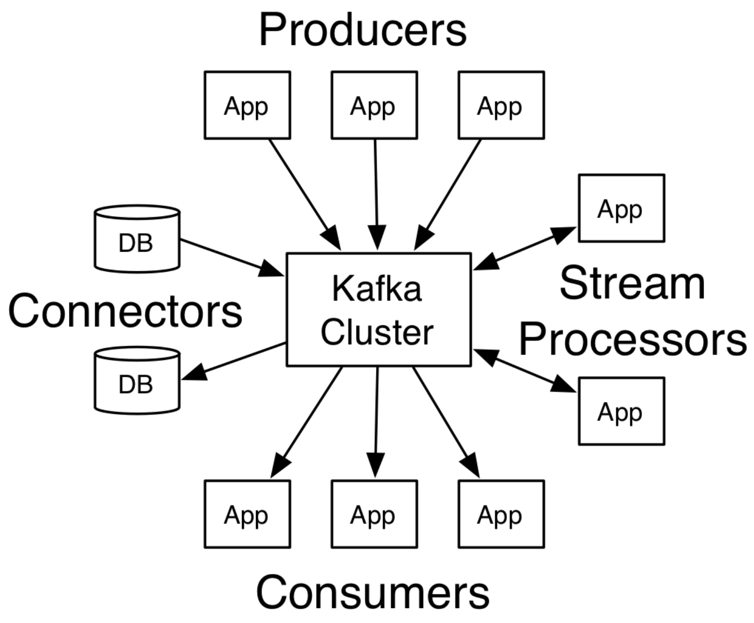
POLITECNICO
MILANO 1863

Apache Kafka

Alessandro Margara
alessandro.margara@polimi.it
https://margara.faculty.polimi.it

Event-based communication

- Architectural style to build distributed systems
- Components only interact by exchanging messages / events
  - Producers publish notifications of the events they observe
    - E.g., Security camera observes a person entering the room
  - Consumers subscribe to the events they are interested in
    - E.g., Security dashboard subscribes to all security-related events
- Producers and consumers need not know each other
  - The communication is mediated by a middleware service such as Kafka



Event-based communication

- Widely used to build large-scale dynamic systems, but why?
  - Space decoupling
    - Producers and consumers do not need to know each other
    - You can dynamically add new components without restarting/reconfiguring the system
      - E.g., a new sensor joins the network and produces new events
  - Synchronization decoupling
    - Producers are not blocked while producing events and consumers are notified asynchronously
    - Promotes scalability by removing explicit dependencies between producers and consumers
  - Time decoupling
    - If the event middleware can store events, producers and consumers need not be connected at the same time

Apache Kafka

- Apache Kafka is probably the most widely used platform for *event-based communication* today

- From the Website: "More than 80% of all Fortune 100 companies trust and use Kafka"
  – LinkedIn, Netflix, New York Times, Zalando, AirBnB, CISCO, Uber, PayPal, Spotify, Strava, Twitter, …

- If you are interested in some use cases
  – https://kafka.apache.org/uses
  – https://www.confluent.io/blog/category/use-cases/
  – https://www.confluent.io/designing-event-driven-systems/

Apache Kafka

- Originally developed by LinkedIn to efficiently handle logs
- Open sourced in 2011
- Apache project since 2012
- In 2014, a group of developers who originally worked on Kafka at LinkedIn created a new company named Confluent
  - Develops Kafka
  - Offers a commercial platform that builds on Kafka
    - Additional management features
    - Solutions for Kafka-as-a-service
  - Since early 2021, Confluent is listed on Nasdaq

