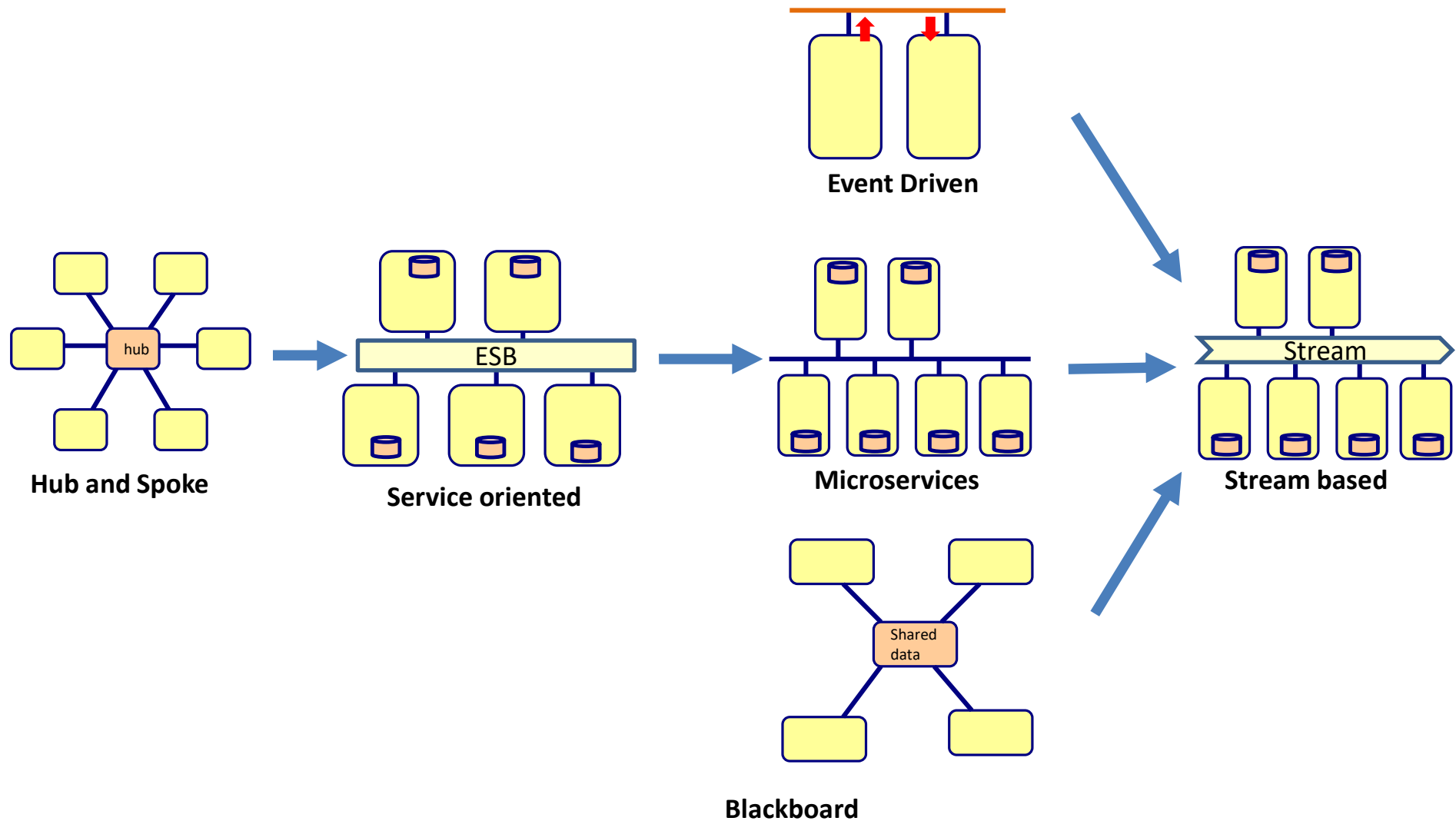


Lesson 12

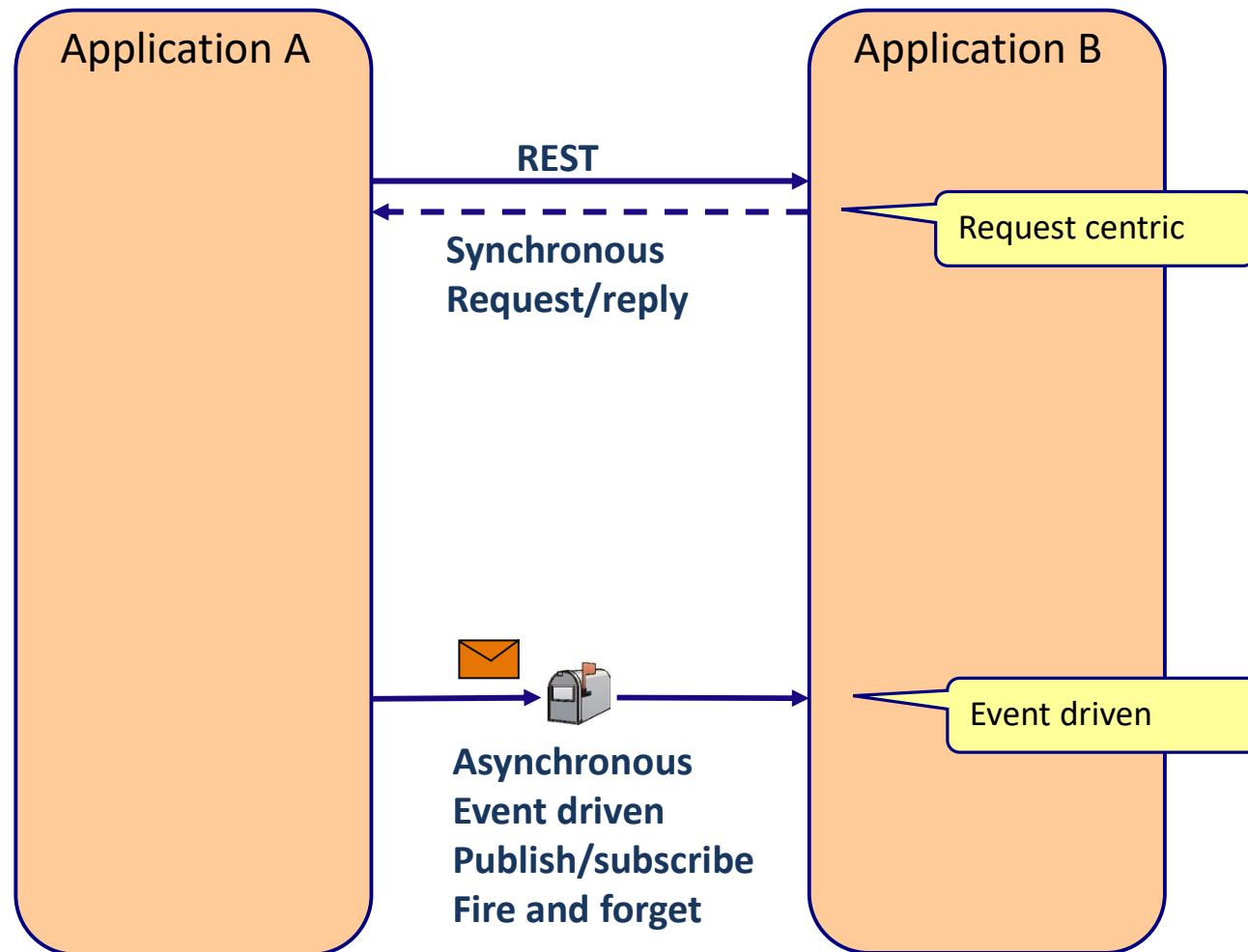
STREAM BASED ARCHITECTURE

Architecture evolution

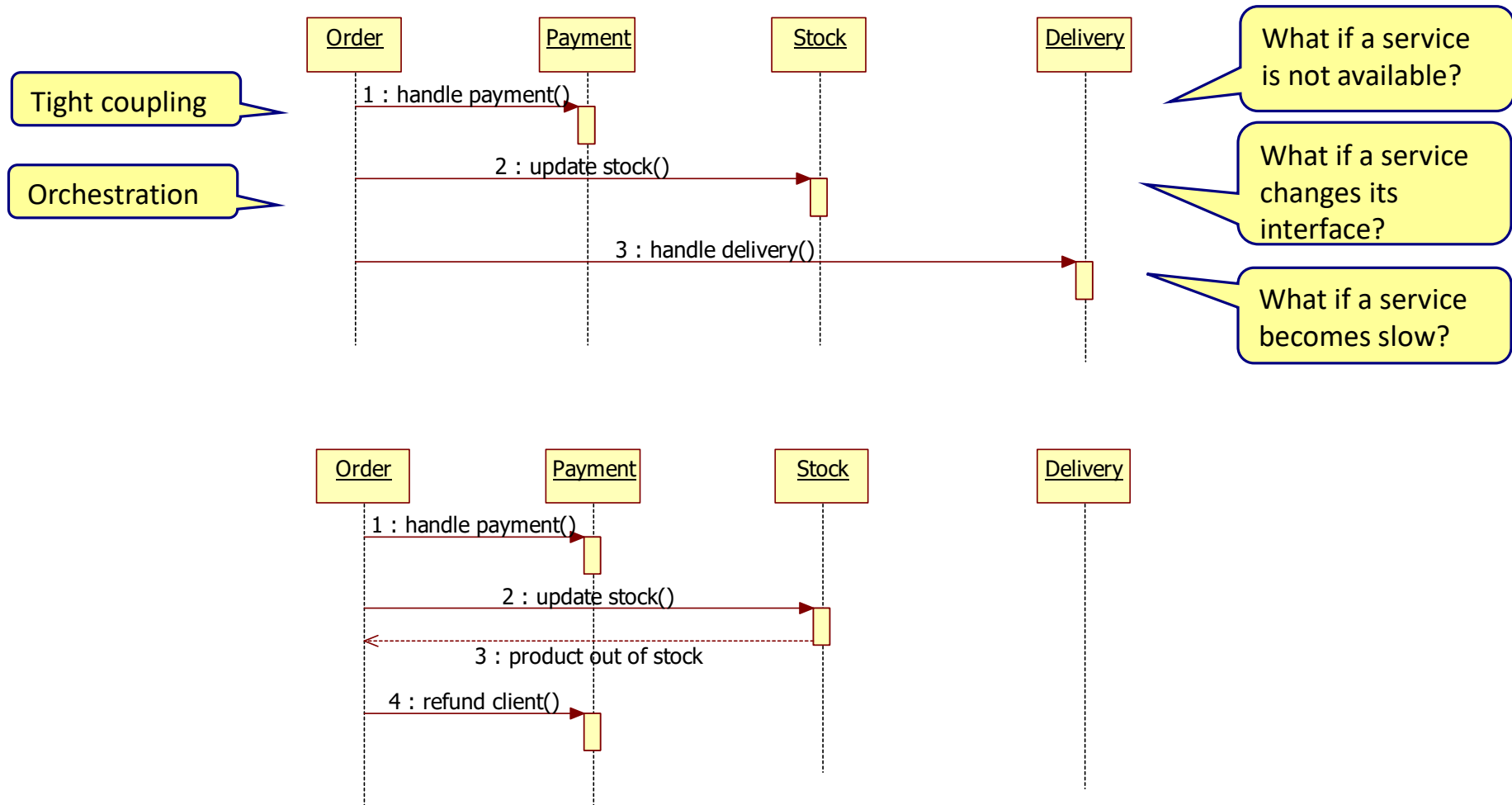


EVENT DRIVEN ARCHITECTURE

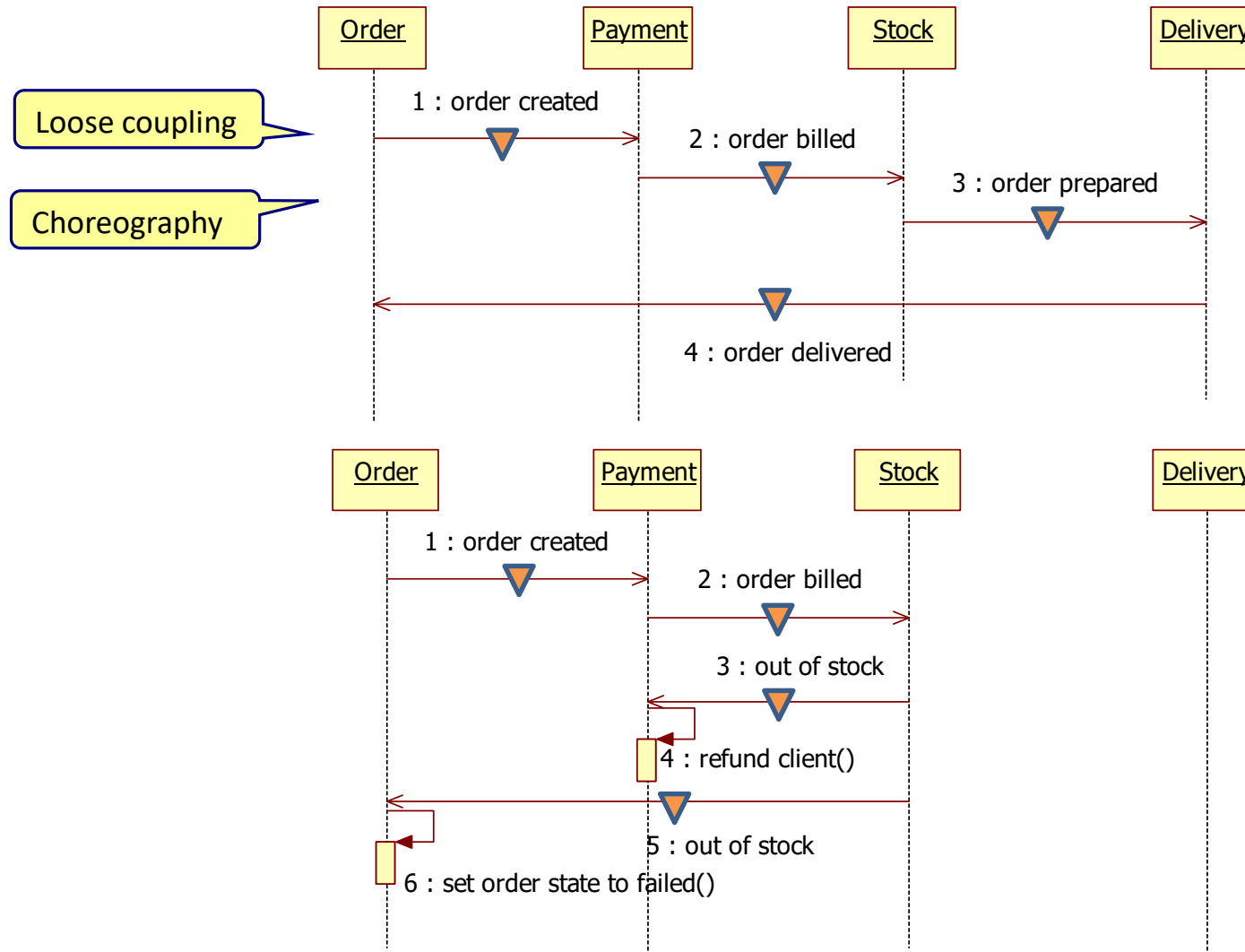
2 ways to communicate



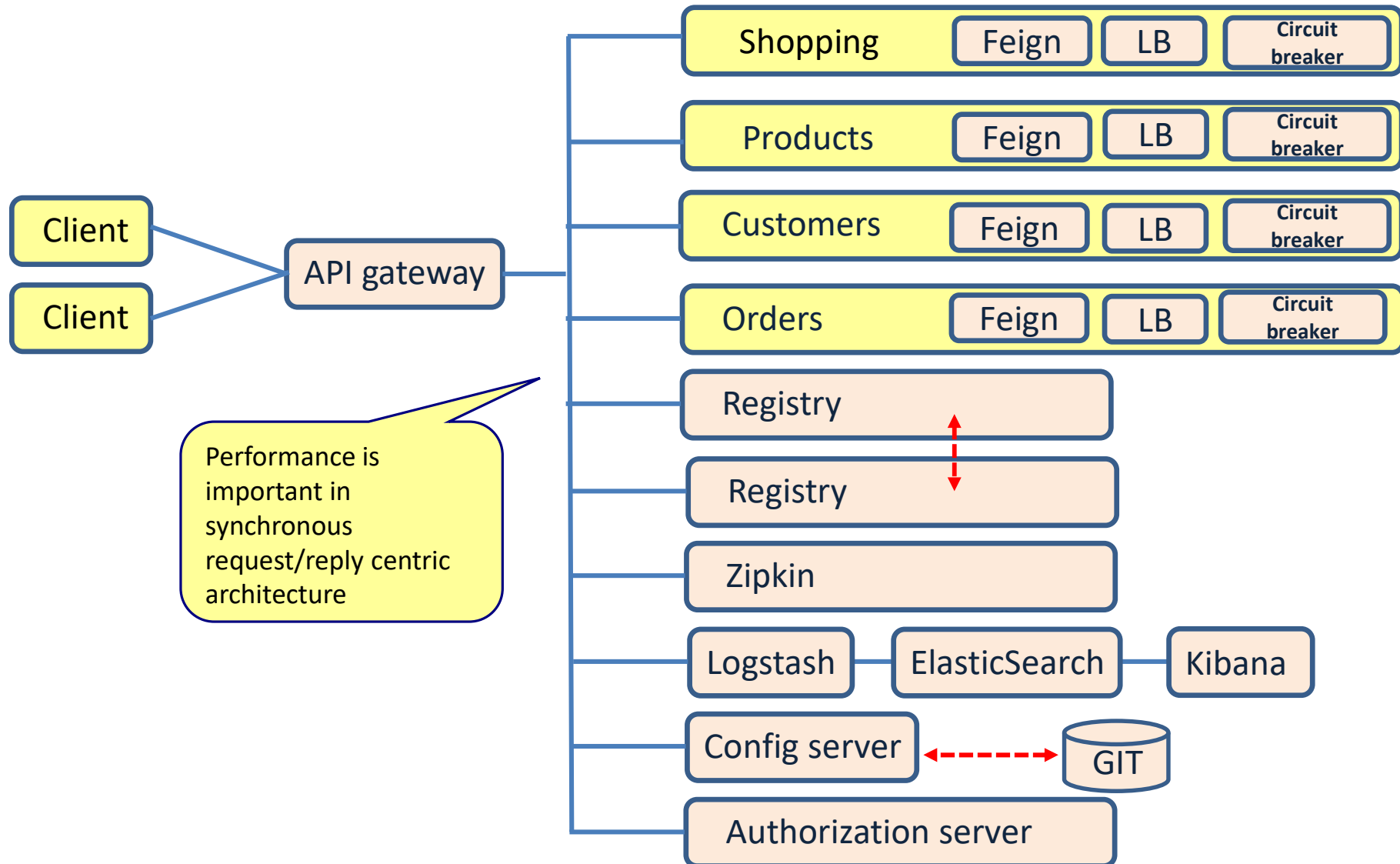
Request centric (REST) calls



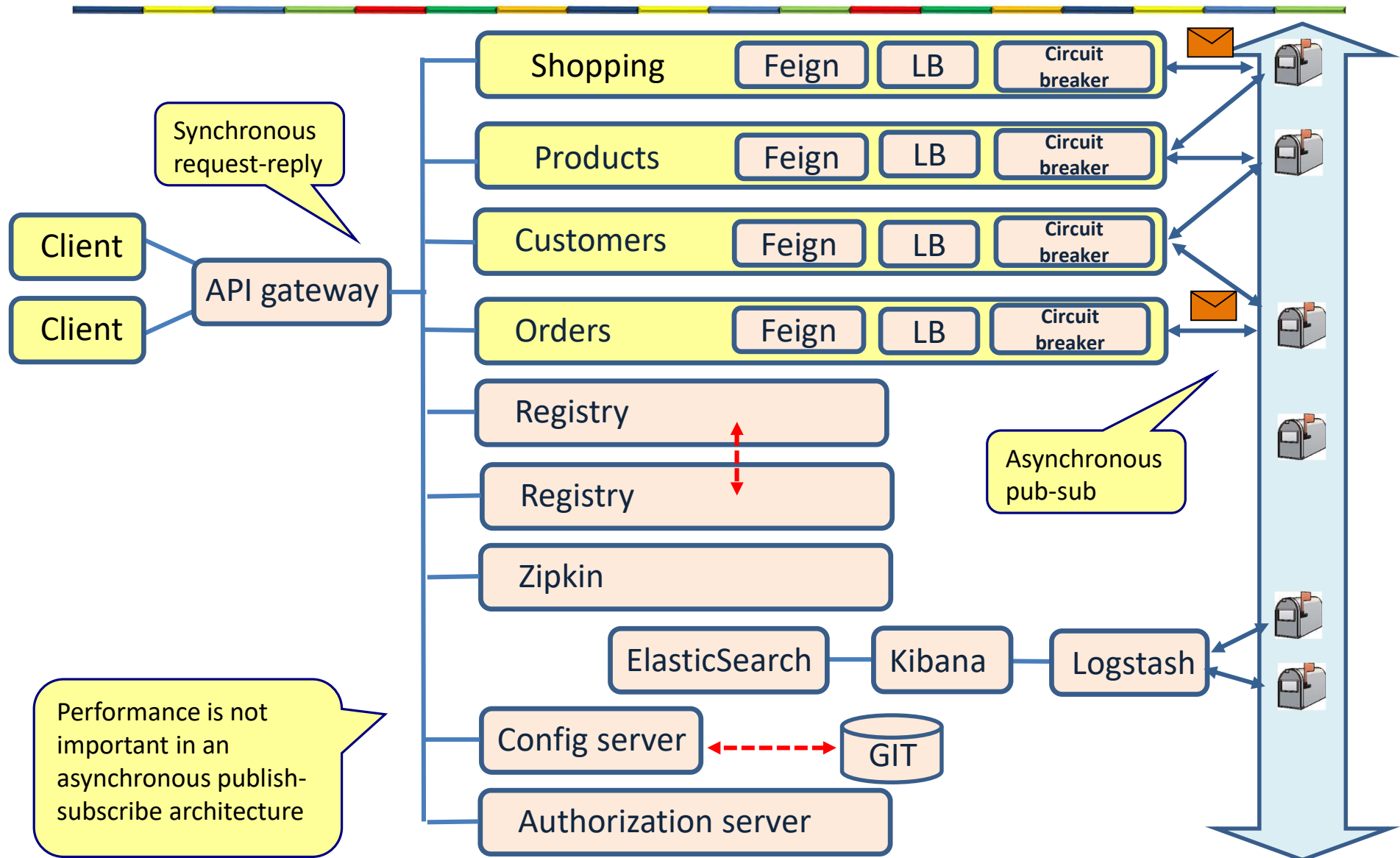
Event driven(messaging)



Implementing microservices



Implementing microservices



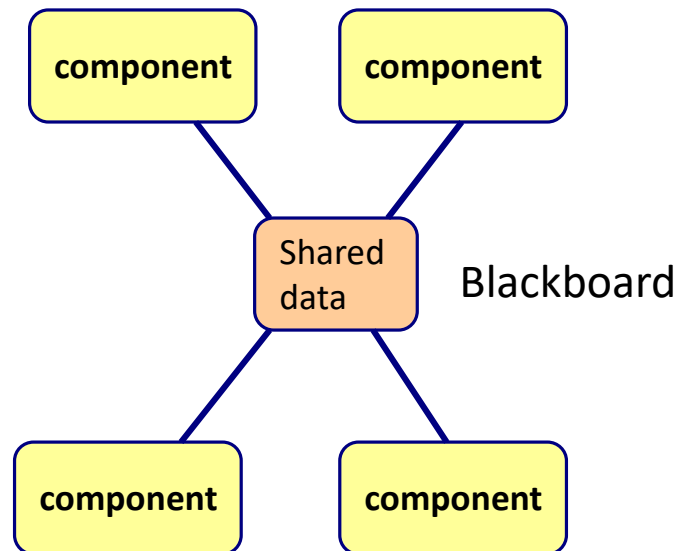
Challenges of a microservice architecture

Challenge	Solution
Complex communication	Feign Registry API gateway
Performance	Event Driven Architecture (EDA)
Resilience	Registry replicas Load balancing between multiple service instances Circuit breaker
Security	Token based security (OAuth2) Digitally signed (JWT) tokens
Transactions	Compensating transactions Eventual consistency
Keep data in sync	Publish-subscribe data change event
Keep interfaces in sync	Spring cloud contract
Keep configuration in sync	Config server
