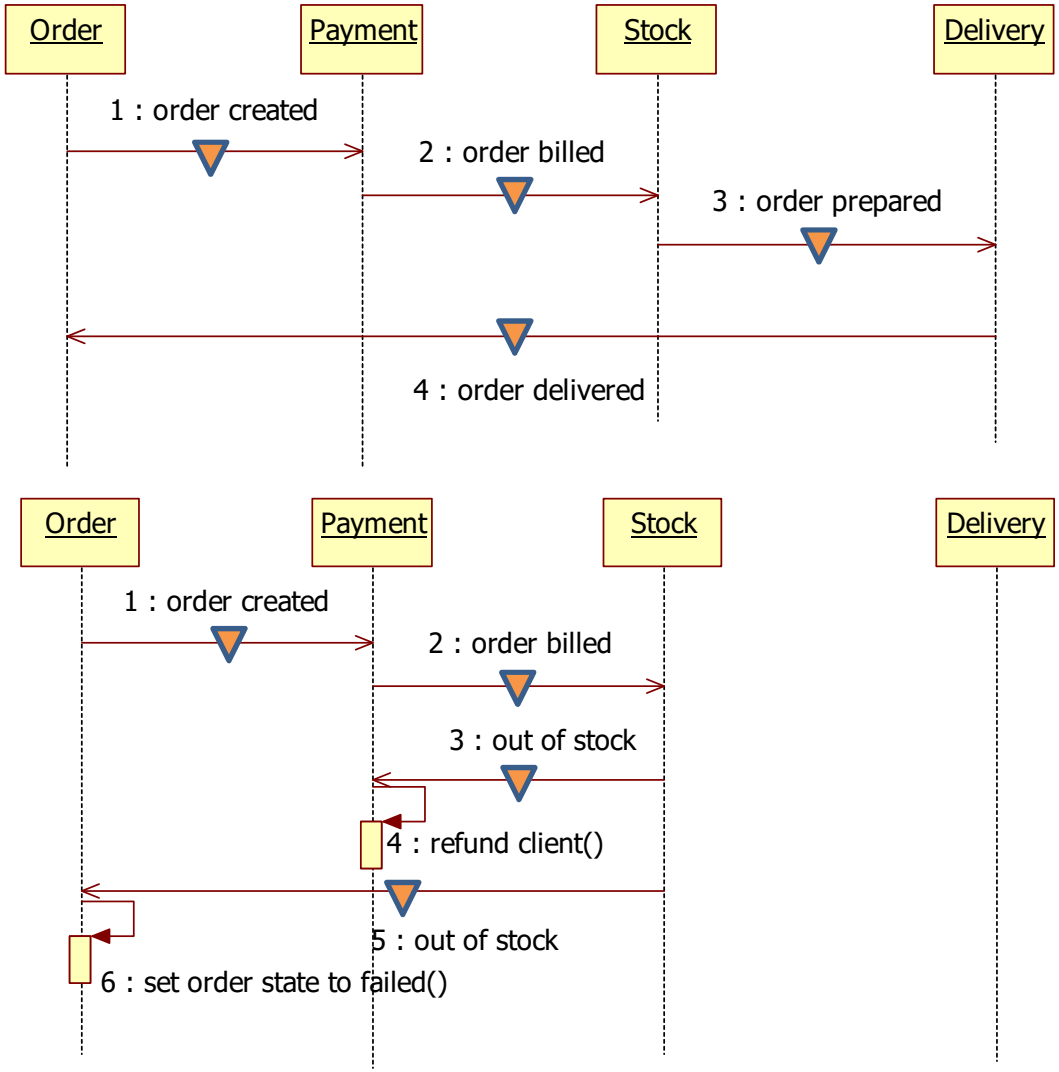
- Loosely coupled services
- Asynchronous
  - No blocking calls
  - No threads that are just waiting
- Flexible
  - Publish-subscribe
    - Easy to add new publishers
    - Easy to add new listeners
- Buffer
  - If a service is slow or down