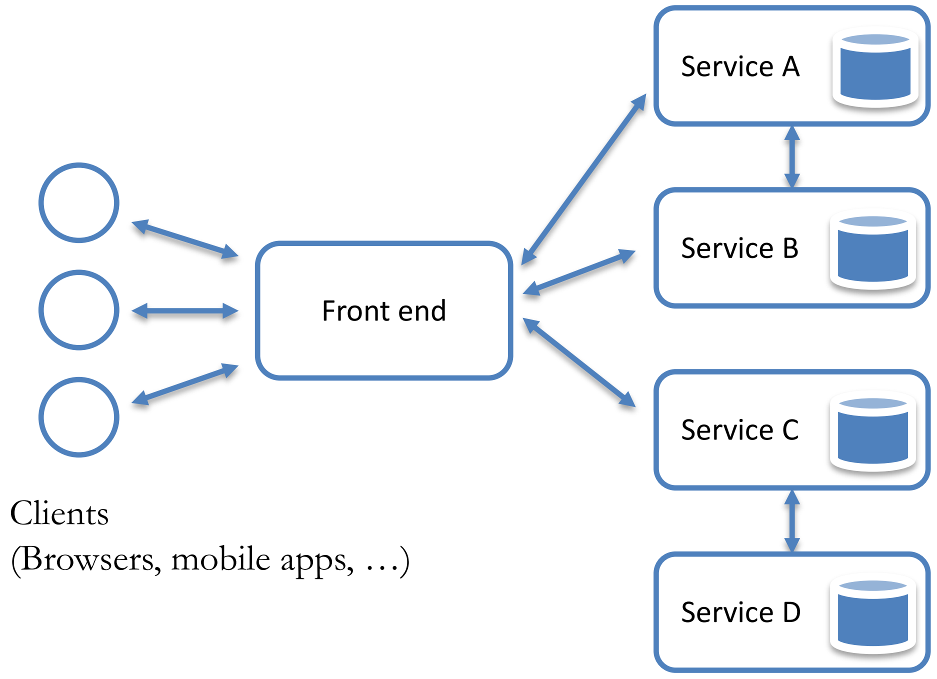
Apache Kafka overview

- Topic-based communication
  - Messages/events are put into Kafka queues/logs, each of them representing a topic
- Topics are persistent
  - Topics are stored on durable storage (disk) and replicated for fault-tolerance
- Consumers can read the same events more than once
  - Useful in the case of failure: a component can resume its state by replaying the history of events
- Scalability
  - Topics are partitioned to improve access performance and scalability

Case study

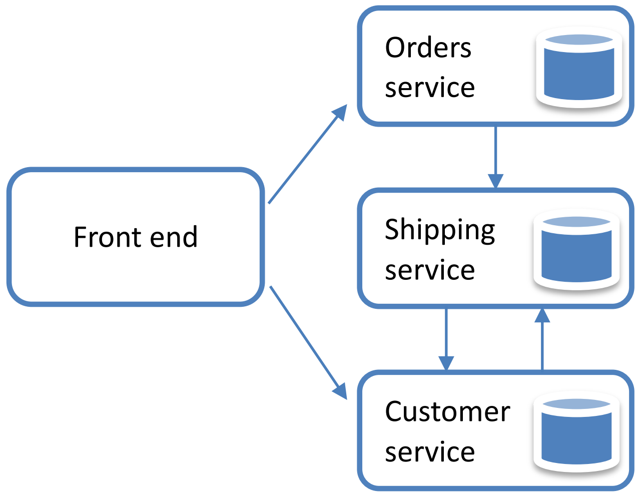
Event-based communication in microservices architectures

Case study: microservices architectures



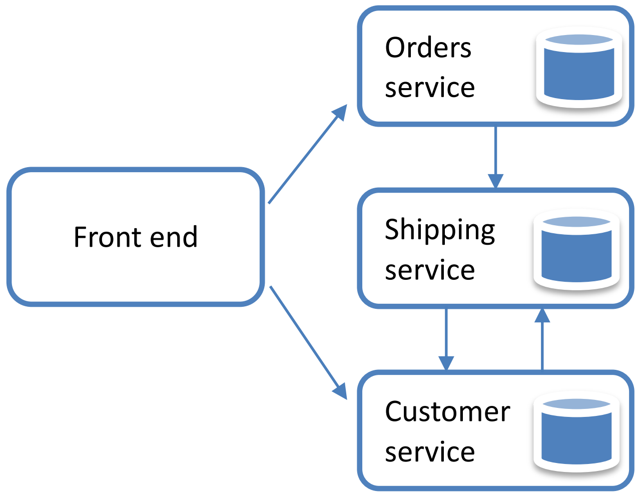
- Architectural paradigm where an application is split into small independent services
- No shared-state
  - Each service has its own local state (database)
  - Services communicate only by RPC or by exchanging events