Monitor health of microservices	ELK + beats
Follow/monitor business processes	Zipkin ELK

BLACKBOARD

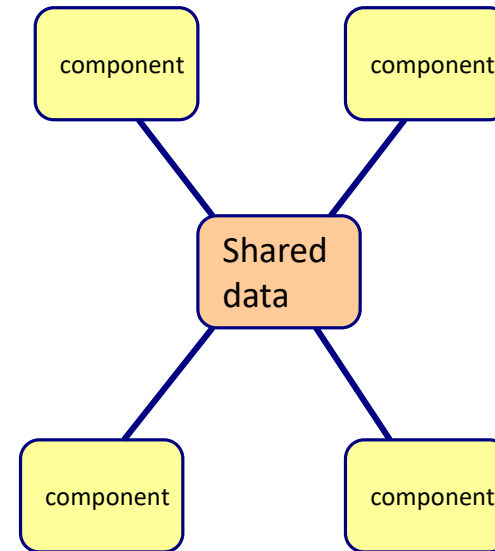
Blackboard

- Used for non deterministic problems
 - There is no fixed straight-line solution to a problem
- Every component adds her information on the blackboard



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



Blackboard

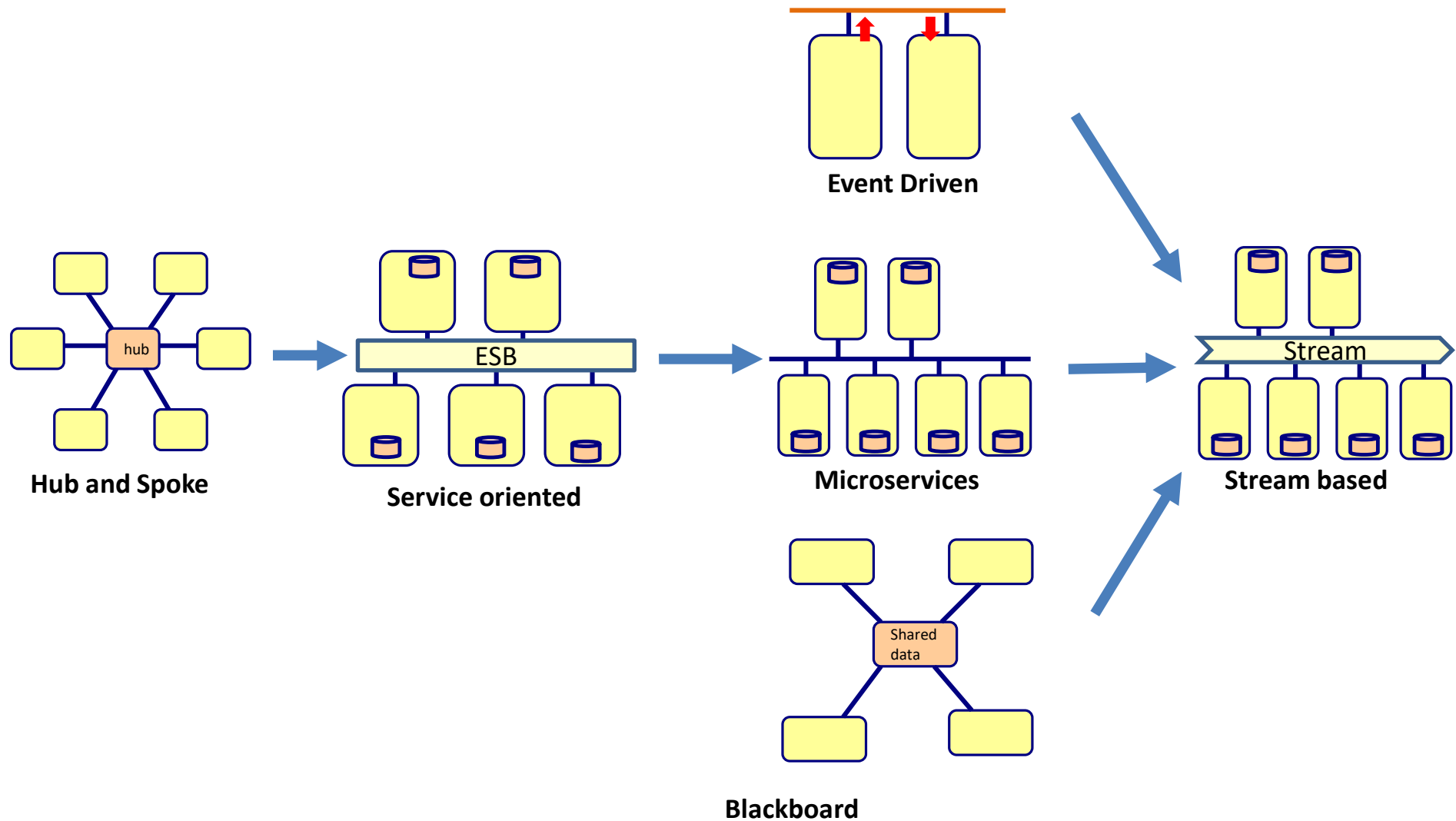
- Benefits

- Easy to add new components
- Components are independent of each other
- Components can work in parallel

- Drawbacks

- Data structure is hard to change
 - All components share the same data structure
- Synchronization issues

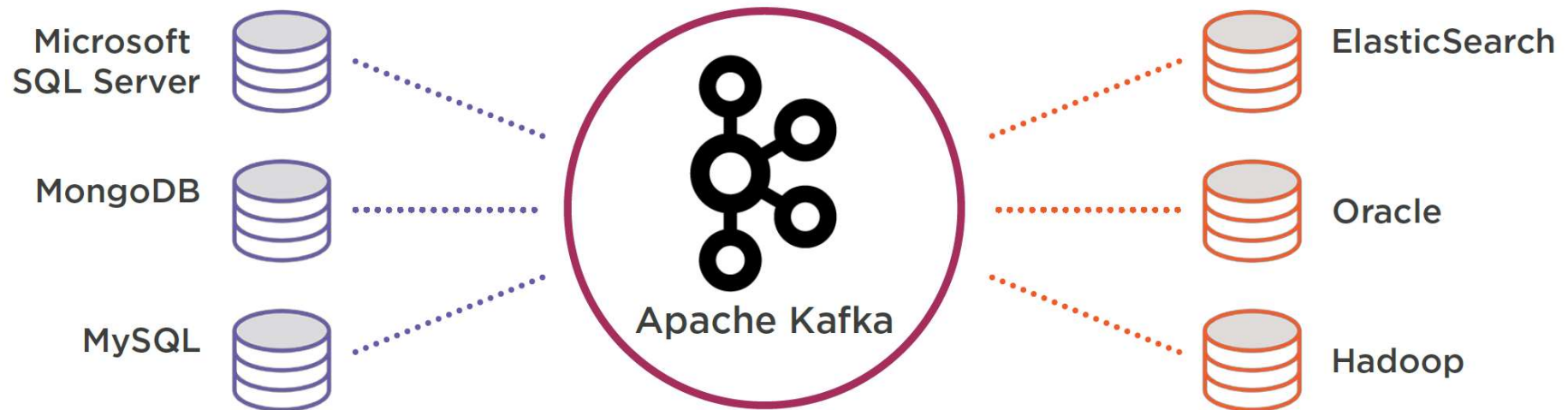
Architecture evolution



KAFKA OVERVIEW

What is Kafka?

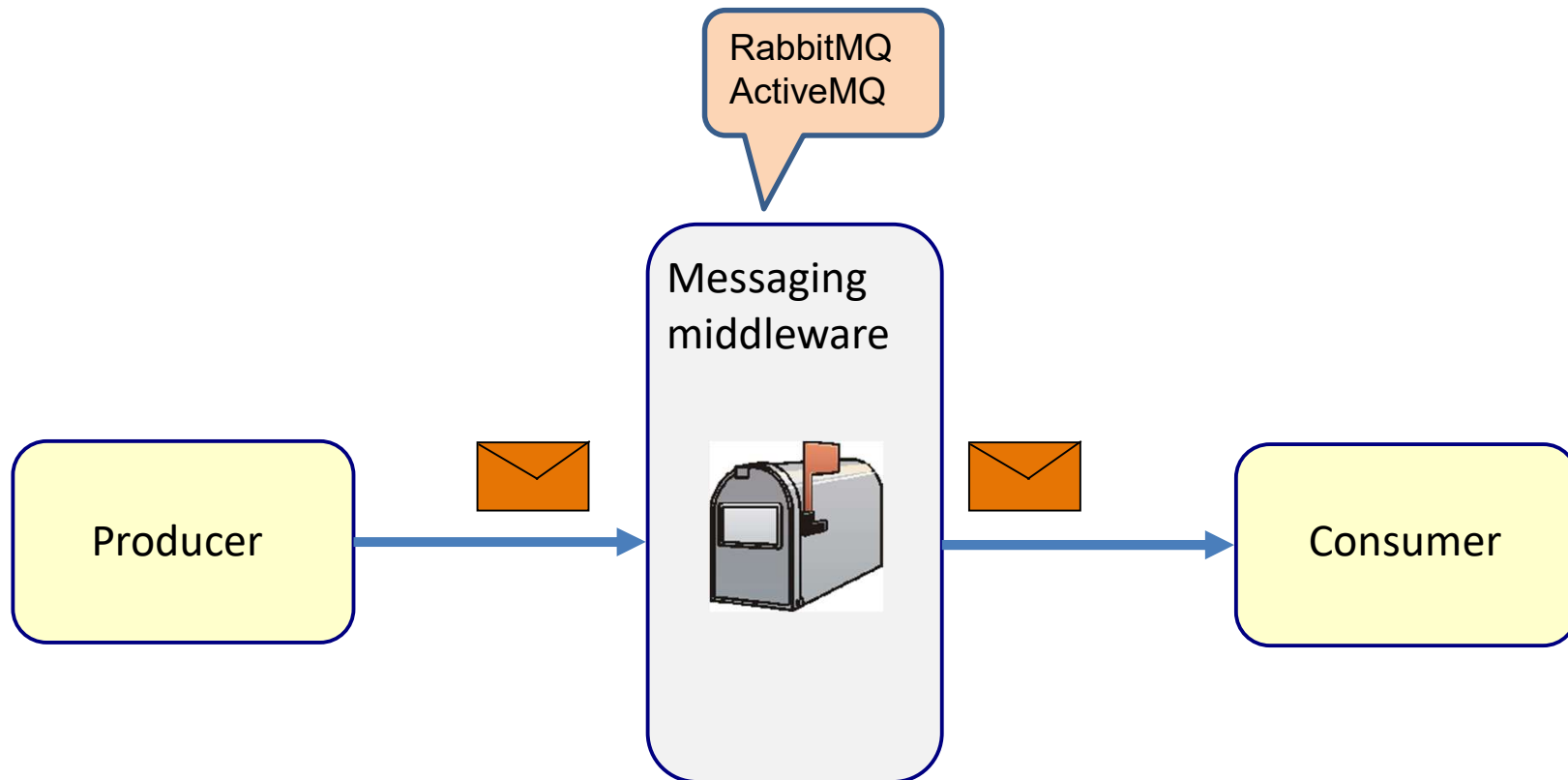
- High-throughput distributed messaging system



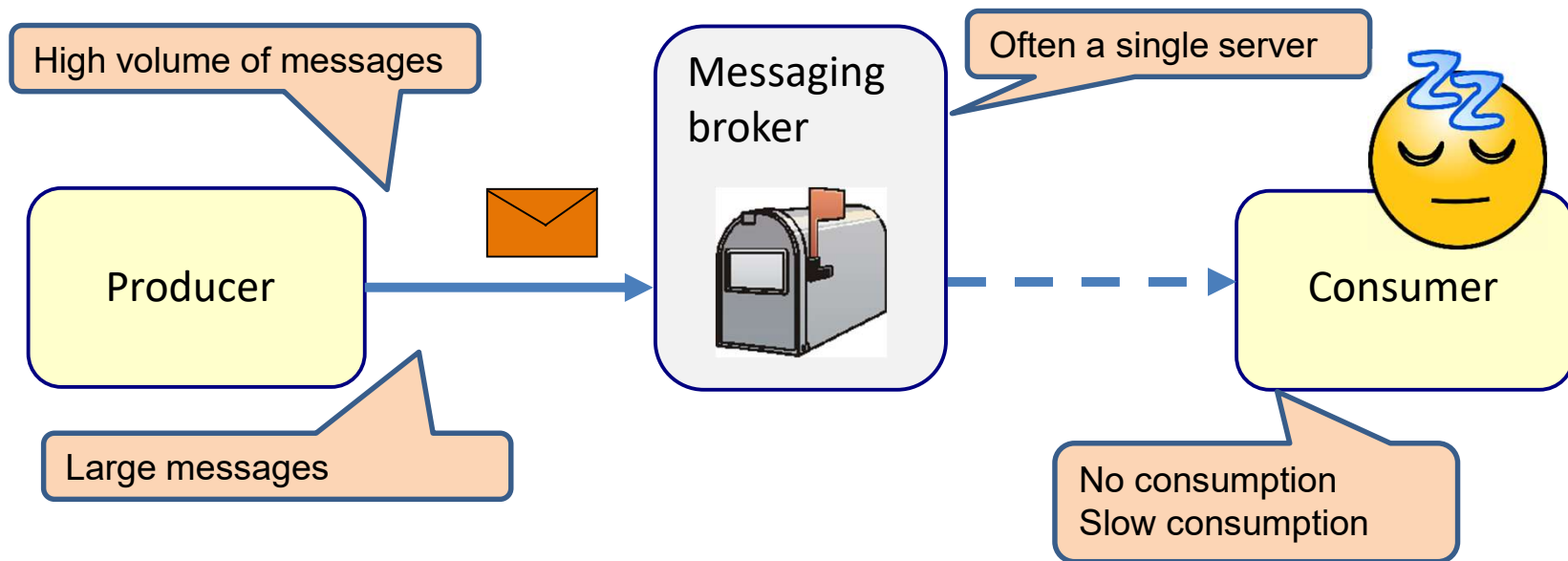
REST vs. Messaging

- REST
 - Synchronous
 - Tight coupling
- Messaging
 - Asynchronous
 - Fire and forget
 - Loose coupling
 - Buffer
 - Middleware needs to be maintained