Synchronous (REST) calls

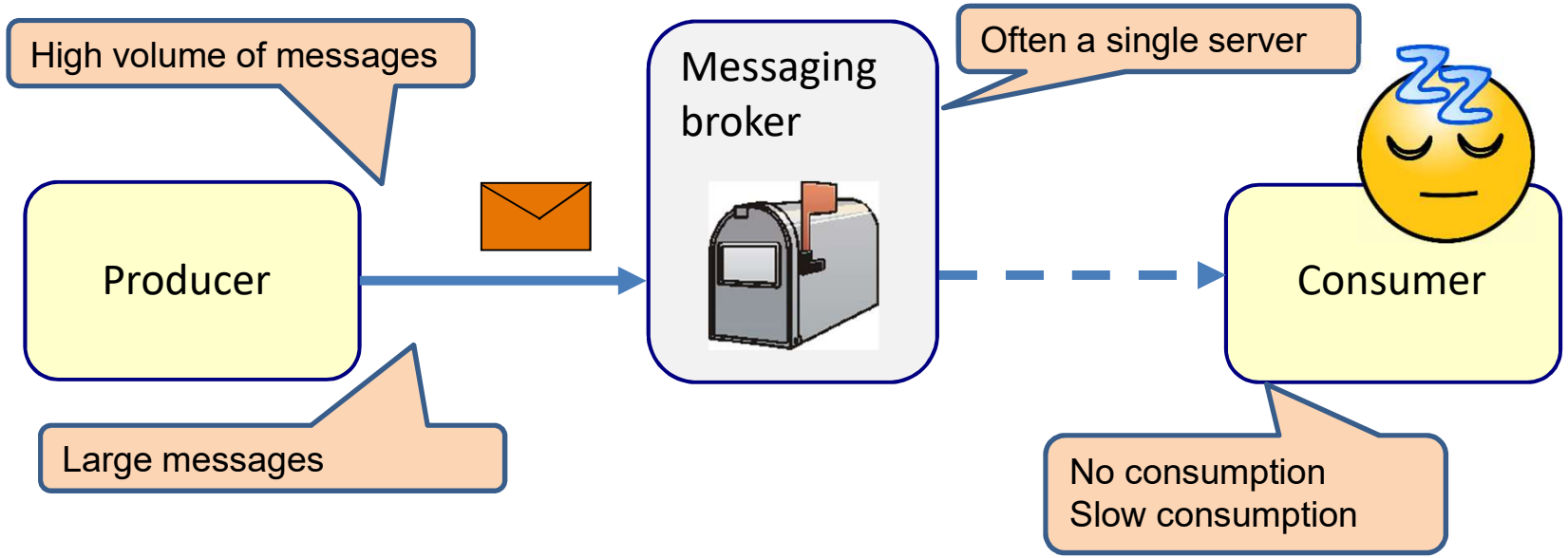




Asynchronous events (messaging)

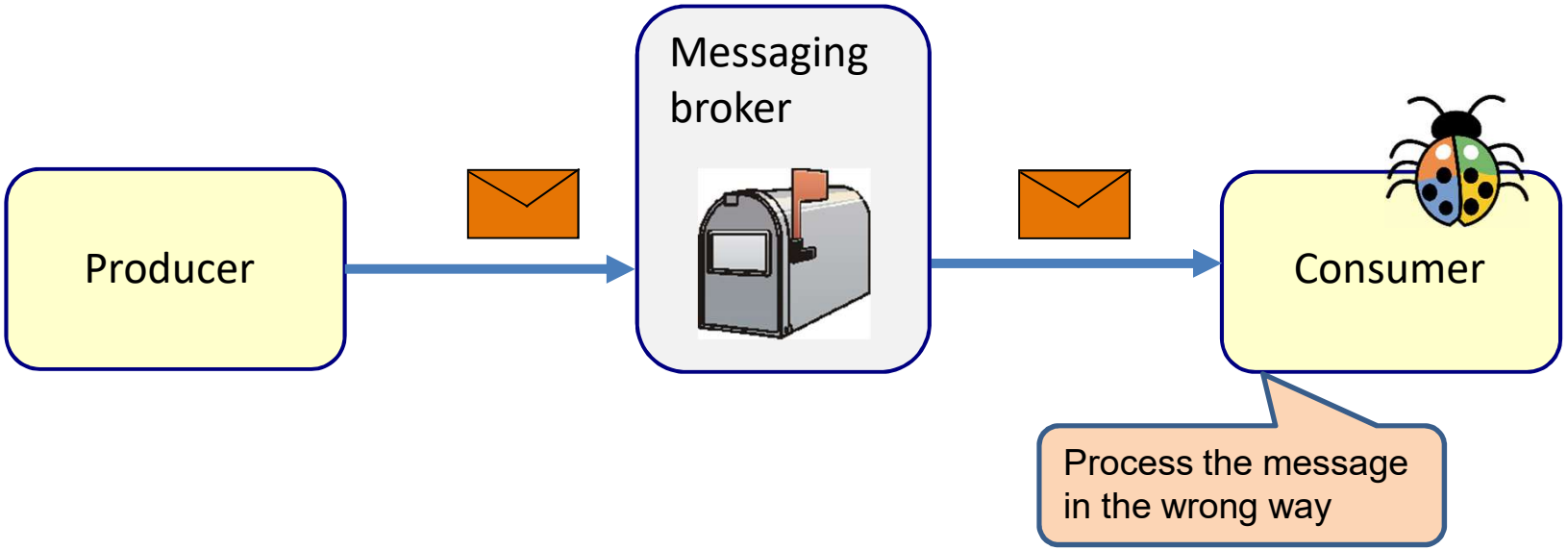


Problems with traditional messaging middleware



- If the consumer is temporally not available (or very slow) the message middleware has to store the messages
  - This restricts the volume of messages and the size of the messages
  - Eventually the message broker will fail