Case study: microservices architectures



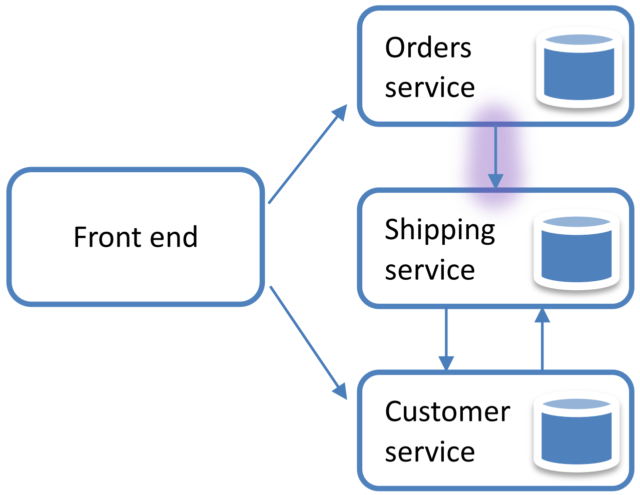
- Orders processes orders from customers
- Orders are sent to the shipping service
- The shipping service queries the customer service for the address of the customer

Case study: microservices architectures



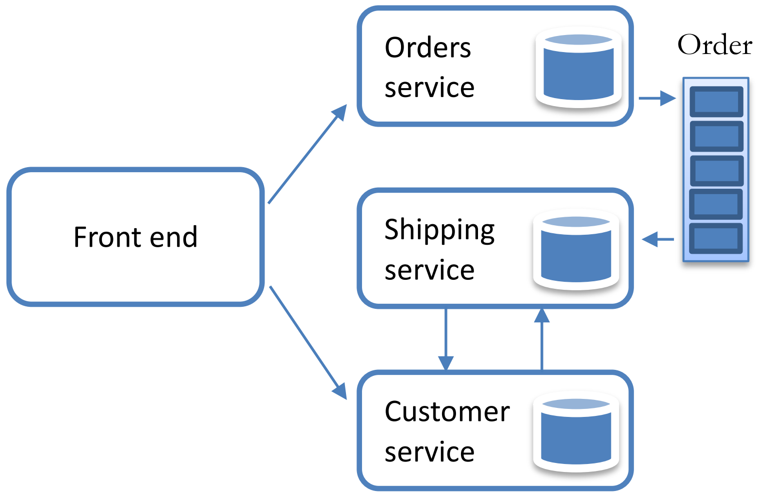
- The orders service invokes a command on the shipping service
  - Changes the state by adding a new order
- The shipping service queries the customer service

Case study: microservices architectures



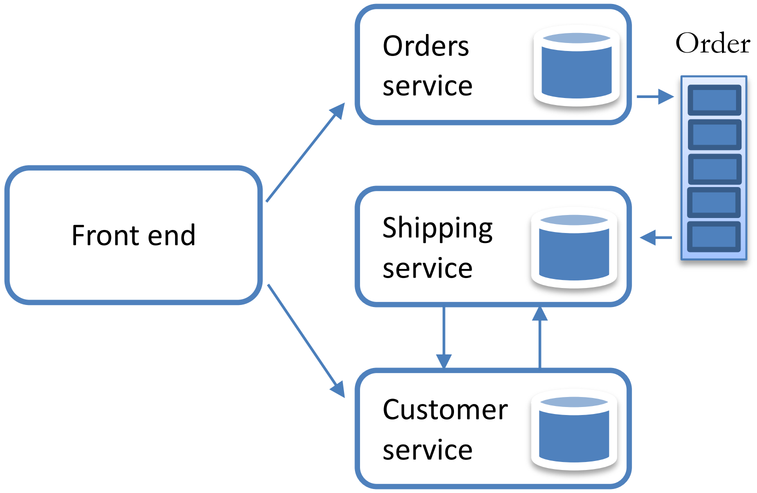
- Focus on the interaction between the orders service and the shipping service
- We can use Kafka to mediate the communication
  - Instead of invoking a command on the shipping service …
  - … we register a new order event on the order topic