Traditional Messaging Systems

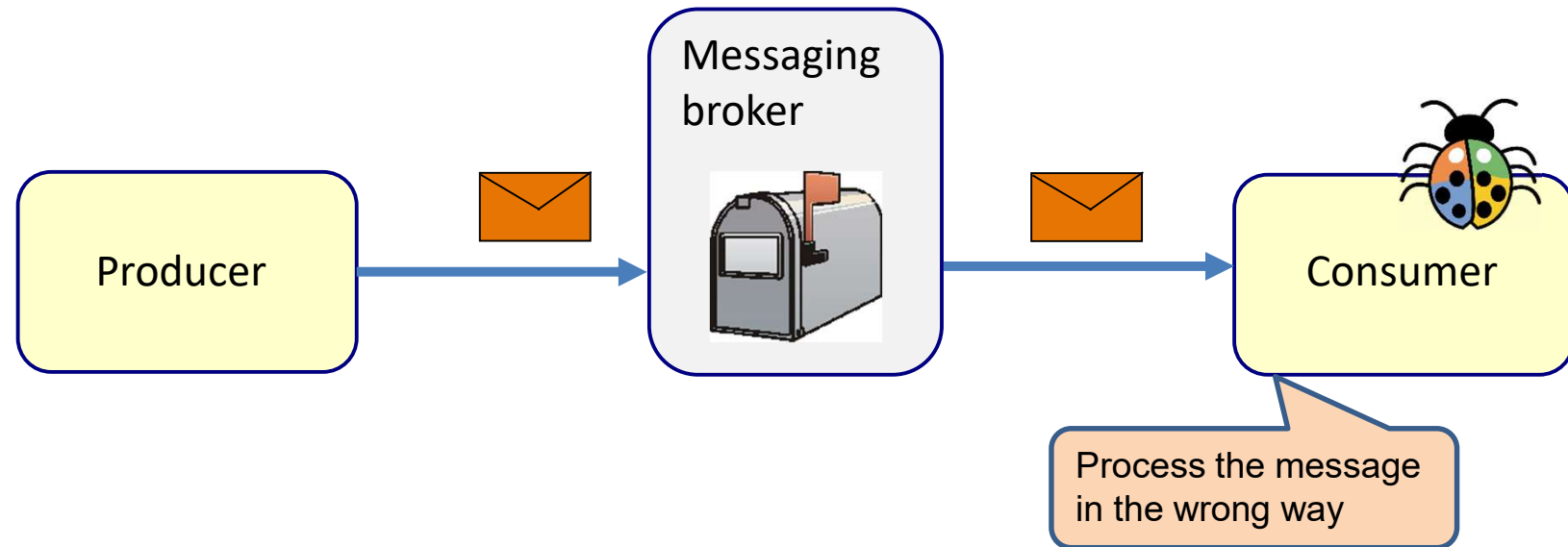


Problems with traditional messaging middleware



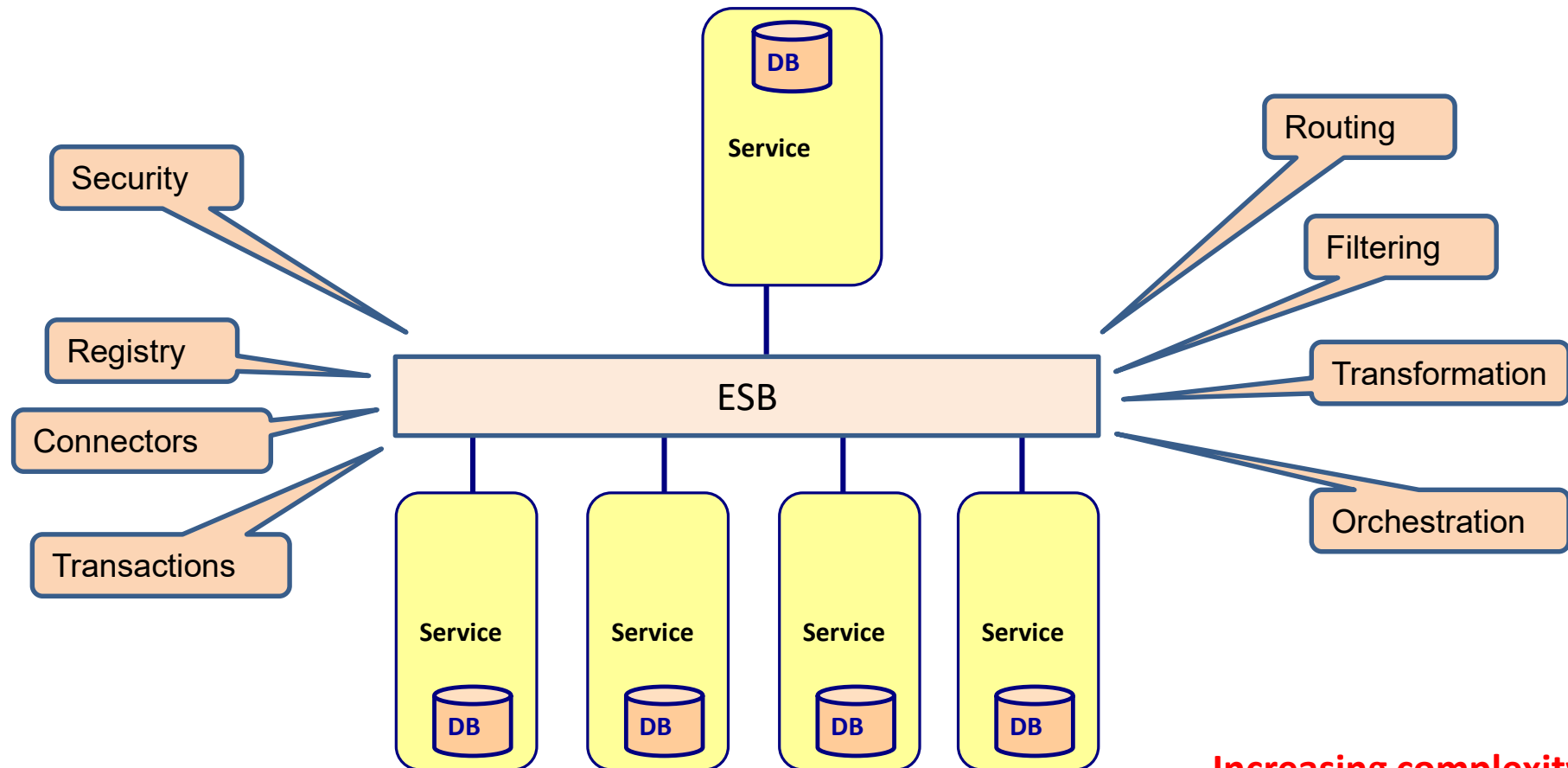
- If the consumer is temporally not available (or very slow) the message middleware has to store the messages
 - This restricts the volume of messages and the size of the messages
 - Eventually the message broker will fail

Problems with traditional messaging middleware



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

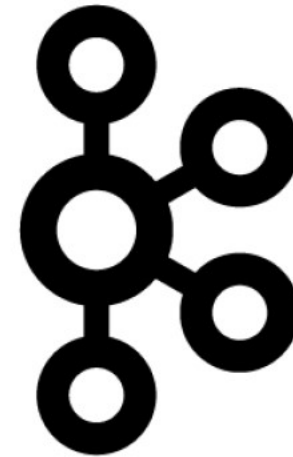
Enterprise Service Bus



Increasing complexity
Deceiving
Potentially expensive

What we need?

- Move data around
 - Cleanly
 - Reliably
 - Quickly
 - Autonomously



Kafka

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

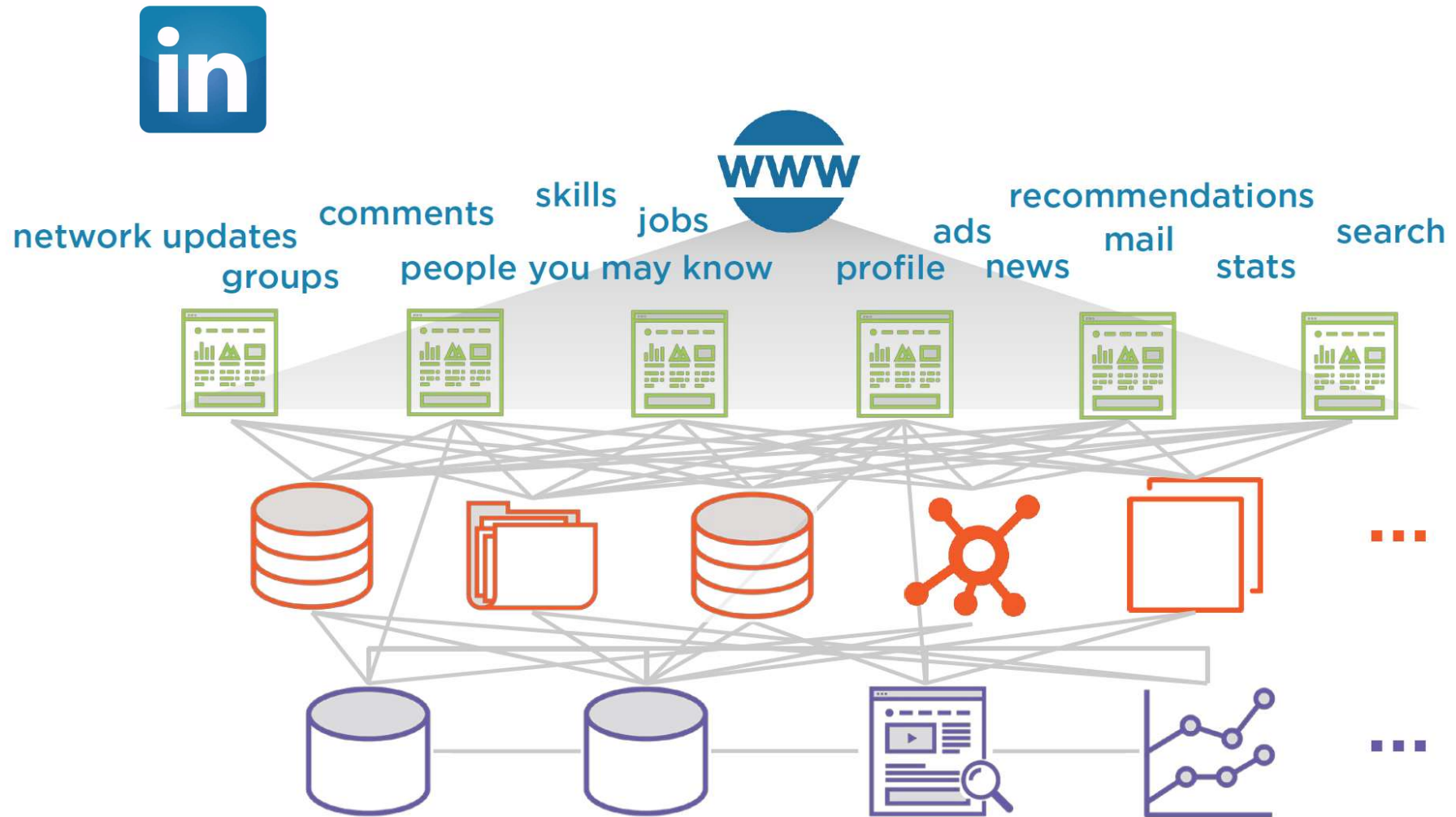
High Volume:

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

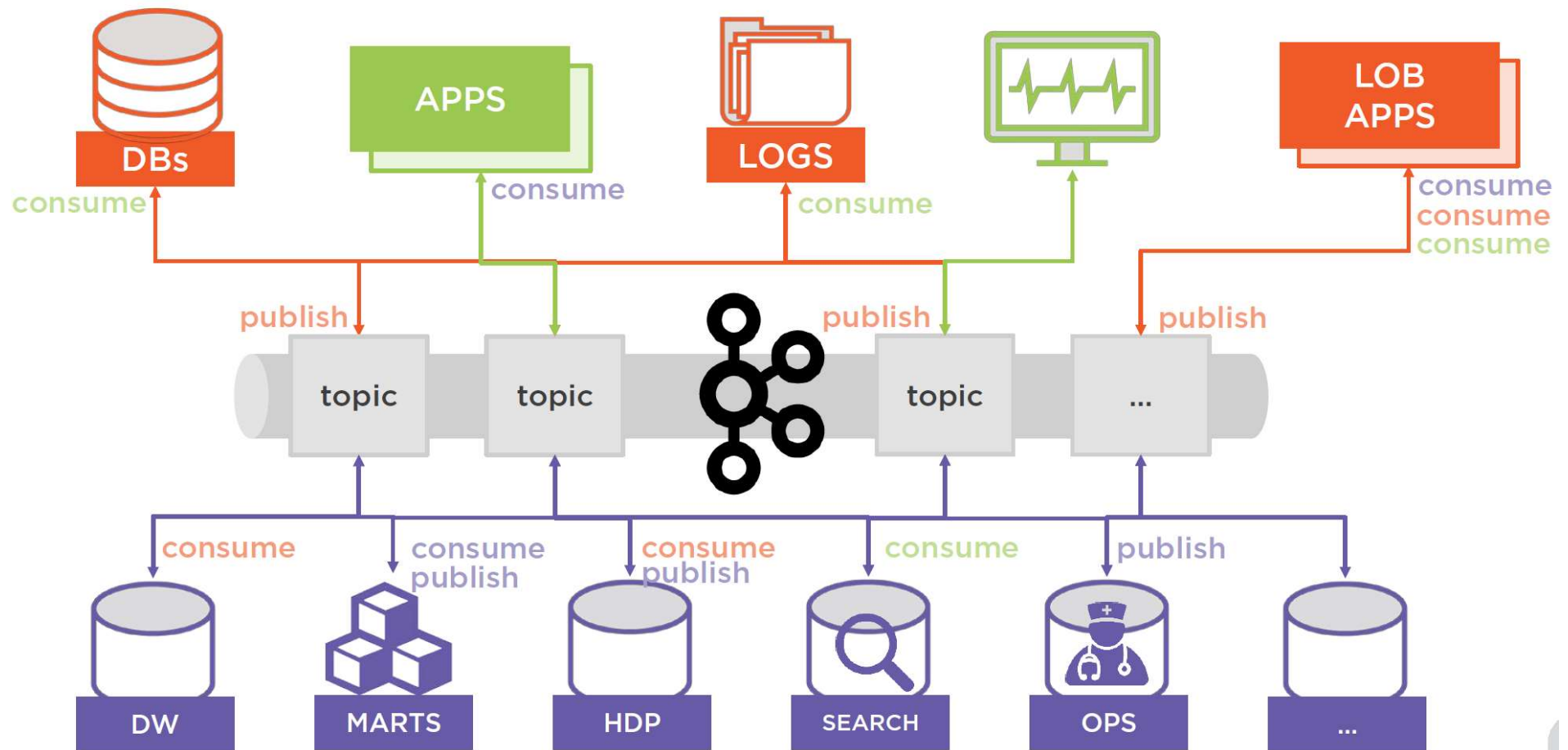
High Velocity:

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

Pre 2010 LinkedIn data architecture

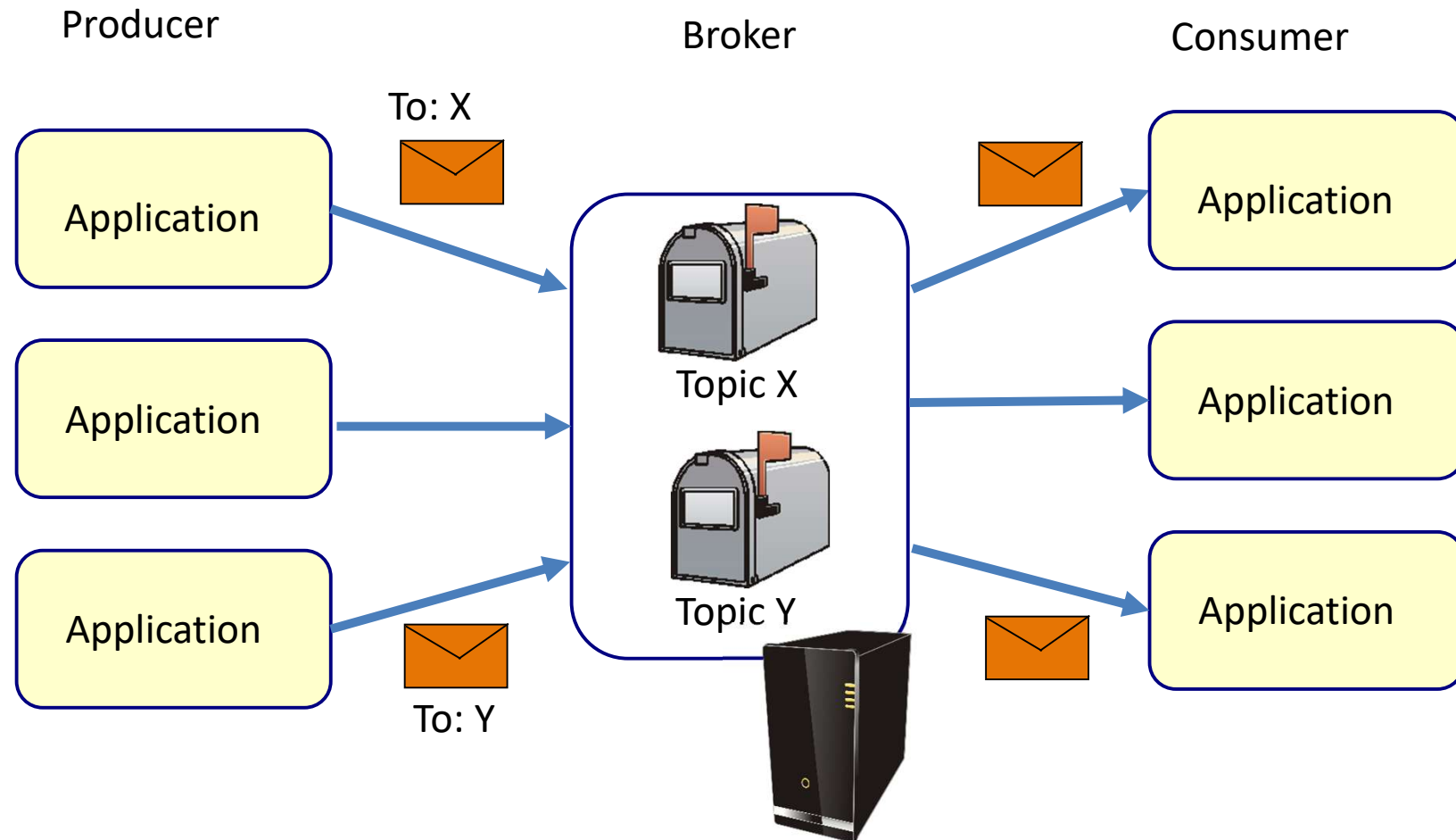


Post 2010 LinkedIn data architecture

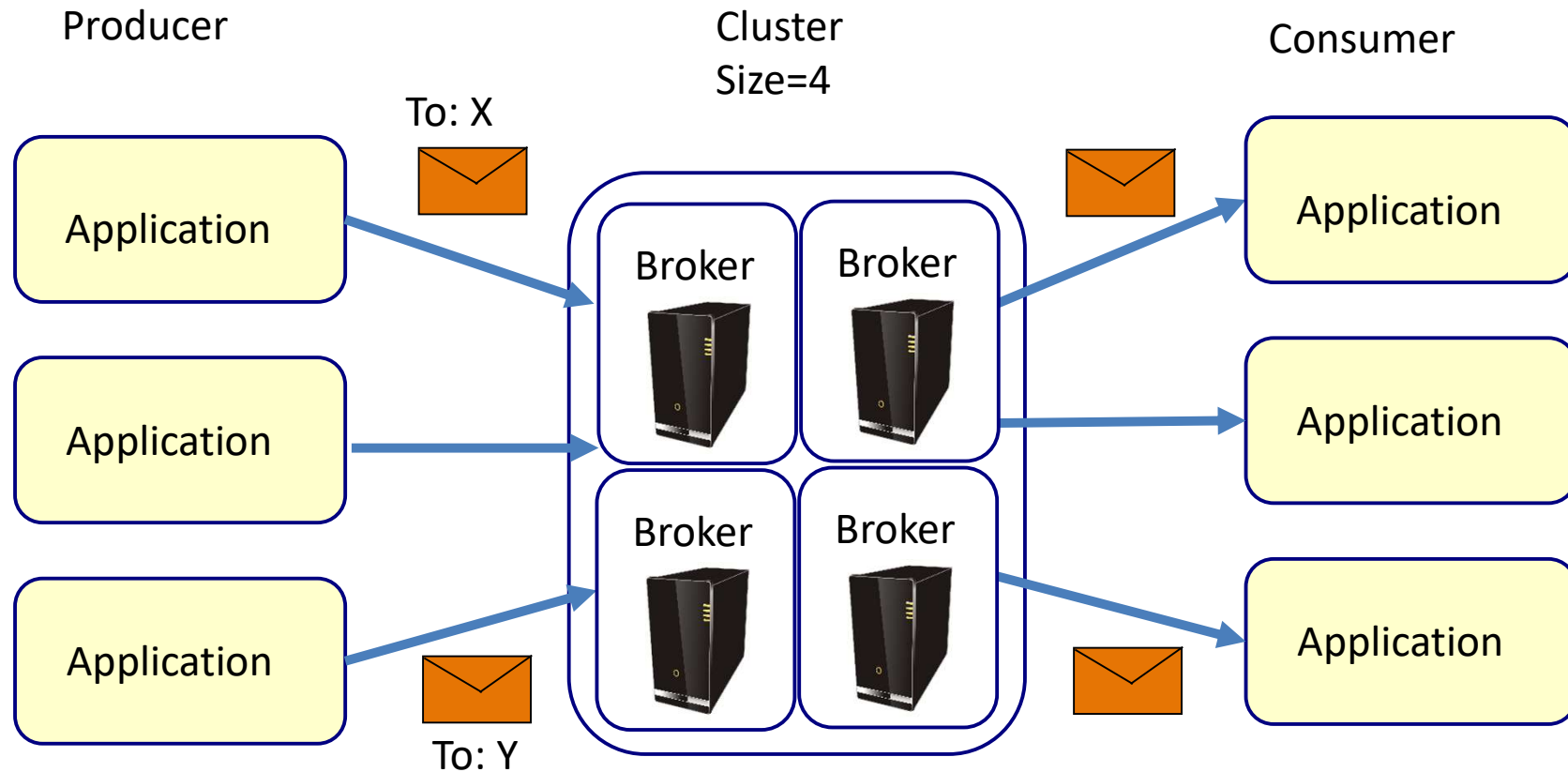


KAFKA'S ARCHITECTURE

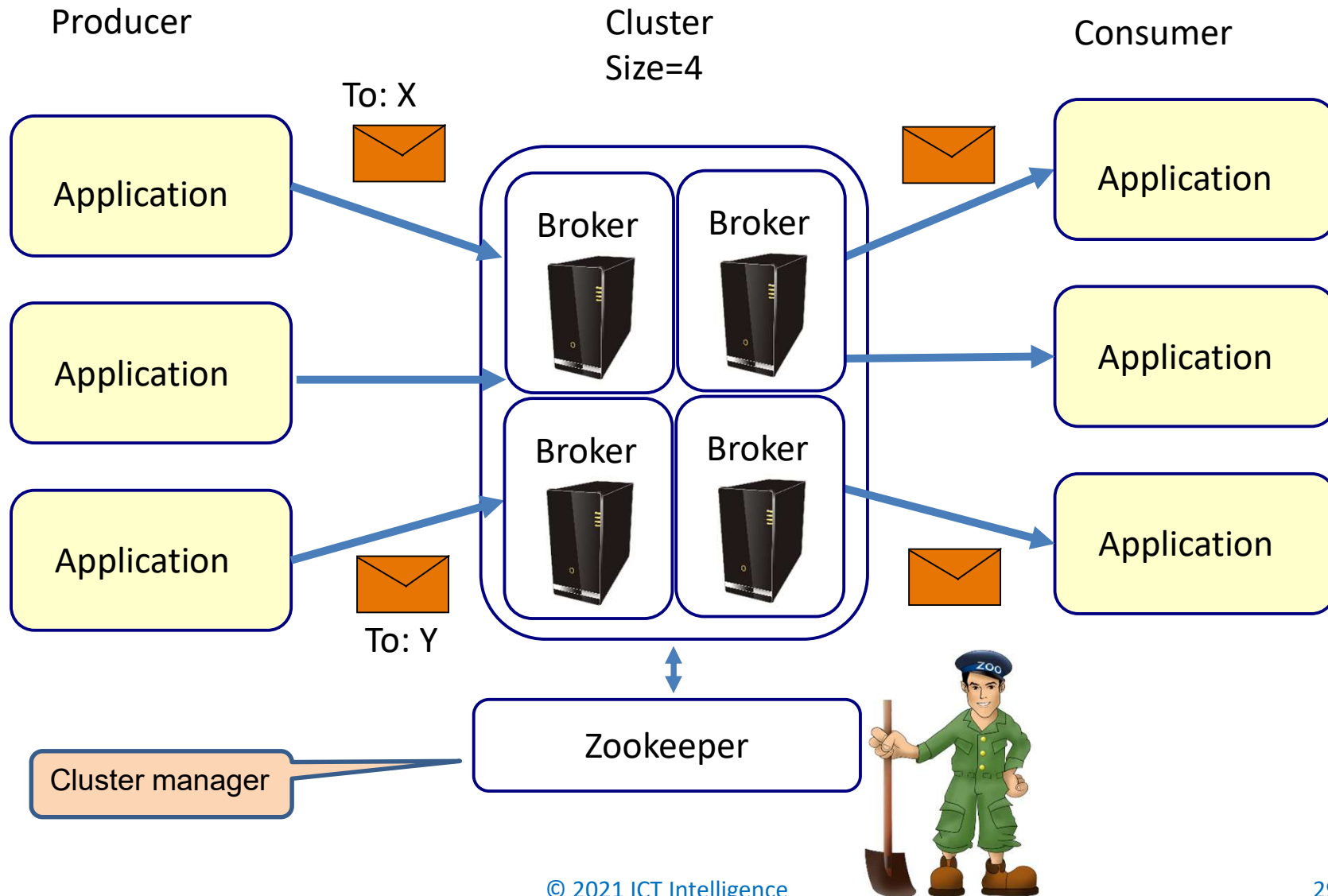
Kafka



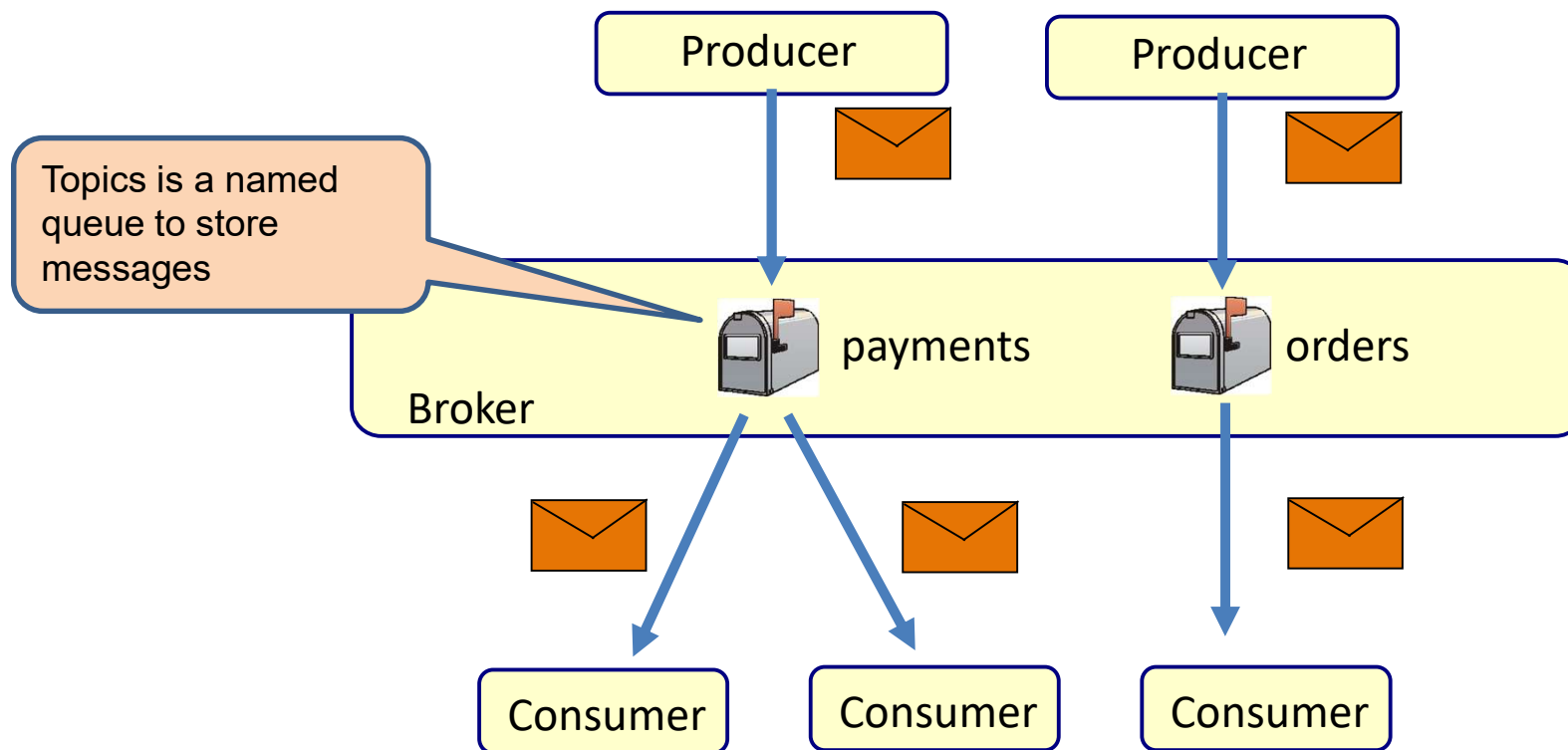
Cluster of Brokers



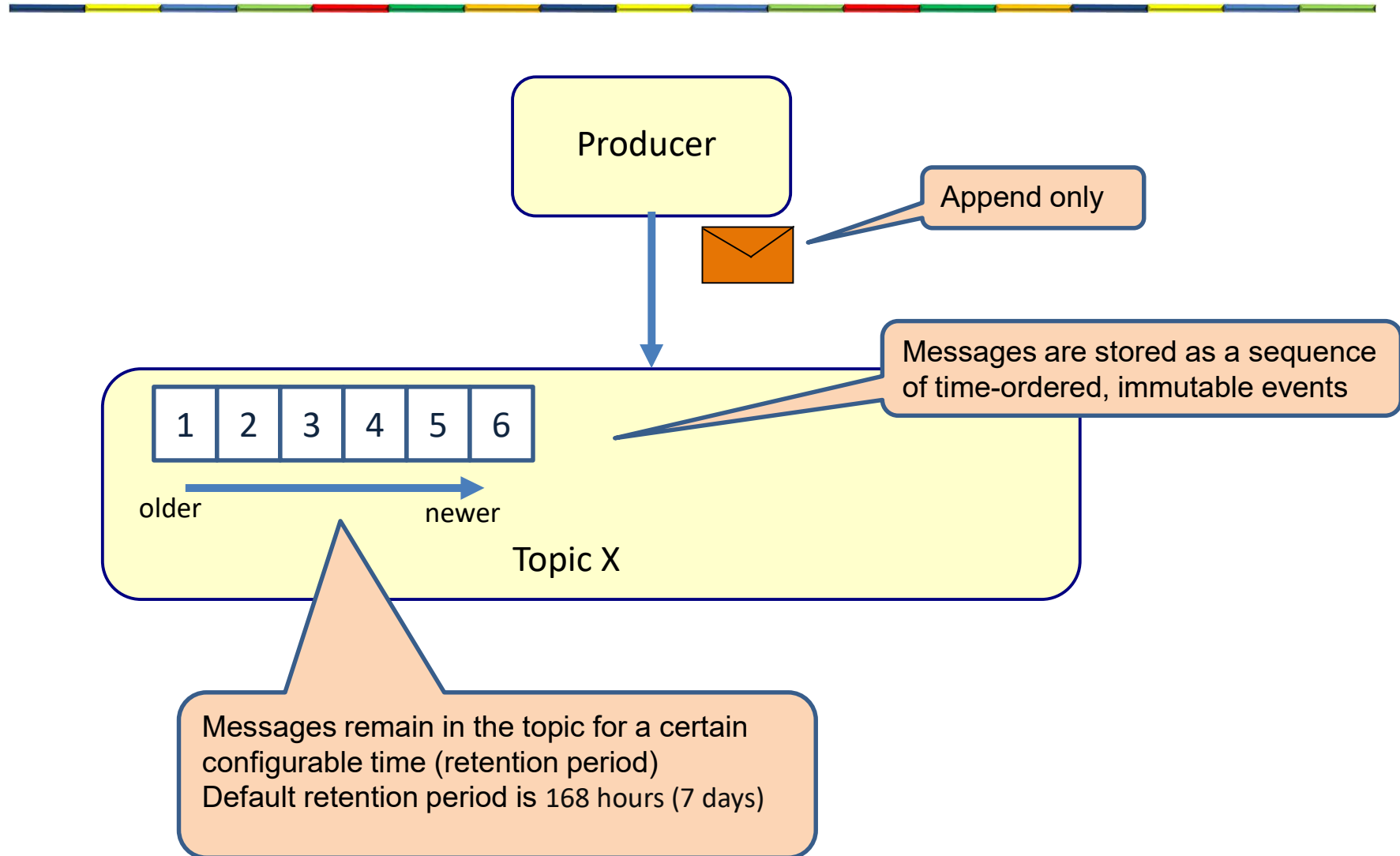
Apache Zookeeper



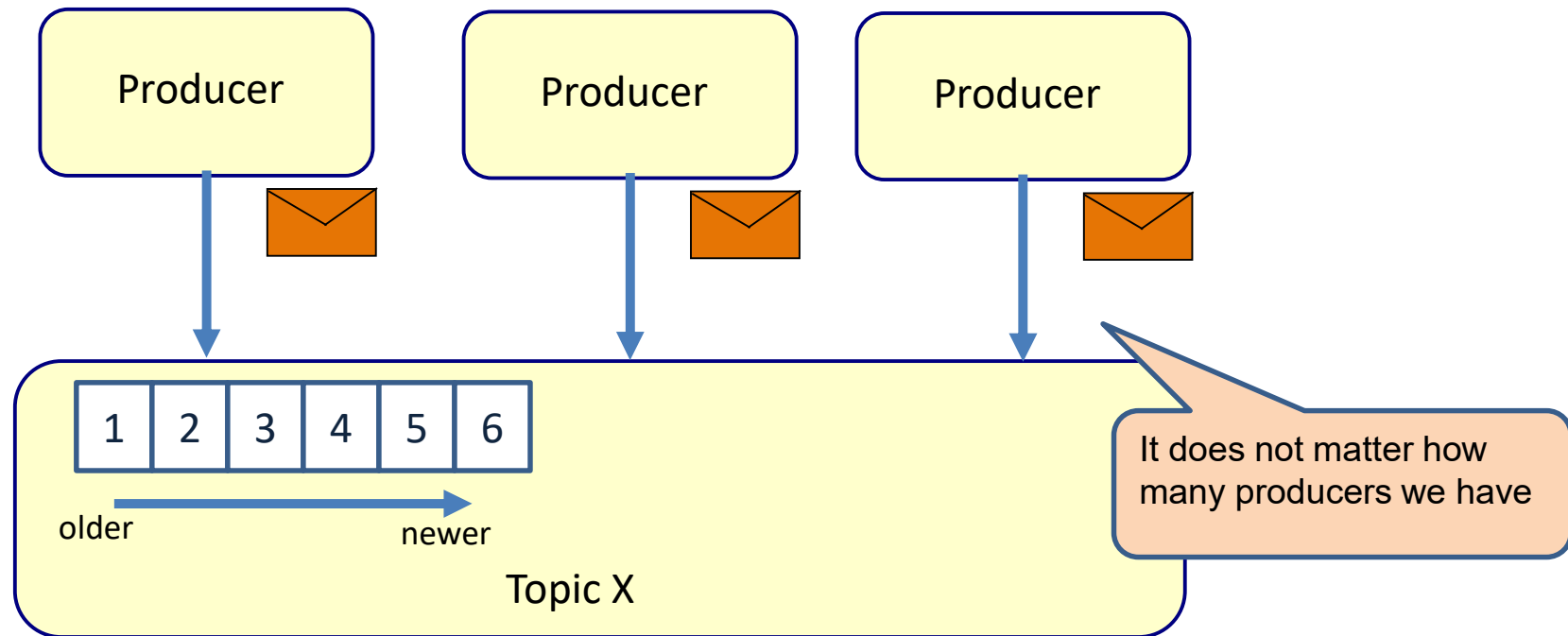