Problems with traditional messaging middleware



- If the consumer has a bug, and handles the messages incorrectly, then the messages are gone.
  - Not fault-tolerant

Apache Kafka

- Created by Linked In
- Characteristics
  - High throughput
  - Distributed
  - Unlimited scalable
  - Fault-tolerant
    - Reliable and durable
  - Loosely coupled Producers and Consumers
  - Flexible publish-subscribe semantics

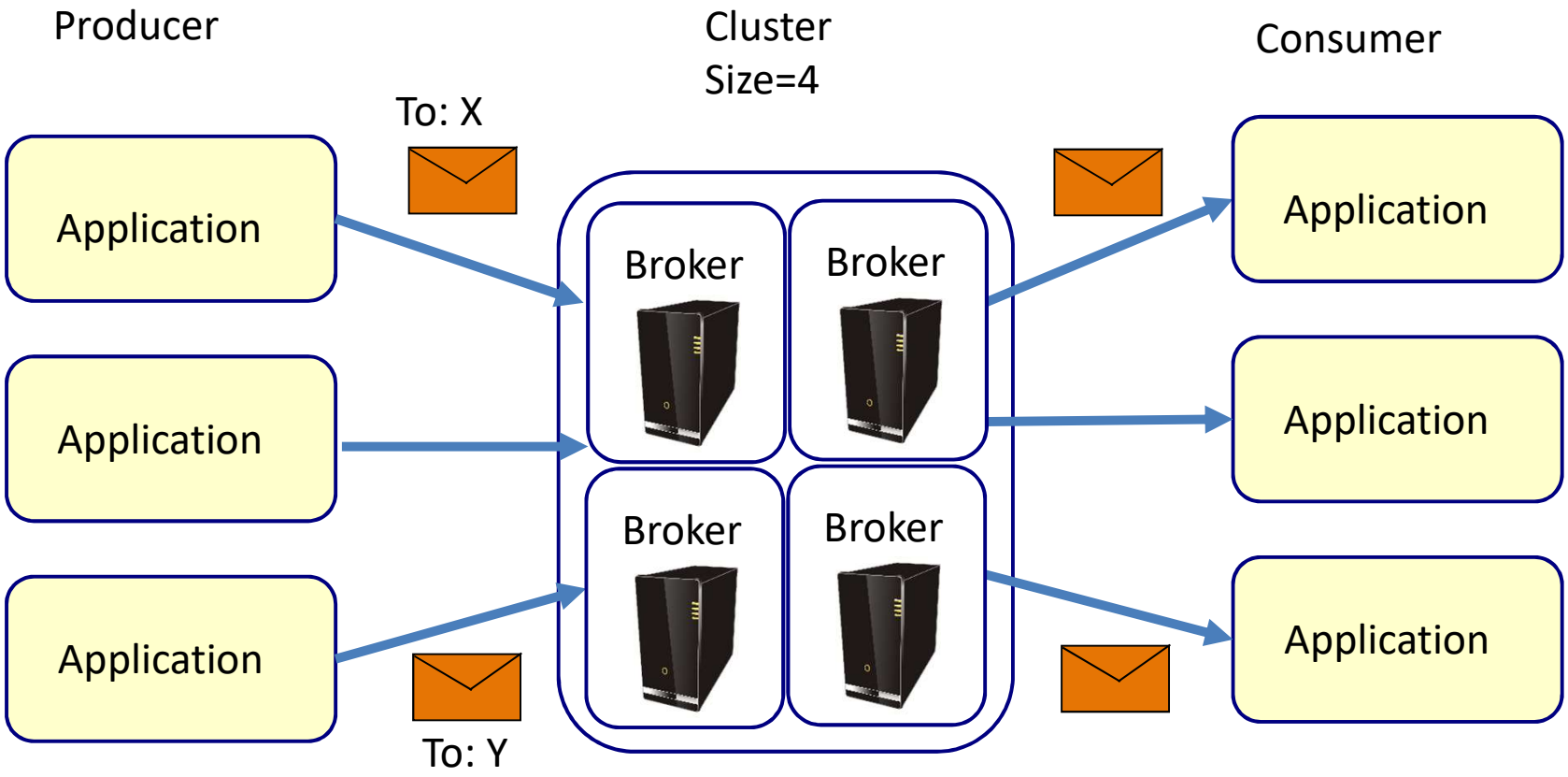
High Volume:

- Over 1.4 trillion messages per day
- 175 terabytes per day

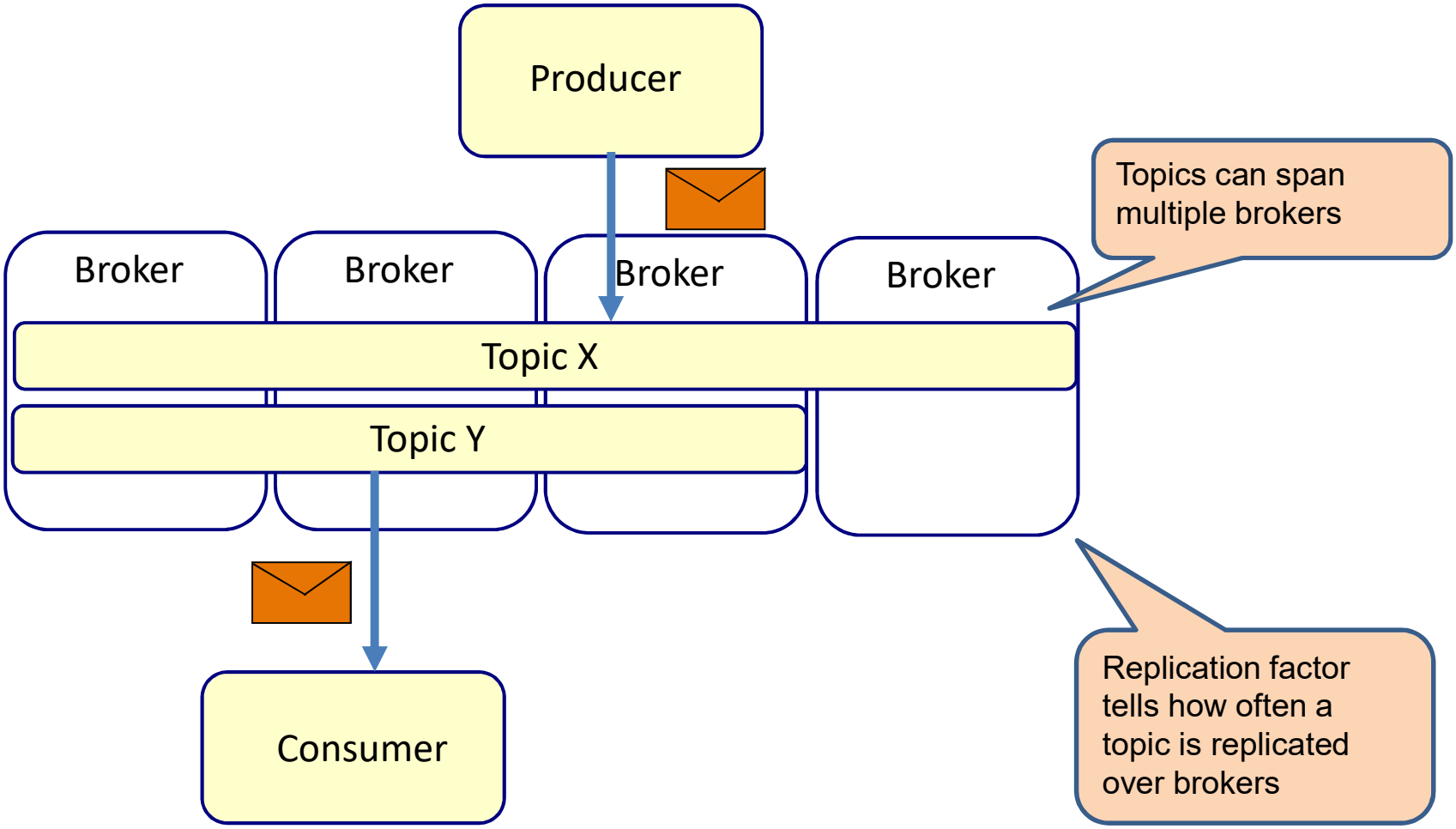
High Velocity:

- Peak 13 million messages per second
- 2.75 gigabytes per second

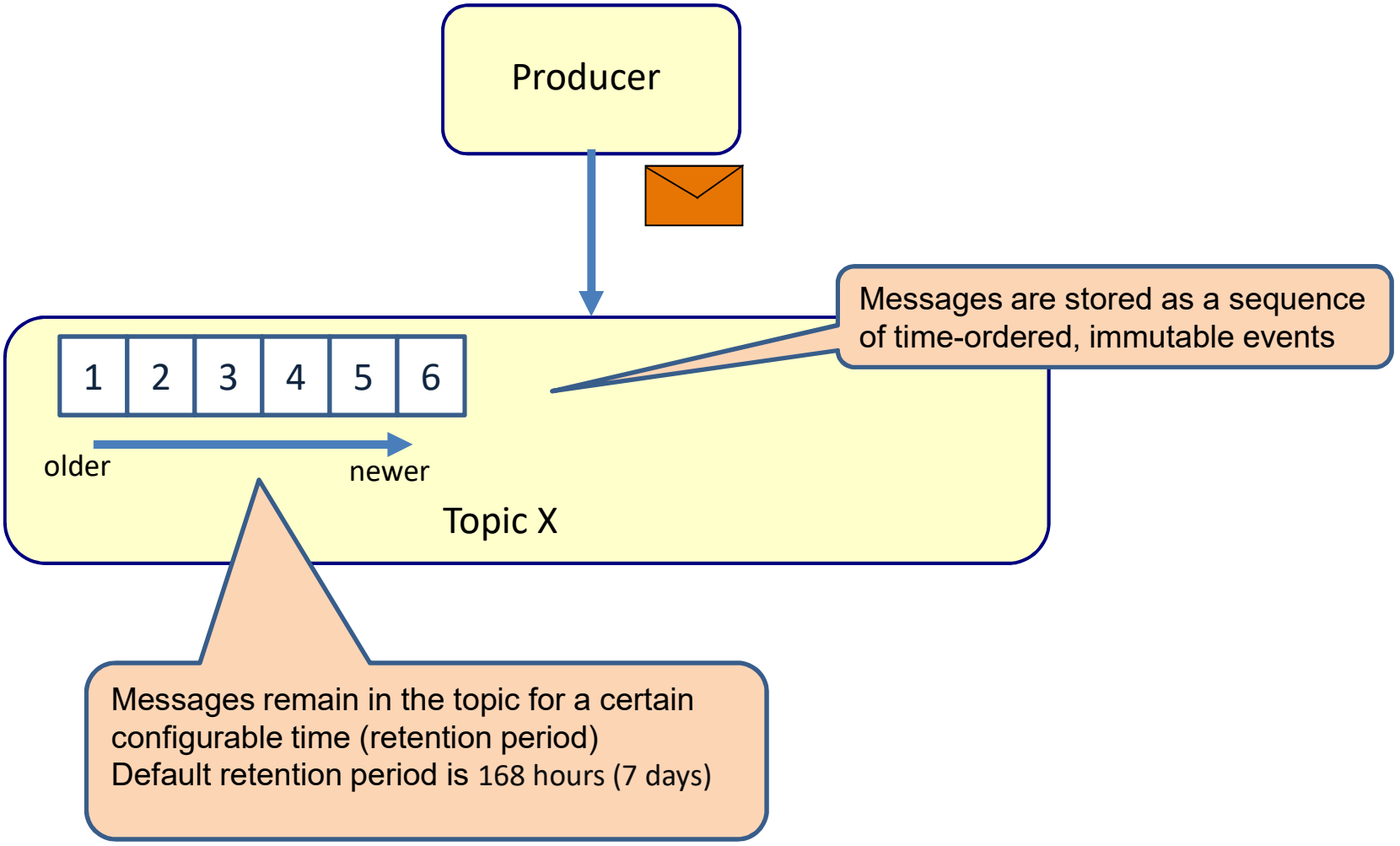
Cluster of Brokers



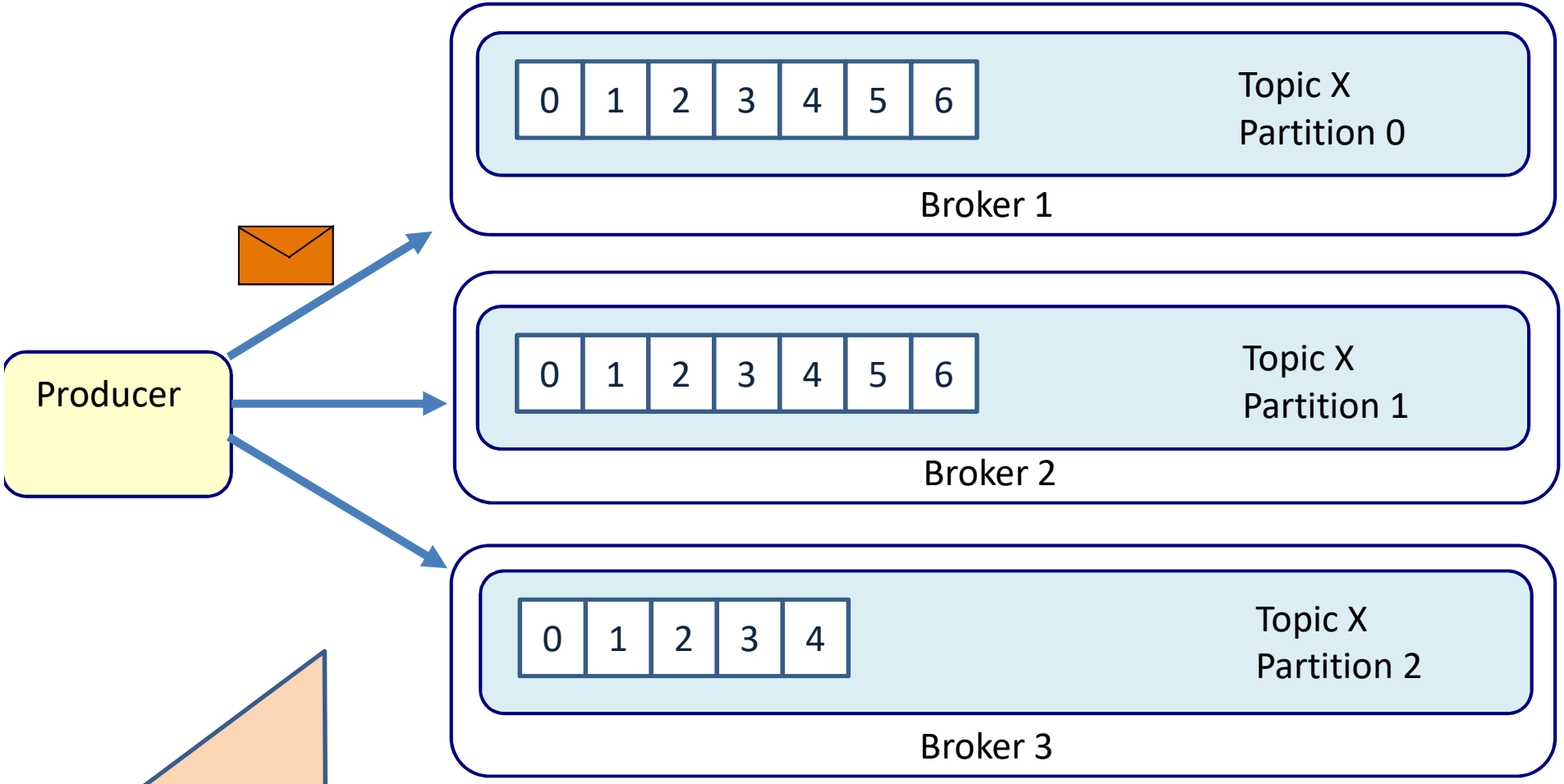
Topics


Producer
Broker
Broker
Broker
Broker
Topic X
Topic Y
Consumer
Topics can span multiple brokers
Replication factor tells how often a topic is replicated over brokers

Event sourcing



Scale out partitions