Topics



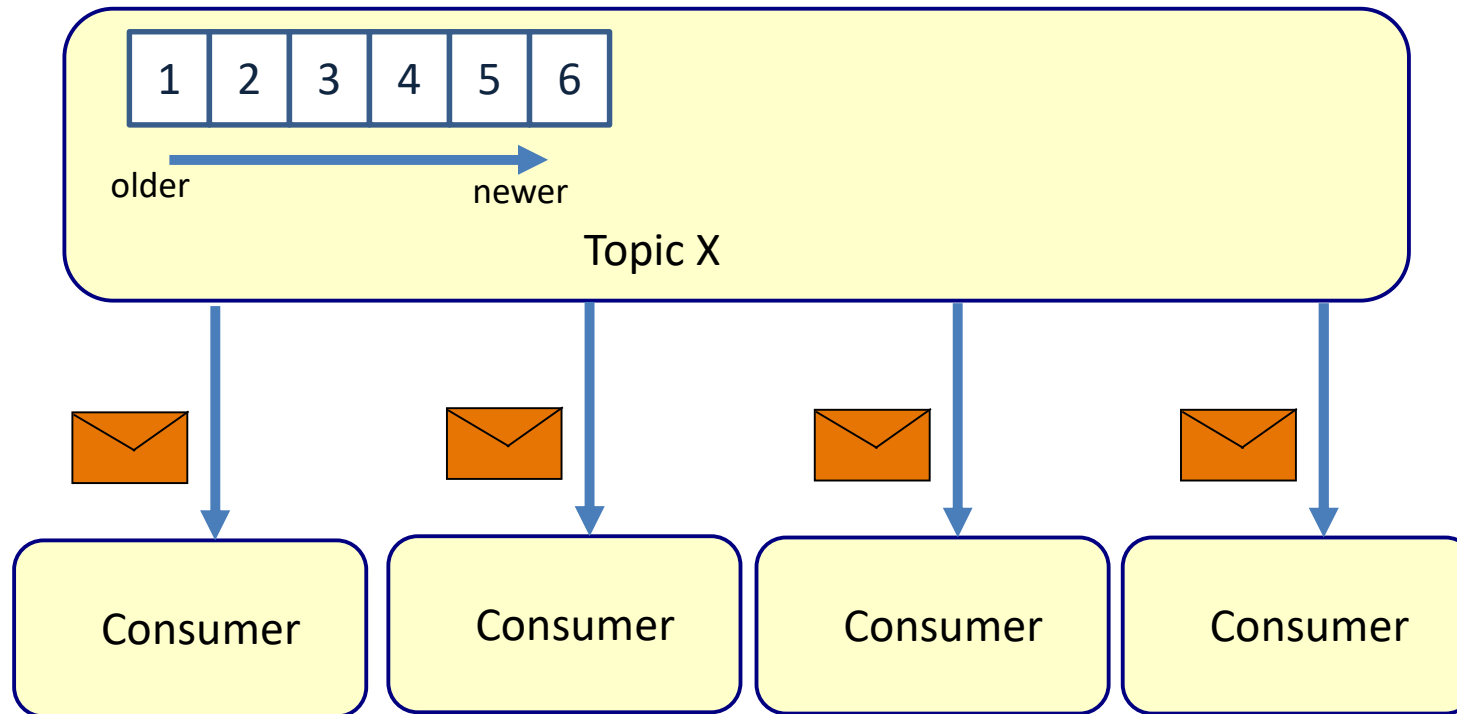
Event sourcing



Why event sourcing?

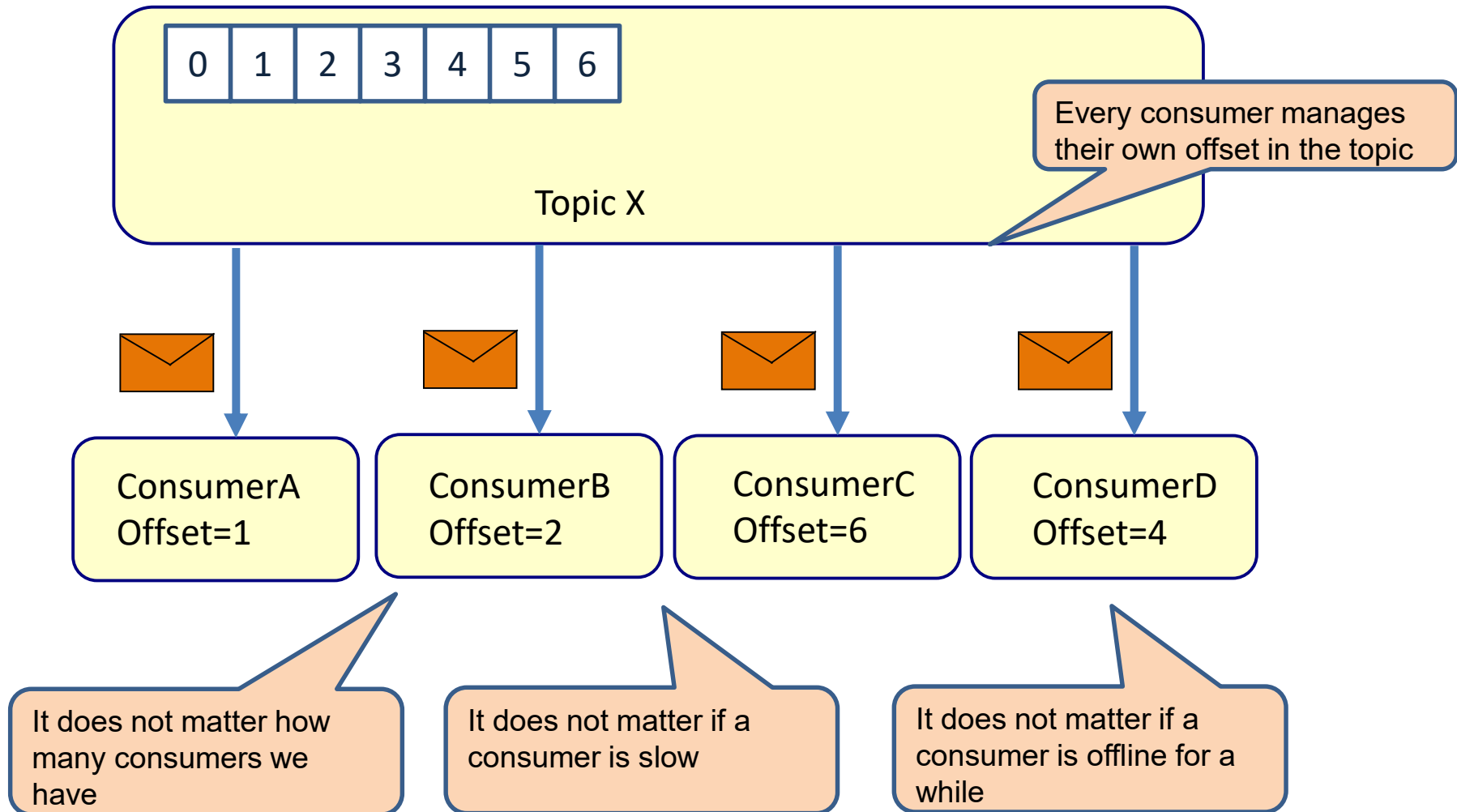


Why event sourcing?



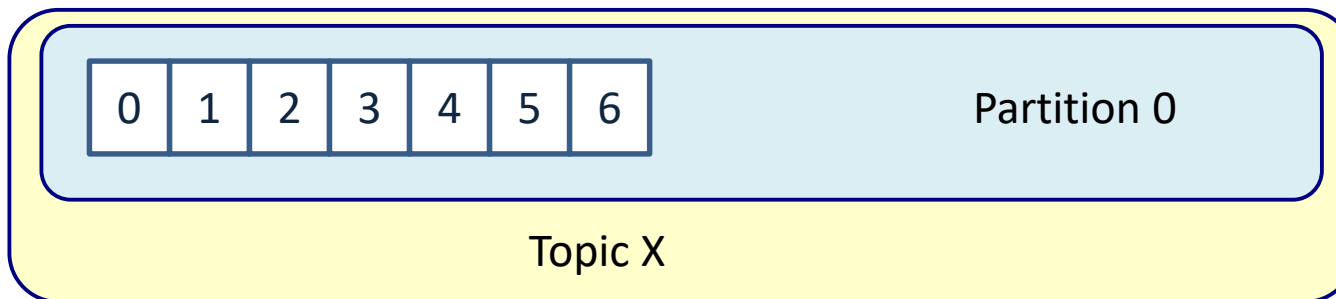
It does not matter how many consumers we have

Offset

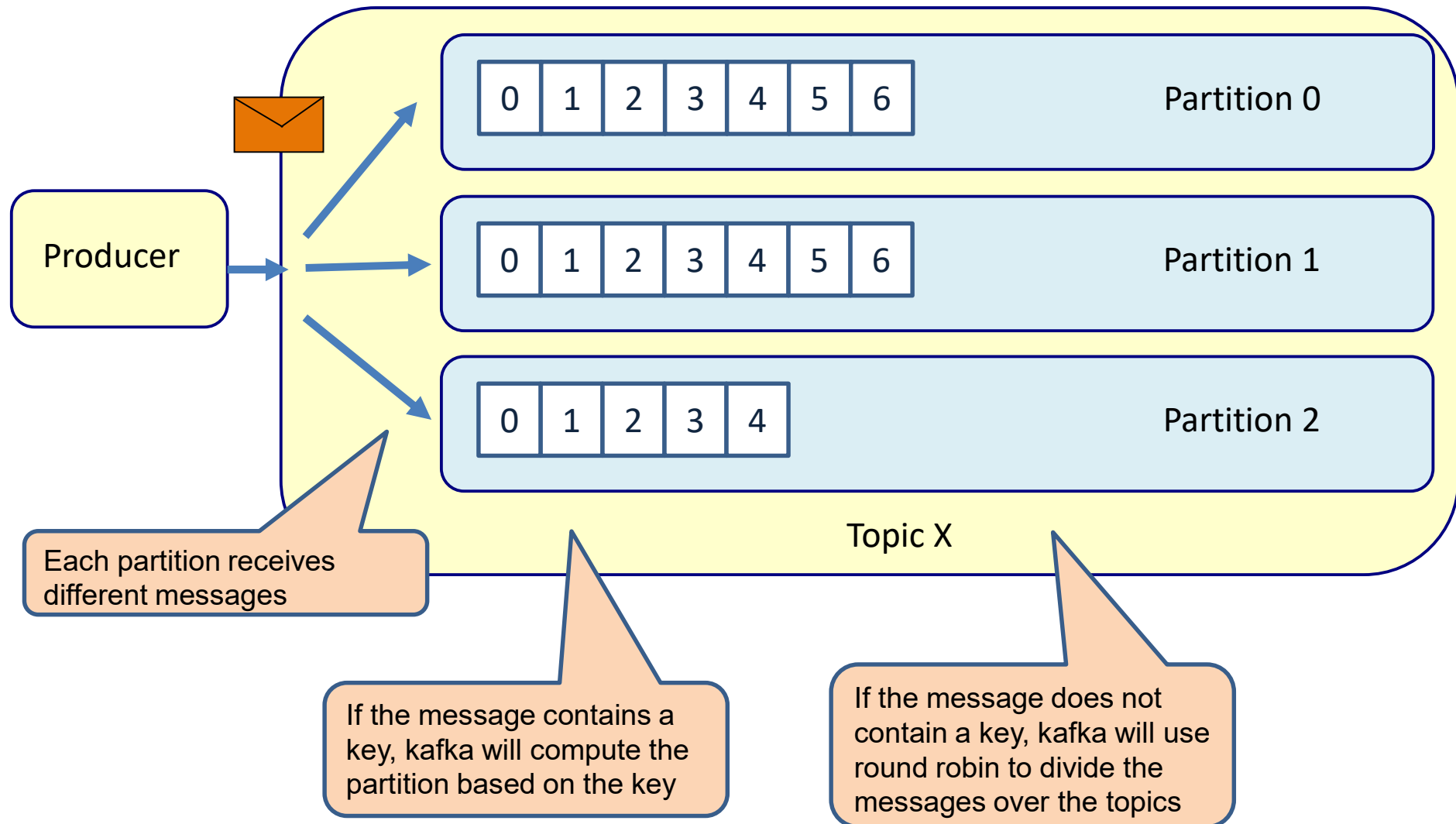


Partition

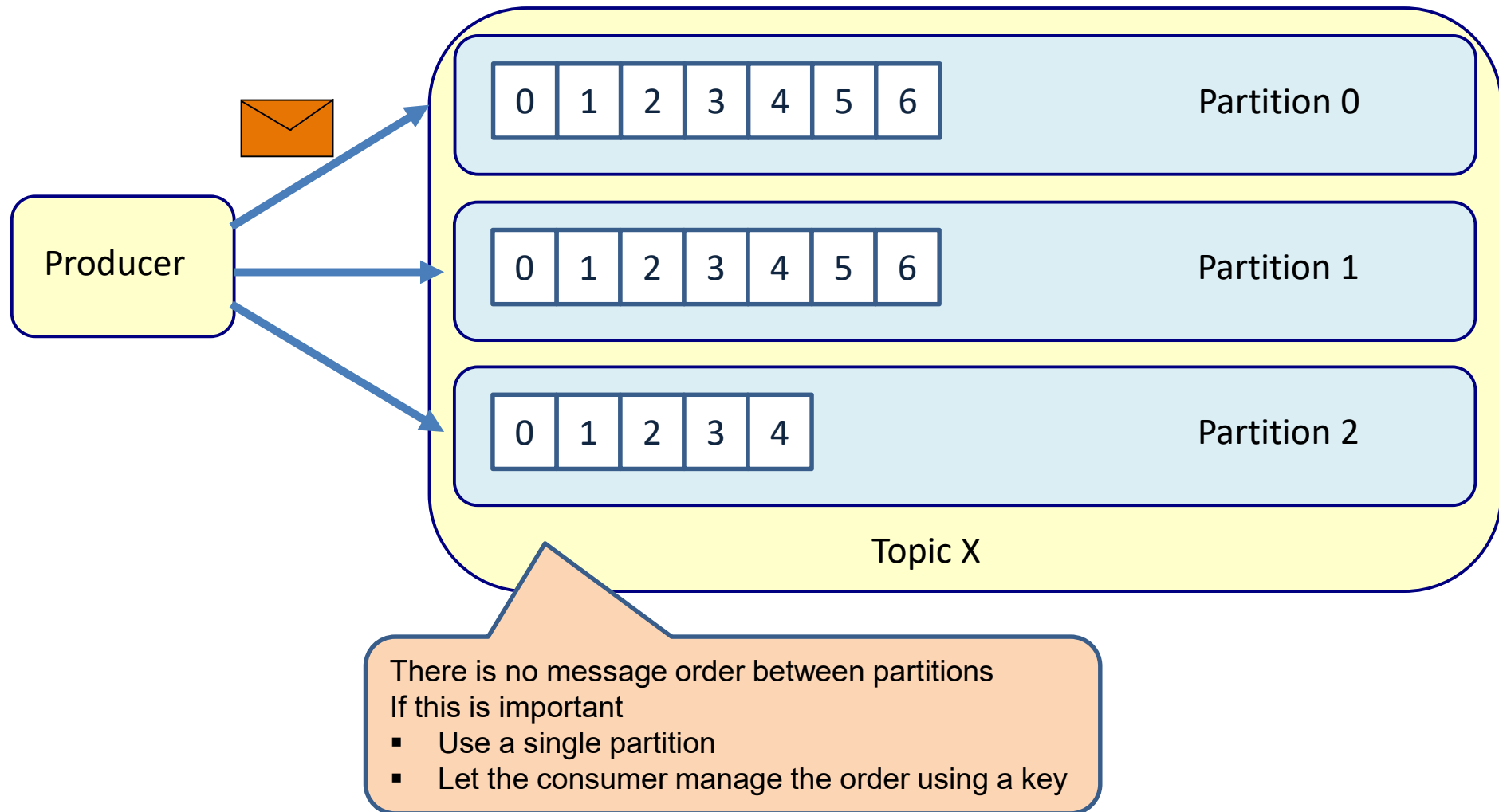
- Each topic has one or more partitions
 - This is configurable
- Each partition must fit on 1 broker



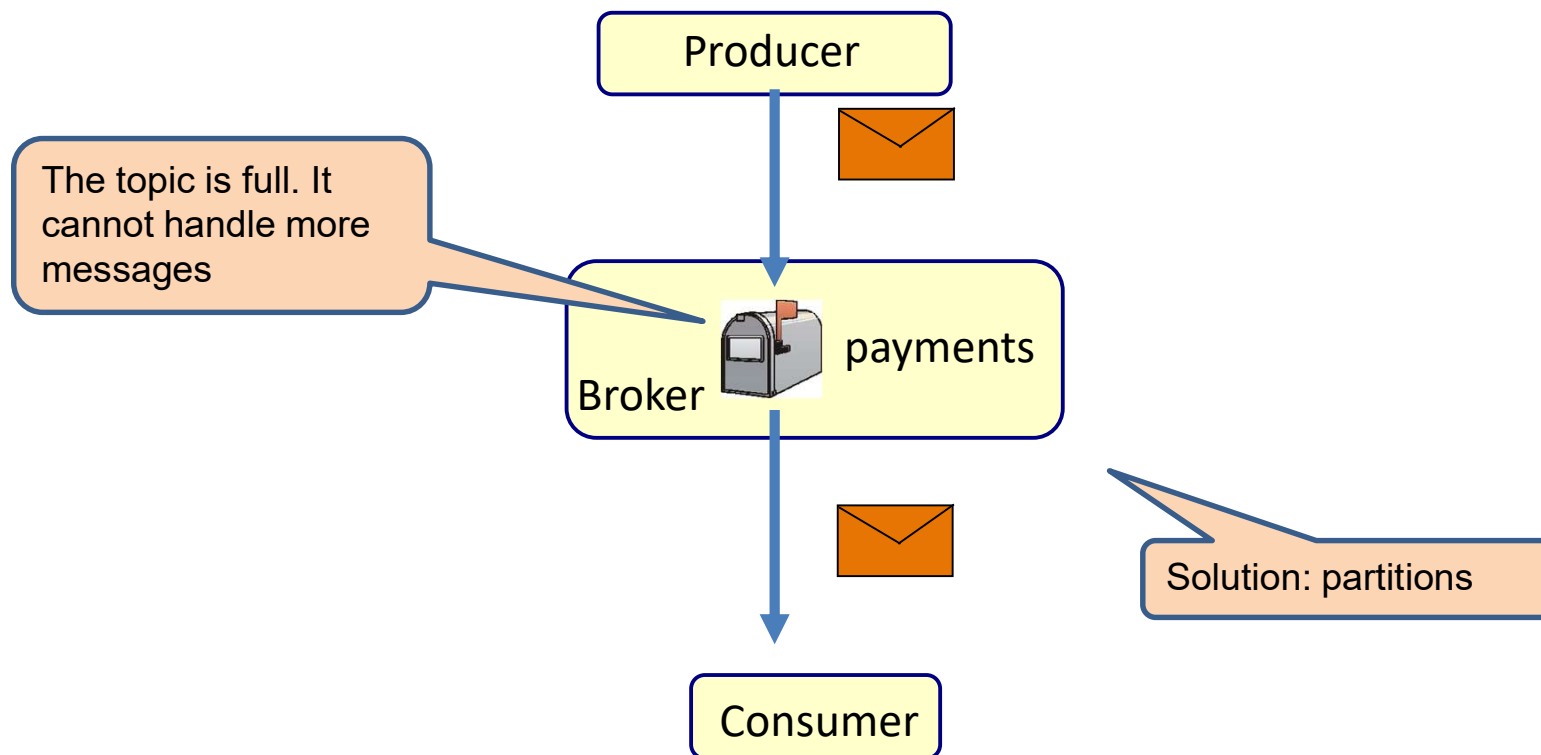
3 partitions



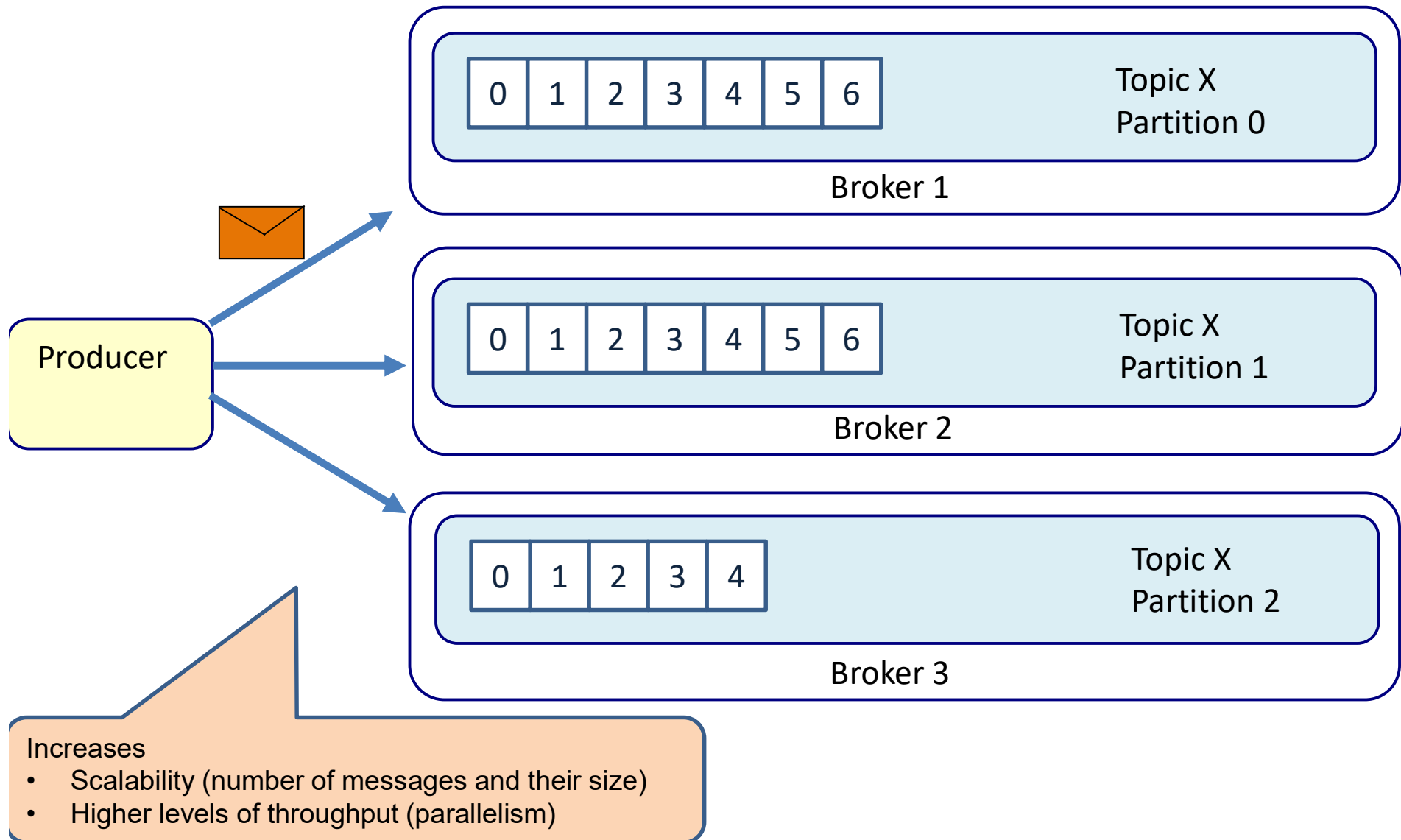
3 partitions



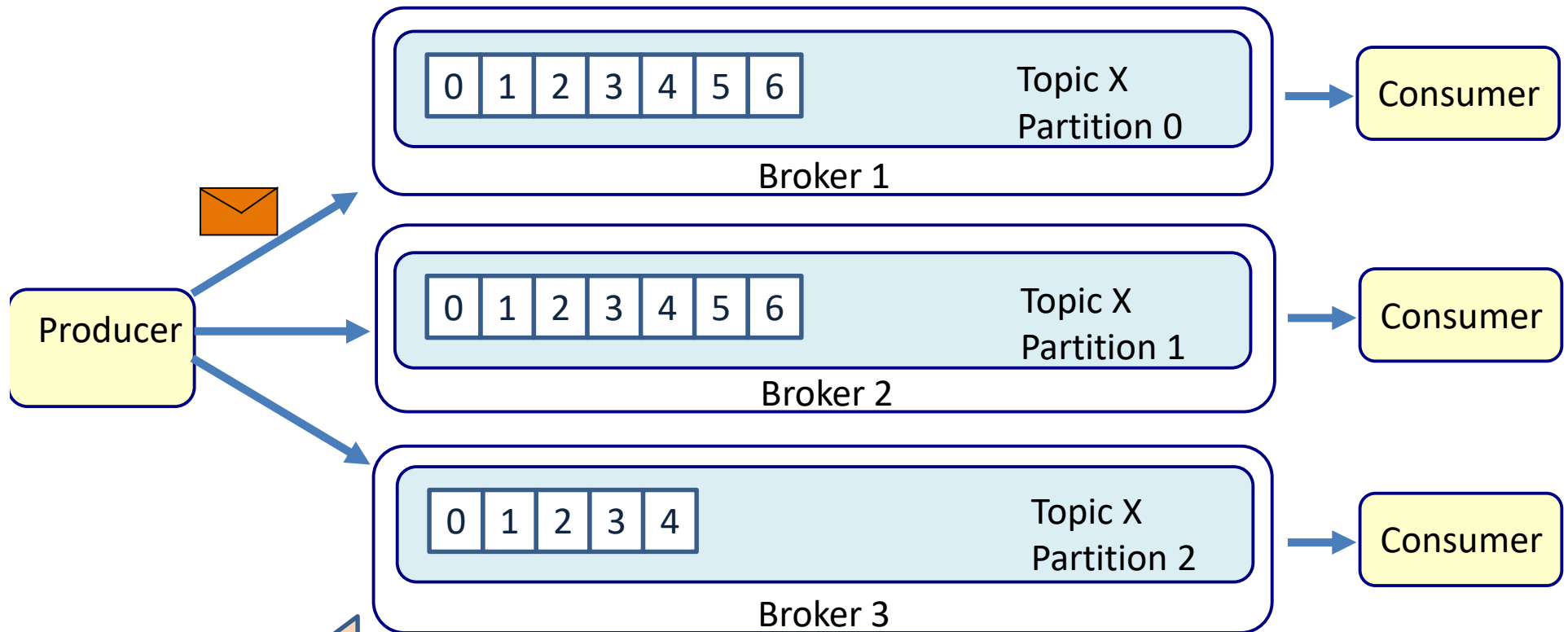
What if the topic gets too full?



Scale out partitions



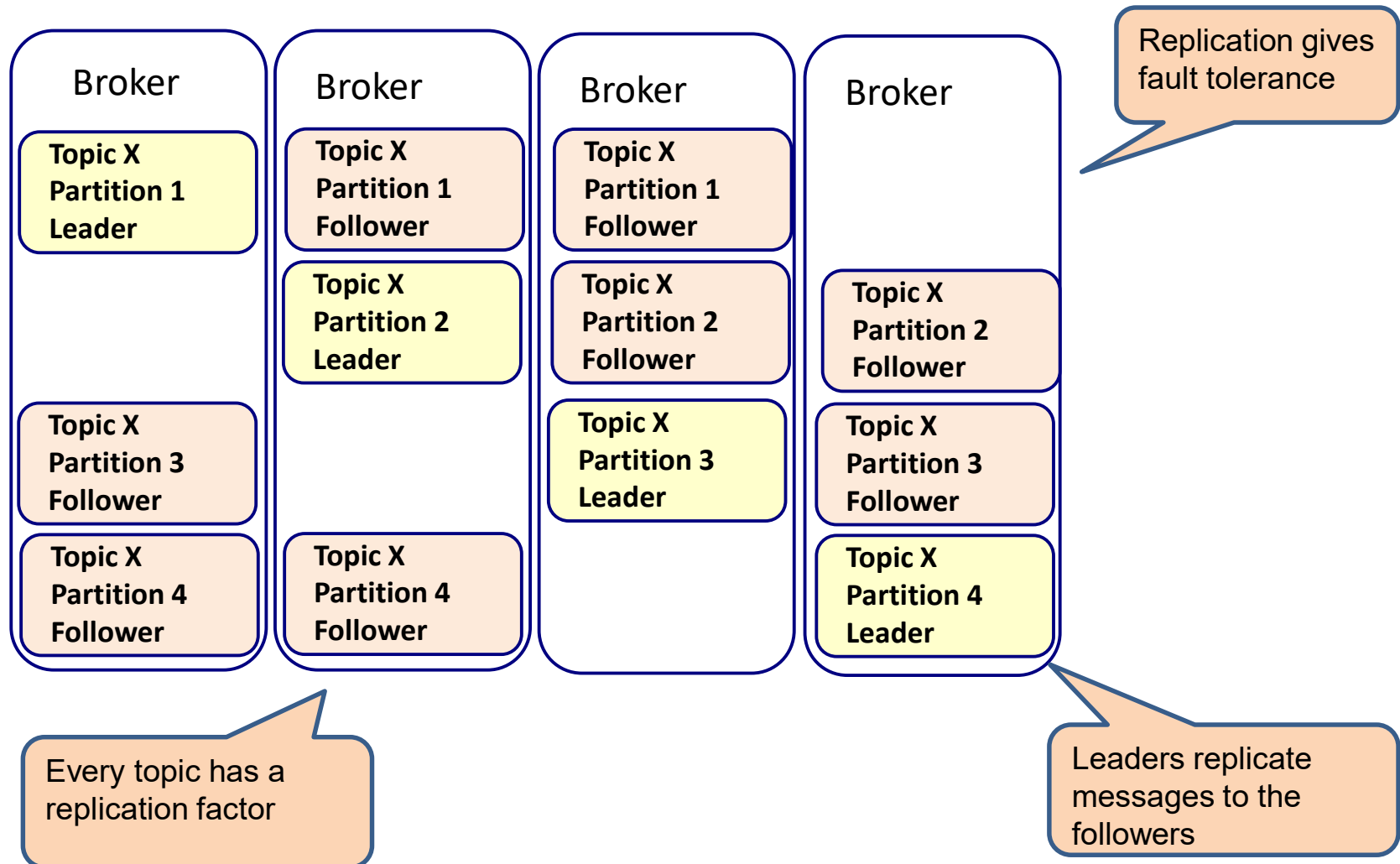
Scale out partitions



Increases

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

Replication



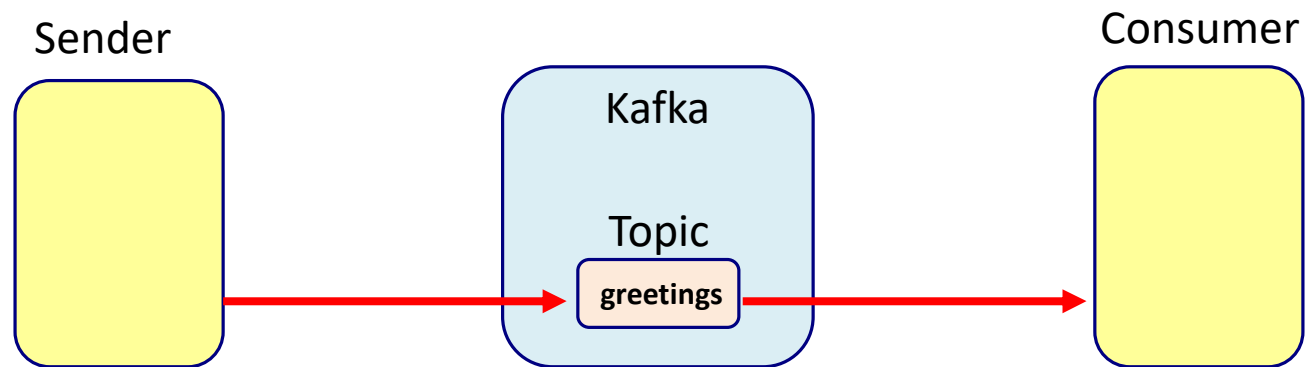
Creating a topic



```
~$ bin/kafka-topics.sh --create --topic my_topic \  
> --zookeeper localhost:2181 \  
> --partitions 3 \  
> --replication-factor 3
```

SPRING BOOT AND KAFKA

Example



SenderApplication



```
@SpringBootApplication
@EnableKafka
public class SenderApplication implements CommandLineRunner {
    @Autowired
    Sender sender;

    public static void main(String[] args) {
        SpringApplication.run(SenderApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        sender.send("topicA", "Hello World");
        System.out.println("Message has been sent");
    }
}
```

Sender

@Service

```
public class Sender {  
    @Autowired  
    private KafkaTemplate<String, String> kafkaTemplate;  
  
    public void send(String topic, String message){  
        kafkaTemplate.send(topic, message);  
    }  
}
```

application.properties

```
spring.kafka.bootstrap-servers=localhost:9092  
spring.kafka.consumer.group-id= gid  
spring.kafka.consumer.auto-offset-reset= earliest  
spring.kafka.consumer.key-deserializer= org.apache.kafka.common.serialization.StringDeserializer  
spring.kafka.consumer.value-deserializer= org.springframework.kafka.support.serializer.JsonDeserializer  
spring.kafka.producer.key-serializer= org.apache.kafka.common.serialization.StringSerializer  
spring.kafka.producer.value-serializer= org.springframework.kafka.support.serializer.JsonSerializer  
spring.kafka.consumer.properties.spring.json.trusted.packages=kafka
```

```
logging.level.root= ERROR  
org.springframework= ERROR
```

ReceiverApplication



@SpringBootApplication

@EnableKafka

public class ReceiverApplication **implements** CommandLineRunner {

```
    public static void main(String[] args) {  
        SpringApplication.run(ReceiverApplication.class, args);  
    }
```

@Override

```
    public void run(String... args) throws Exception {  
        System.out.println("Receiver is running and waiting for messages");  
    }  
}
```

Receiver

```
@Service
public class Receiver {

    @KafkaListener(topics = {"topicA"})
    public void receive(@Payload String message) {
        System.out.println("Receiver received message= "+ message);
    }
}
```

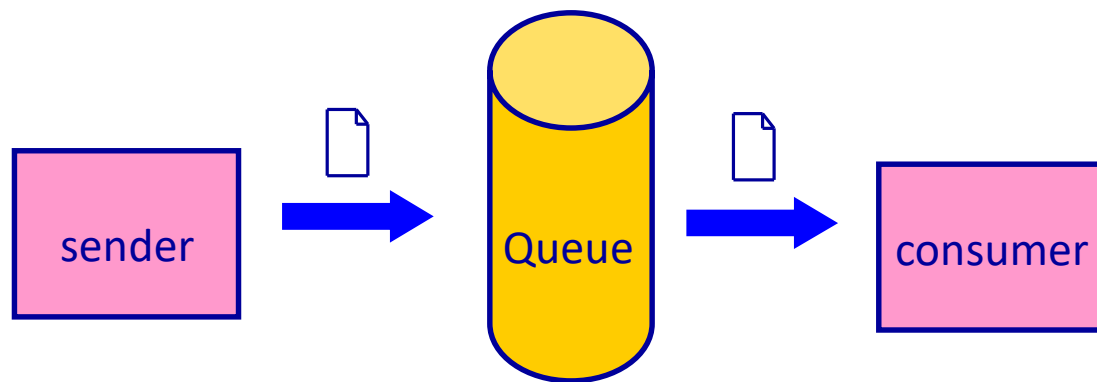
application.properties

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id= gid
spring.kafka.consumer.auto-offset-reset= earliest
spring.kafka.consumer.key-deserializer= org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-deserializer= org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.producer.key-serializer= org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer= org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.consumer.properties.spring.json.trusted.packages=kafka
```