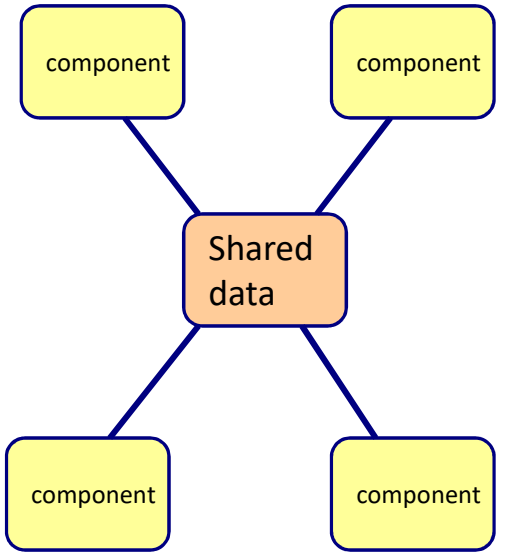


Increases
- Scalability (number of messages and their size)
- Higher levels of throughput (parallelism)

STREAM BASED ARCHITECTURE

Blackboard

- Common data structure
  - Extension is no problem
  - Change is difficult
- Easy to add new components
- Tight coupling for data structure
- Loose coupling for
  - Location
  - Time
  - Technology(?)
- Synchronisation issues