```
logging.level.root= ERROR
org.springframework= ERROR
```

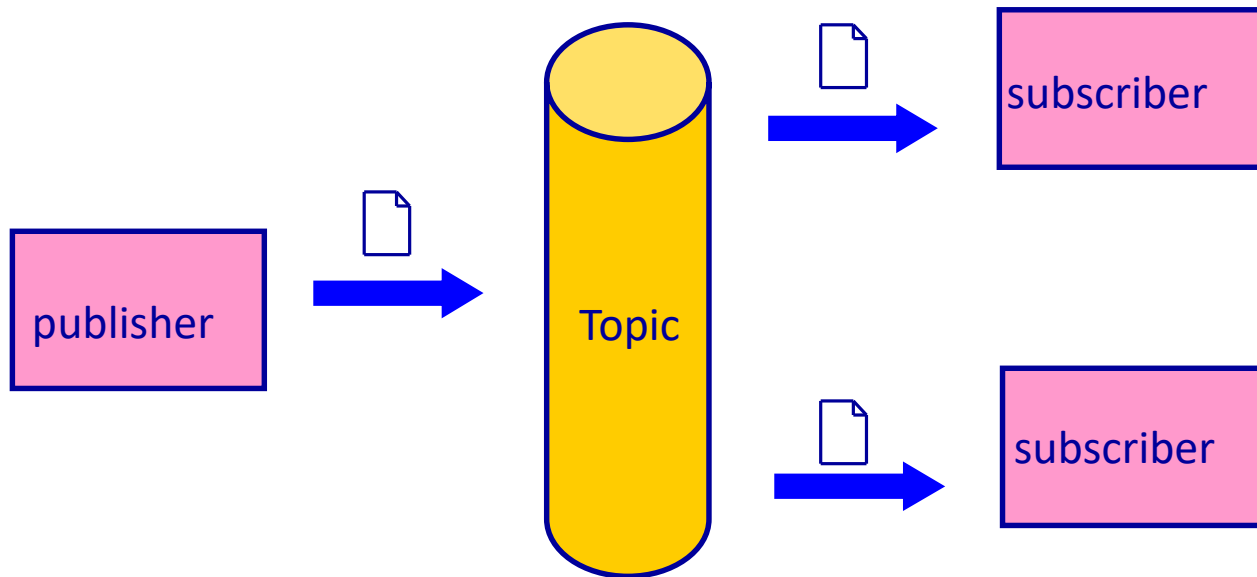

Point-To-Point (PTP)

- A dedicated consumer per Queue message



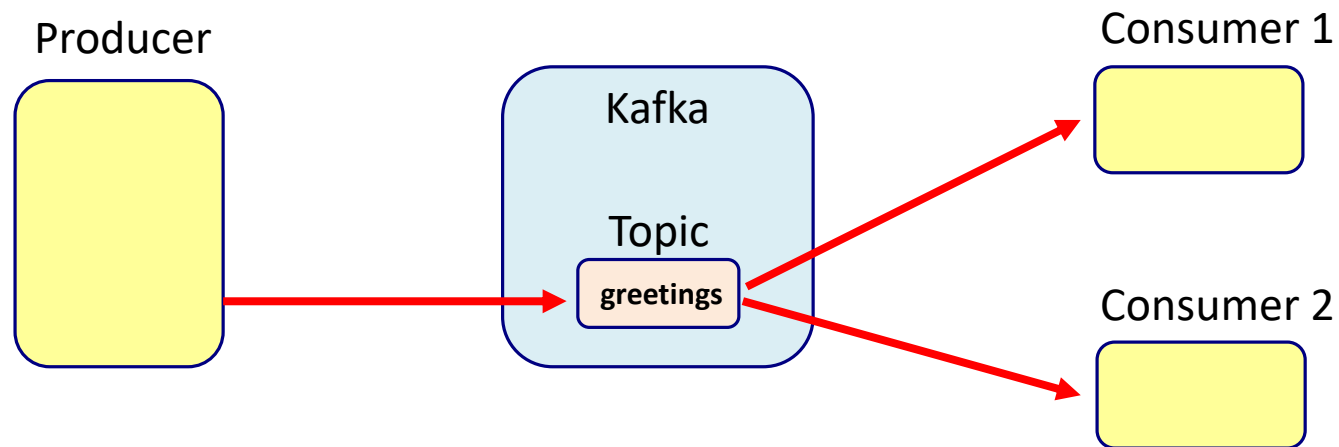
Publish-Subscribe (Pub-Sub)

- A message channel can have more than one '*consumer*'
 - Ideal for broadcasting



What if we have 2 consumers

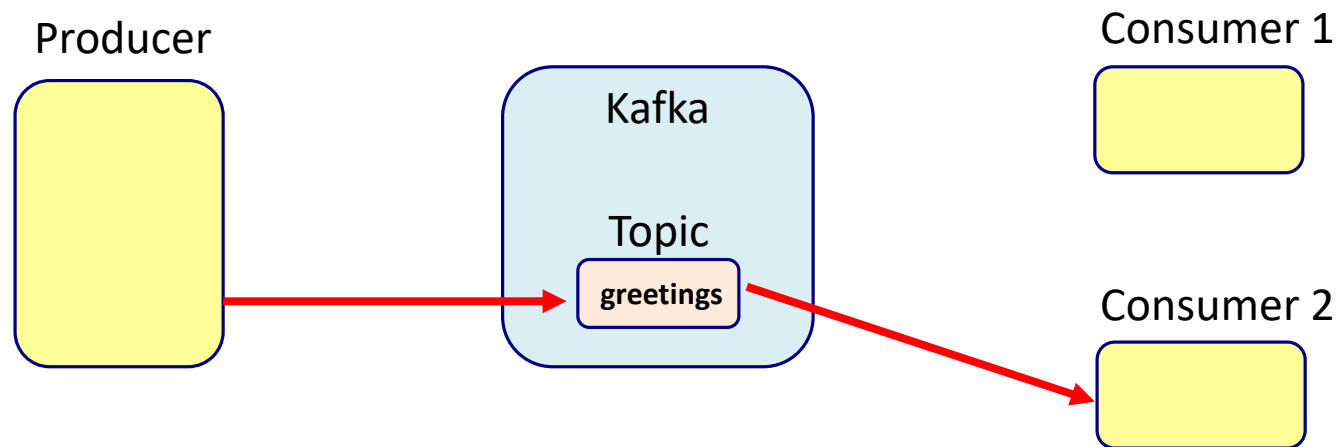
- The default behavior is pub/sub
 - Instead of point to point



- Both consumers receive the message

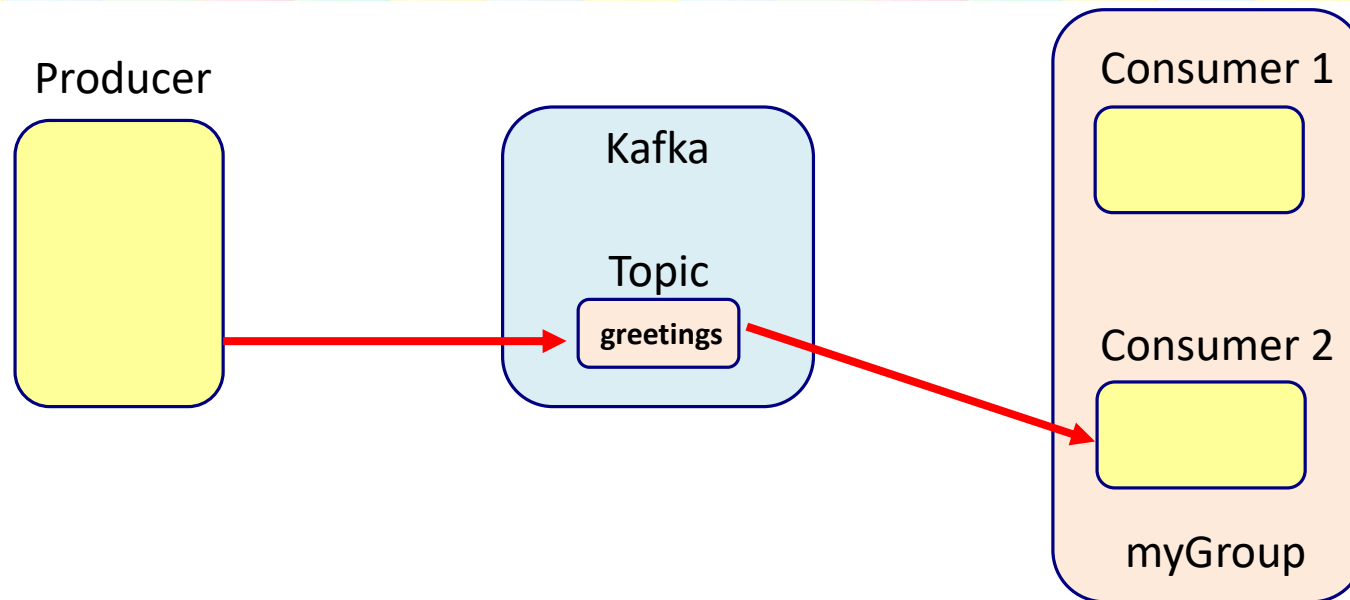
What if we want point to point

- Competing consumers



- Only one consumers receives the message

Consumer groups



application.properties

```
spring.kafka.bootstrap-servers=localhost:9092  
spring.kafka.consumer.group-id= gid  
...
```

Give both consumers
the same group-id

Send an object: Sender

@SpringBootApplication

@EnableKafka

public class OrderApplication implements CommandLineRunner {

@Autowired

Sender sender;

public static void main(String[] args) {

SpringApplication.run(OrderApplication.class, args);

}

@Override

public void run(String... args) throws Exception {

sender.send("ordertopic", new Order("A1276", LocalDate.now().toString(), 1200.0));

System.out.println("Order has been sent");

}

}

public class Order {

private String orderNumber;

private String date;

private double amount;

Sender

@Service

```
public class Sender {  
    @Autowired  
    private KafkaTemplate<String, Order> kafkaTemplate;  
  
    public void send(String topic, Order order){  
        kafkaTemplate.send(topic, order);  
    }  
}
```

application.properties

```
spring.kafka.bootstrap-servers=localhost:9092  
spring.kafka.consumer.group-id= gid  
spring.kafka.consumer.auto-offset-reset= earliest  
spring.kafka.consumer.key-deserializer= org.apache.kafka.common.serialization.StringDeserializer  
spring.kafka.consumer.value-deserializer= org.springframework.kafka.support.serializer.JsonDeserializer  
spring.kafka.producer.key-serializer= org.apache.kafka.common.serialization.StringSerializer  
spring.kafka.producer.value-serializer= org.springframework.kafka.support.serializer.JsonSerializer  
spring.kafka.consumer.properties.spring.json.trusted.packages=kafka
```

```
logging.level.root= ERROR  
org.springframework= ERROR
```

Receiver Application

```
@SpringBootApplication
@EnableKafka
public class OrderApplication {

    public static void main(String[] args) {
        SpringApplication.run(OrderApplication.class, args);
    }

}
```

```
public class Order {
    private String orderNumber;
    private String date;
    private double amount;
}
```


Receiver

@Service

```
public class Receiver {
```

```
    @KafkaListener(topics = {"ordertopic"})
```

```
    public void receive(@Payload Order order) {
```

```
        System.out.println("OrderReceiver 1 received order="+ order);
```

```
    }
```

```
}
```

```
spring.kafka.bootstrap-servers=localhost:9092
```

```
spring.kafka.consumer.group-id= gid
```

```
spring.kafka.consumer.auto-offset-reset= earliest
```

```
spring.kafka.consumer.key-deserializer= org.apache.kafka.common.serialization.StringDeserializer
```

```
spring.kafka.consumer.value-deserializer= org.springframework.kafka.support.serializer.JsonDeserializer
```

```
spring.kafka.producer.key-serializer= org.apache.kafka.common.serialization.StringSerializer
```

```
spring.kafka.producer.value-serializer= org.springframework.kafka.support.serializer.JsonSerializer
```

```
spring.kafka.consumer.properties.spring.json.trusted.packages=kafka
```

application.properties