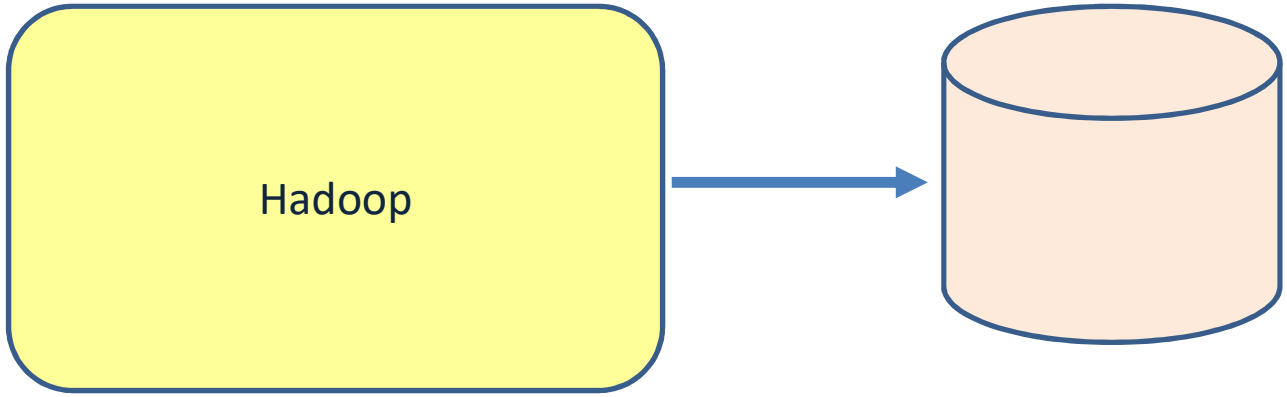


Event sourcing

- Instead of storing the state of an entity in a database, you store the series of events that lead up to the state.
- Storing all of the events increases the analytical capabilities of a business.
- Instead of just asking what the current state of an entity is, a business can ask what the state was at any time in the past

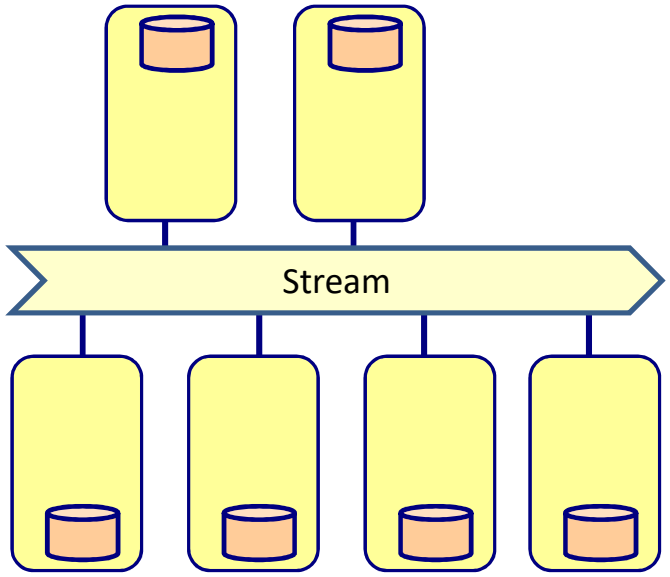
Batch processing

- First store the data in the database
- Then do queries (map-reduce) on the data
- Queries over all or most of the data in the dataset.
- Latencies in minutes to hours

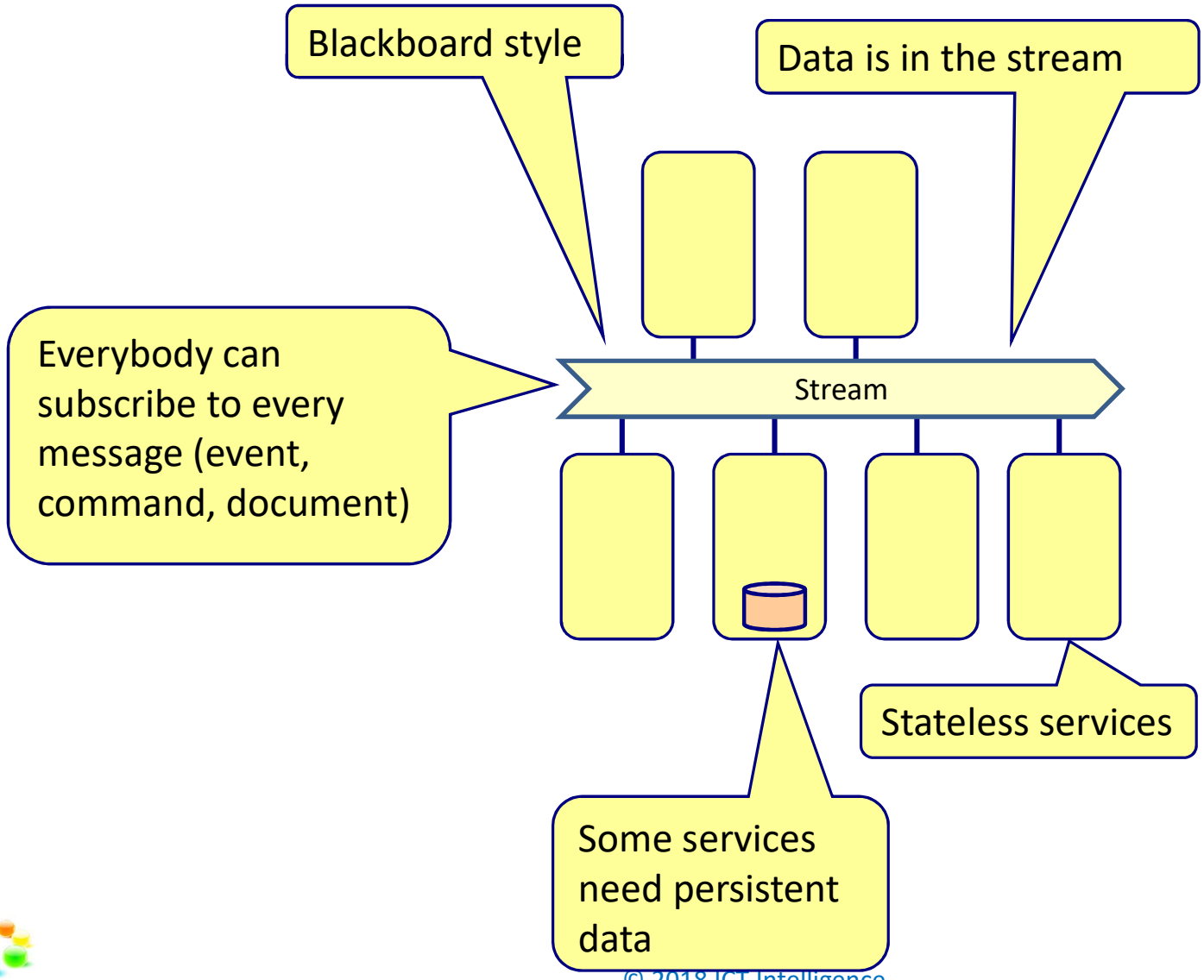


Stream processing

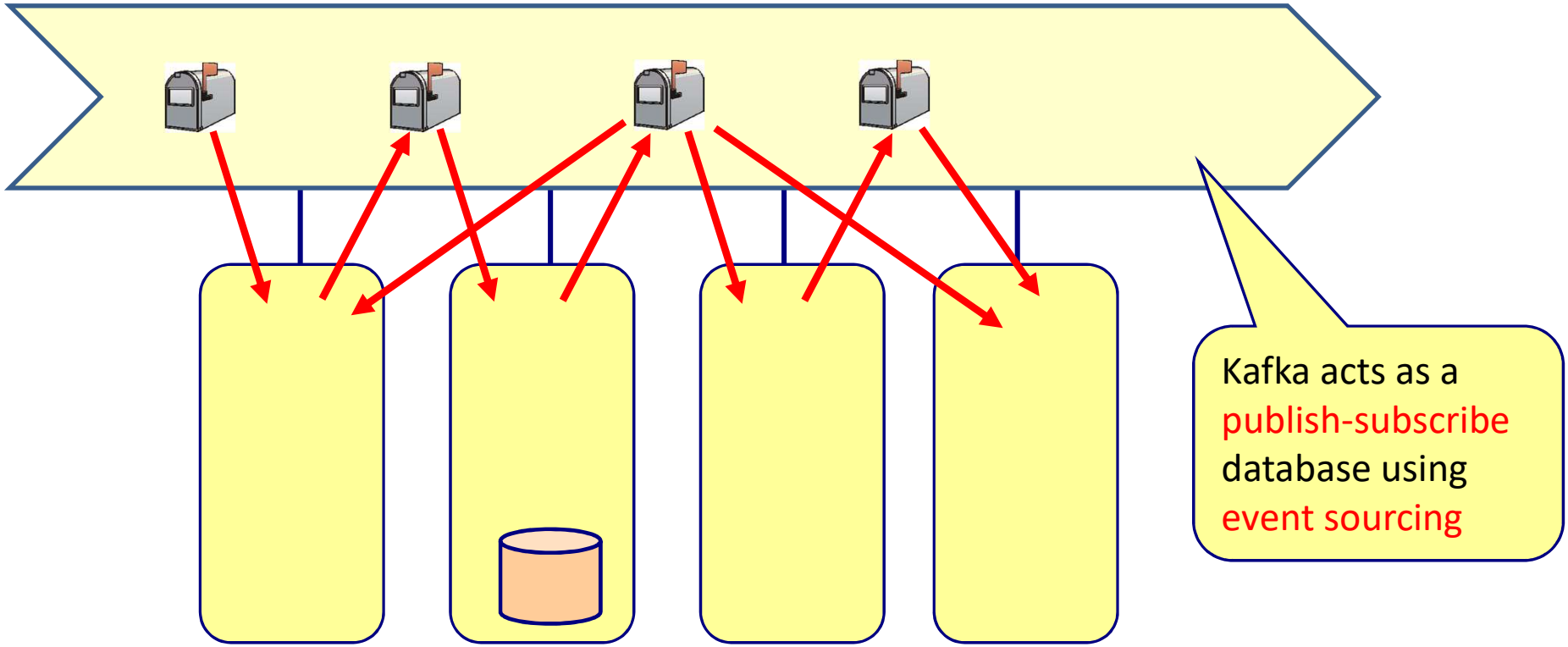
- Handle the data when it arrives
- Handle event (small data) by event
- Latencies in seconds or milliseconds



Where is the data?



Publish-subscribe and event sourcing



Stream based architecture