```
logging.level.root= ERROR
```

```
org.springframework= ERROR
```

STREAM BASED ARCHITECTURE

Stream based systems

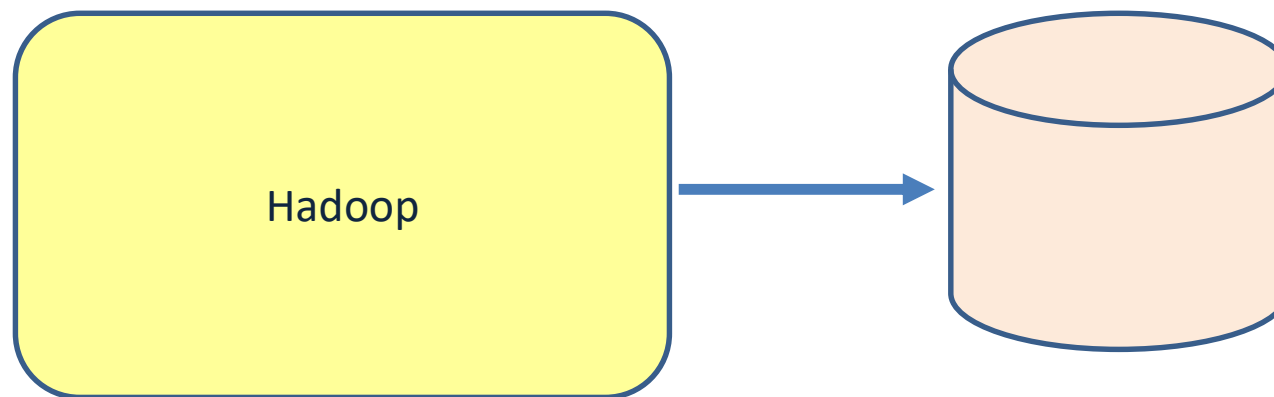
- Continuous stream of data
 - Stock market systems
 - Social networking systems
 - Internet of Things (IoT) systems
 - Systems that handle sensor data
 - System that handle logfiles
 - Systems that monitor user clicks
 - Car navigator software

But also

- Stream of purchases in web shop
- Stream of transactions in a bank
- Stream of actions in a multi user game
- Stream of bookings in a hotel booking system
- Stream of user actions on a web application
- ...

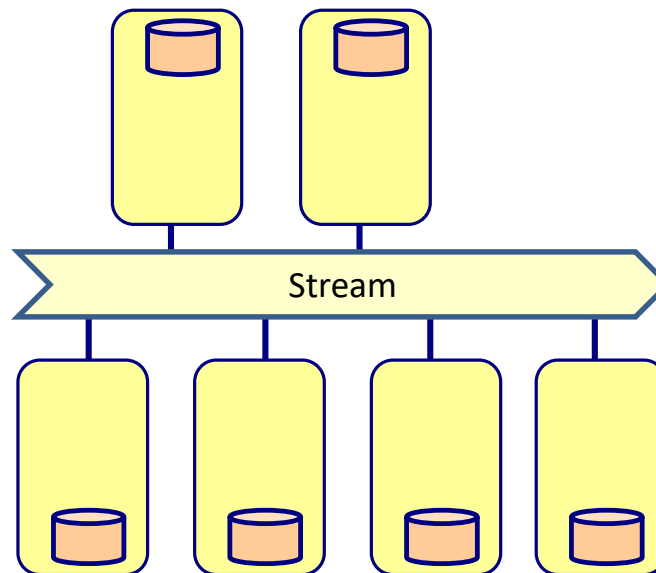
Batch processing

- 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

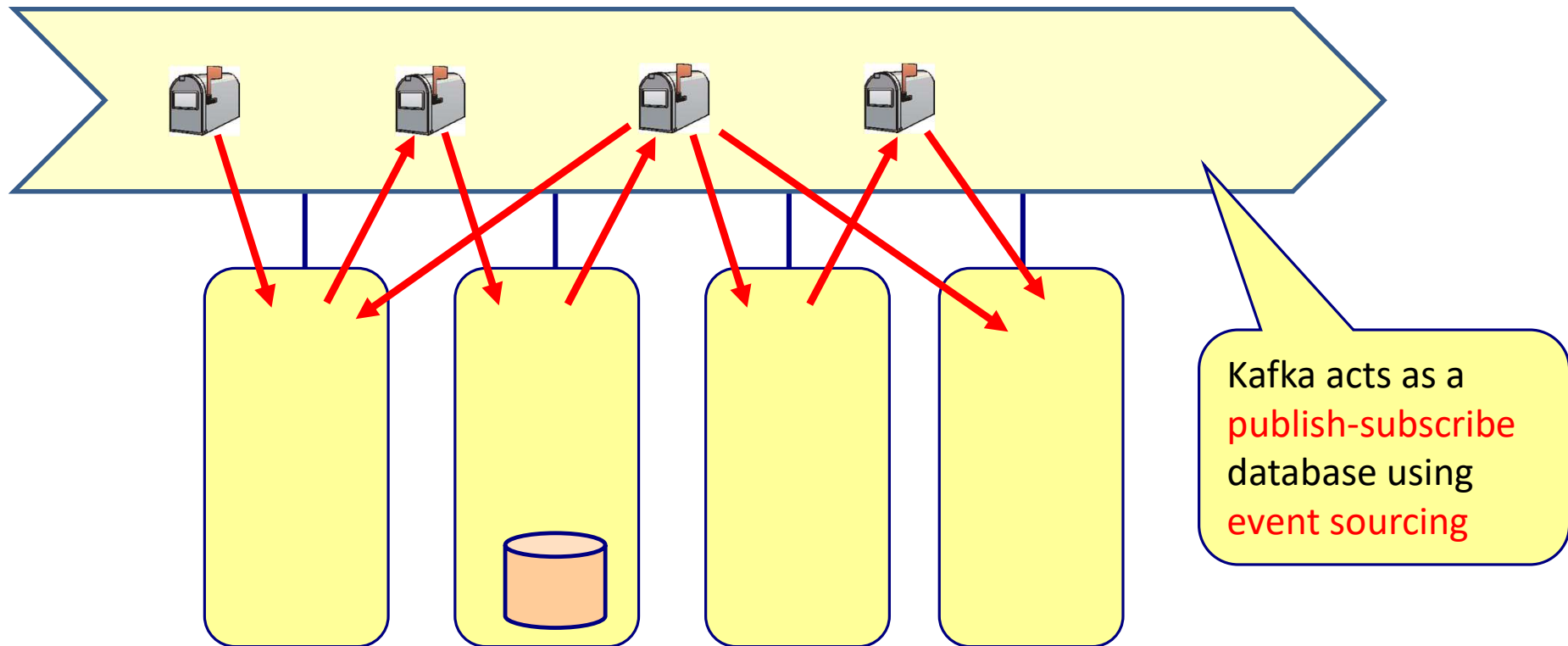


Stream processing

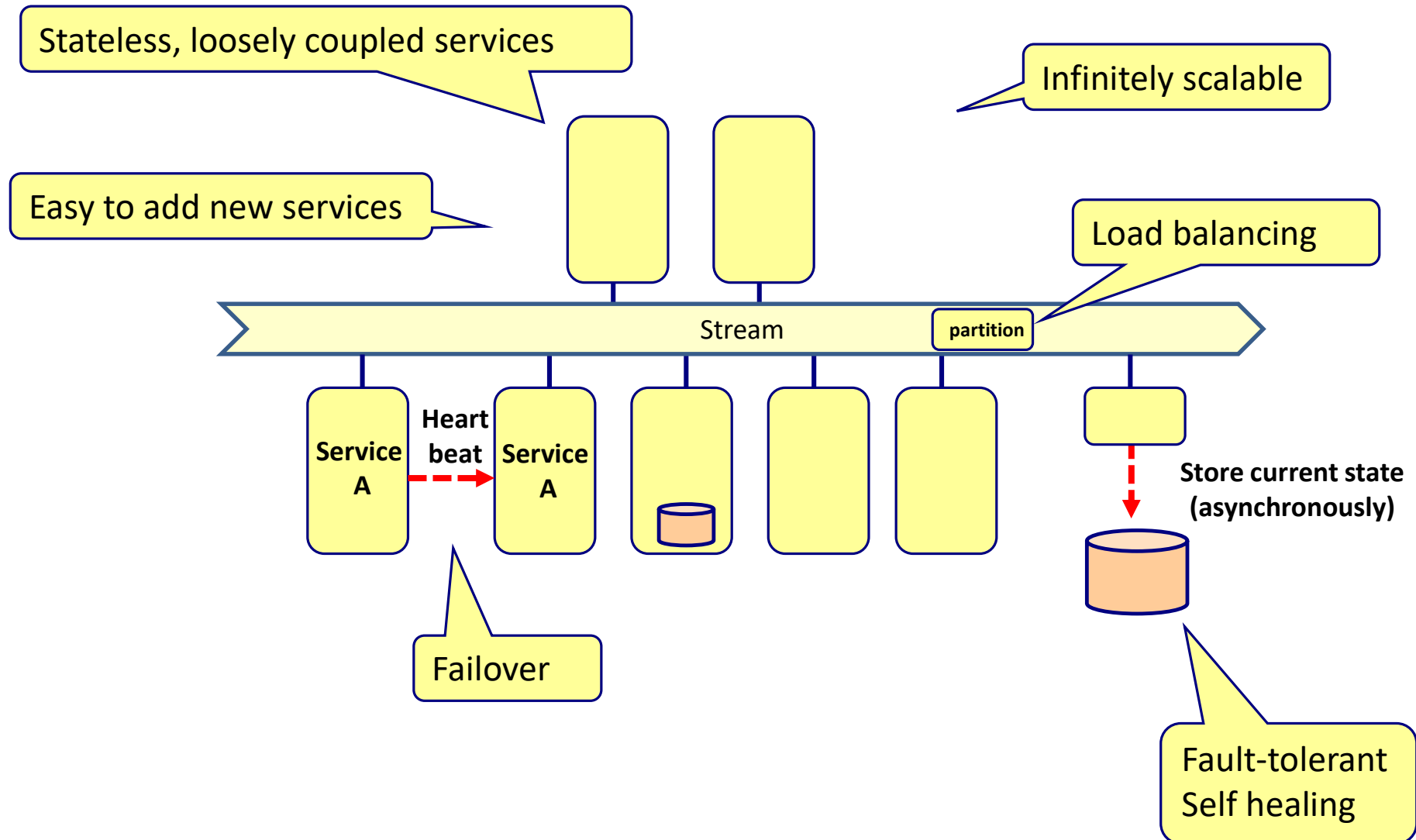
- Handle the data when it arrives
- Handle event (small data) by event
- Latencies in seconds or milliseconds



Publish-subscribe and event sourcing

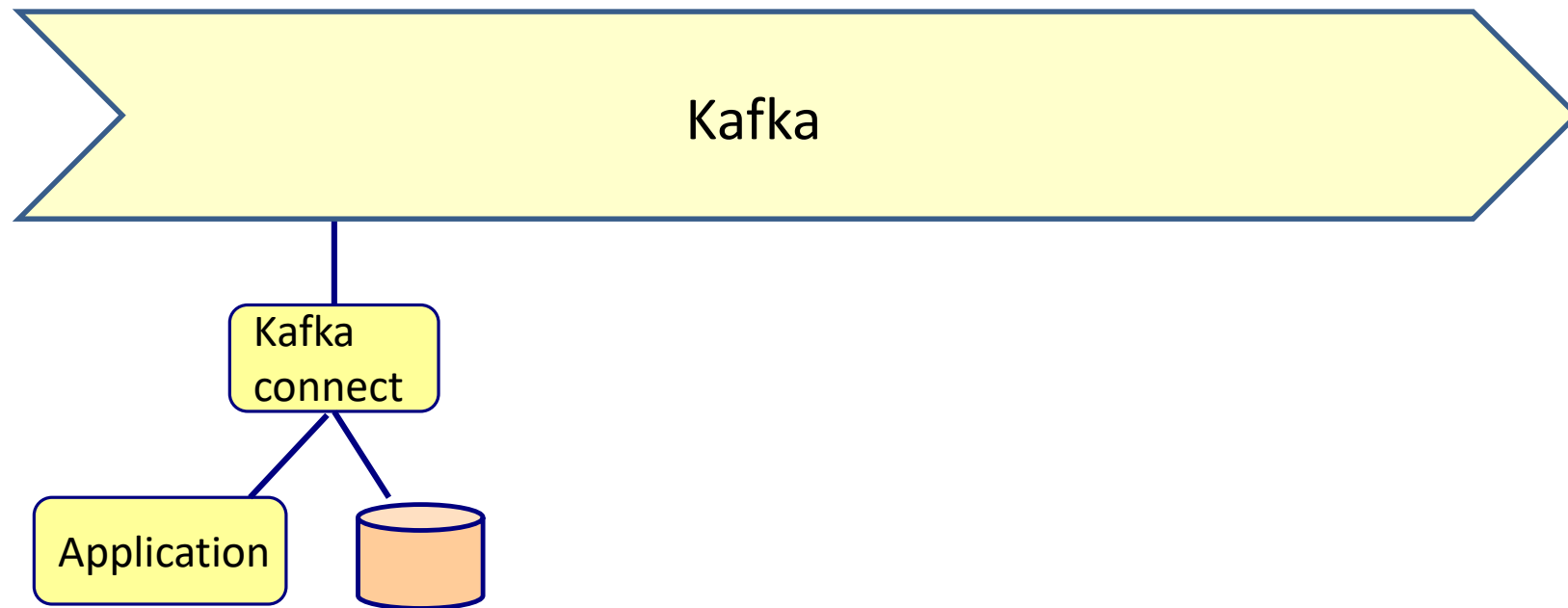


Stream based architecture

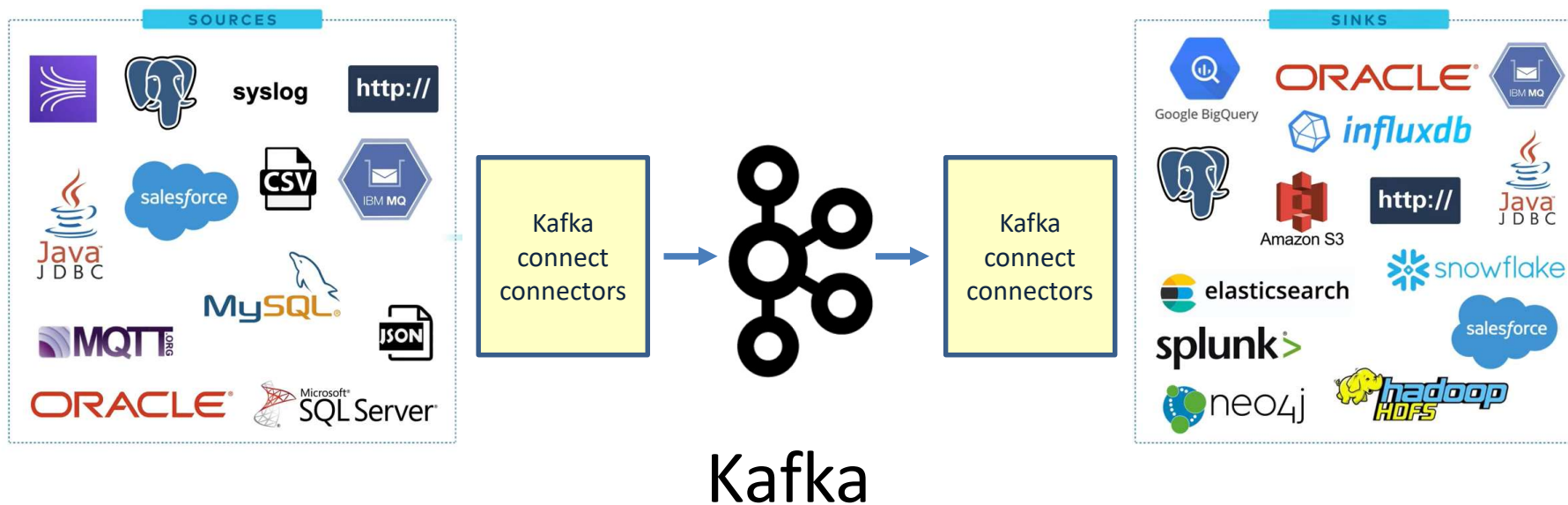


KAFKA ECOSYSTEM

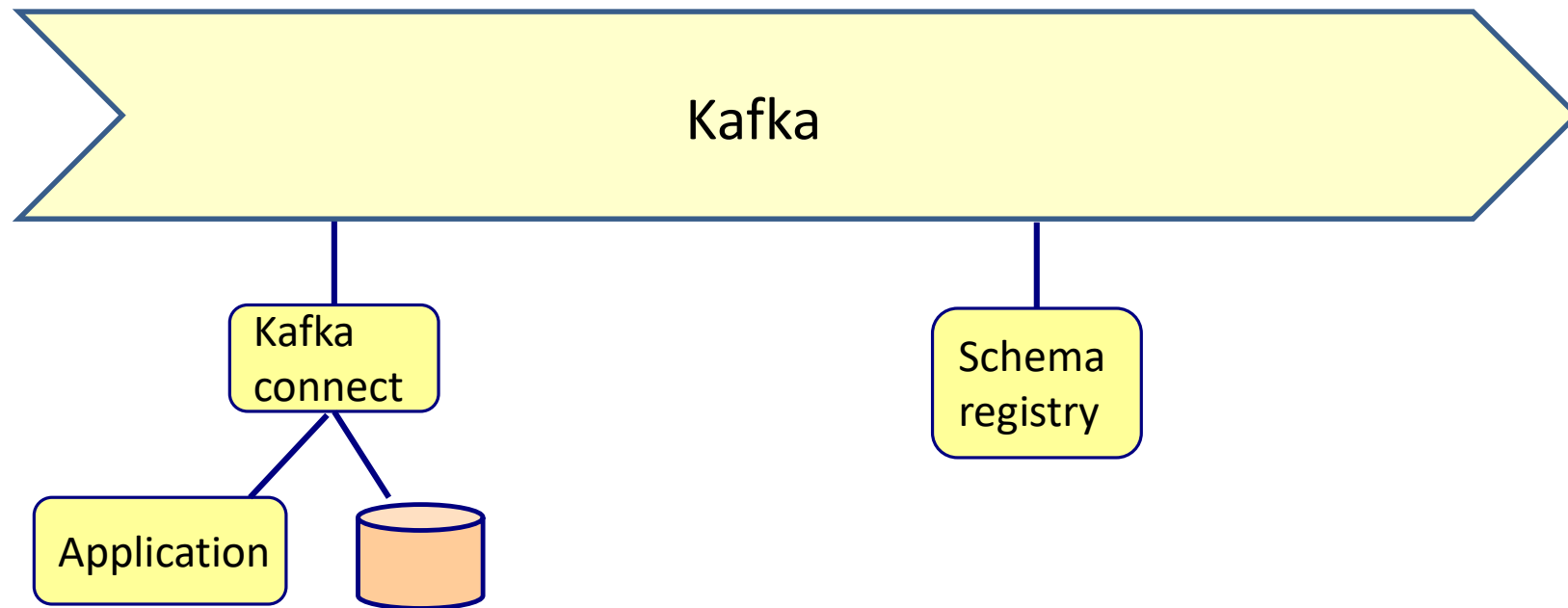
Kafka ecosystem: Kafka connect



Kafka connect



Kafka ecosystem: Schema registry

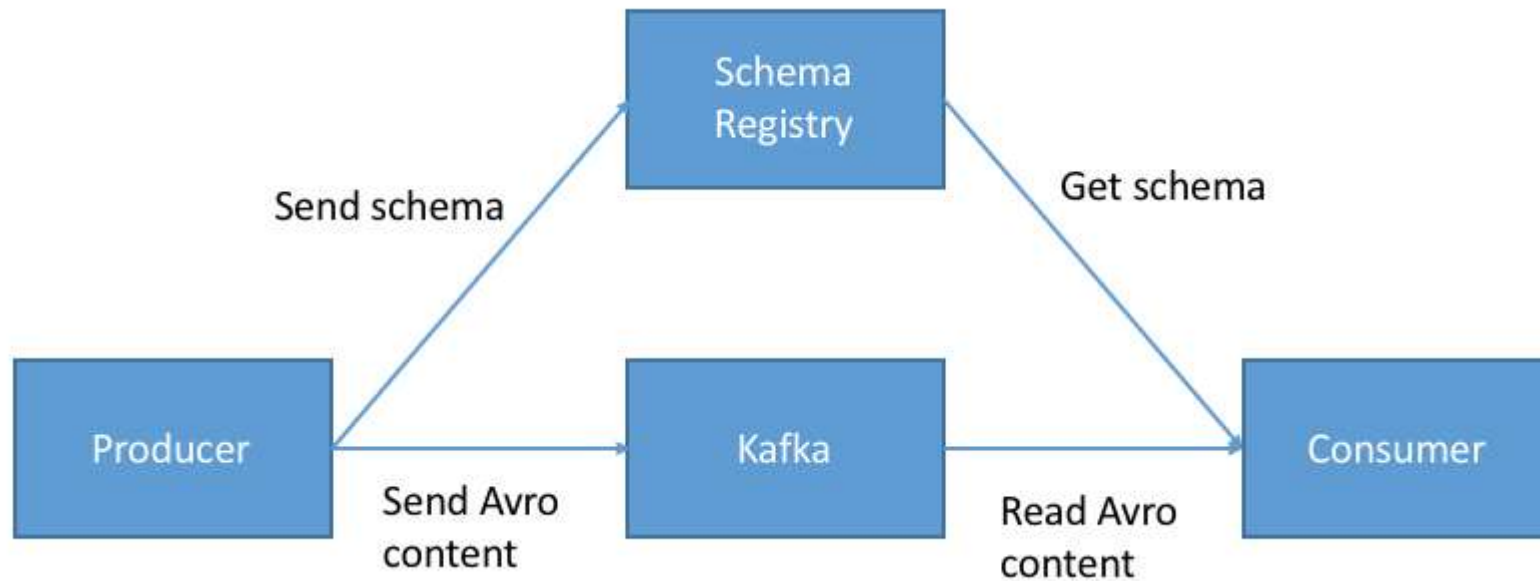


Need for a schema registry

- What if the producer sends bad data?
- What if a field gets renamed?
- What if the data format changes ?
- The consumer breaks

Kafka does not verify the message

- Schema registry is a separate component (server)
- Maintains a database of schema's

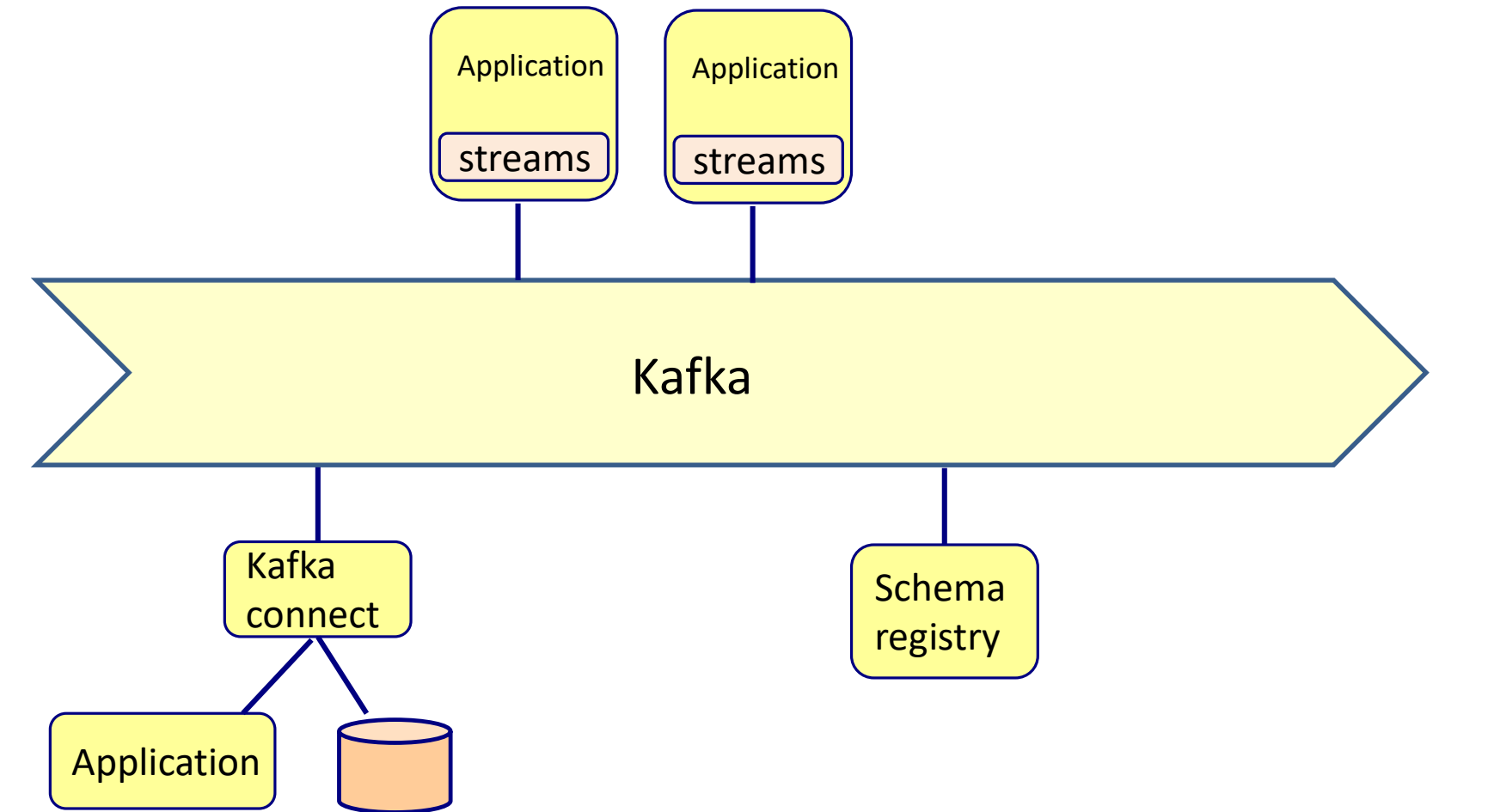


Avro

- JSON + schema
 - Data is fully typed
 - Schema is in the data
 - Schema can evolve over time

```
{
  "namespace": "example.avro",
  "type": "record",
  "name": "user",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "favorite_number", "type": "int" }
  ]
}
```

Kafka ecosystem: Kafka streams



Kafka streams

- Java library for making stream processing simpler
 - Simple concise code
 - Threading and parallelism
 - Stream DSL (map, filter, aggregations, joins,...)

Kafka security

- Authentication
 - Are you allowed to access kafka?
 - SSL & SASL
 - Using certificates
- Authorization
 - Who is allowed to publish or consume which topic?
 - Access Control Lists (ACL)
- Encryption
 - Data sent is not readable by others
 - SSL
 - Only inflight security