Case study: microservices architectures



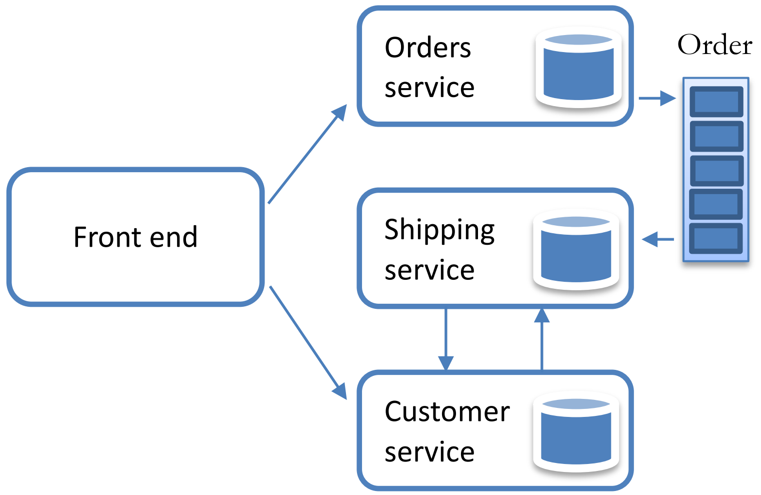
- The two services are now decoupled!
- This brings several advantages

Case study: microservices architectures



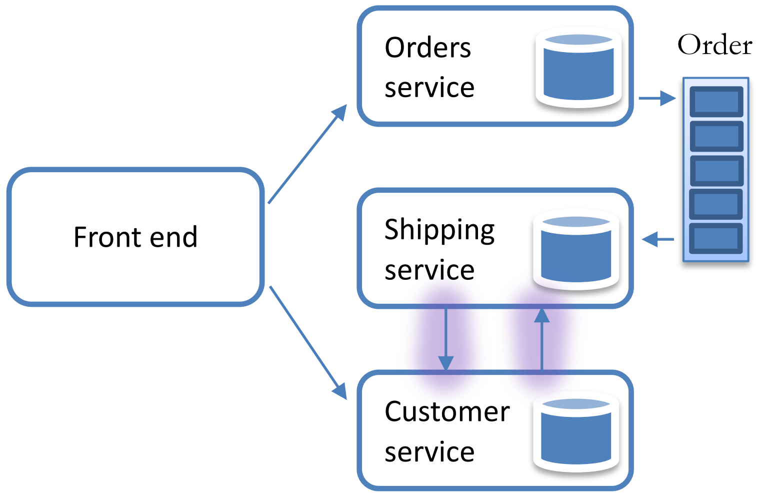
- Orders does not need to know which services are consuming order events
  - We can change shipping service without Orders even noticing
  - We can add other services that consume order events without any change to the Orders service
    - E.g., a service that computes statistics about orders

Case study: microservices architectures



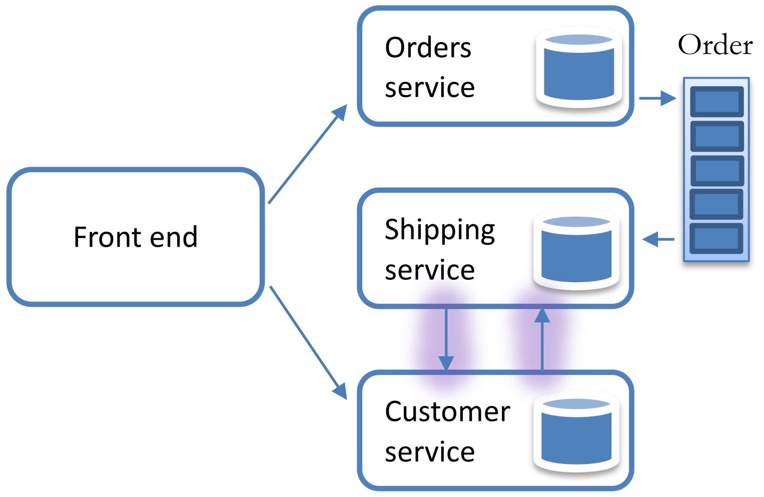
- The two services are decoupled in time
  - Shipping might be unavailable for a while …
  - … maybe because of maintenance or rollout of a new versions
  - When Shipping comes back online, it processes orders from the log
  - Orders can continue without noticing the temporary unavailability of Shipping

Case study: microservices architectures



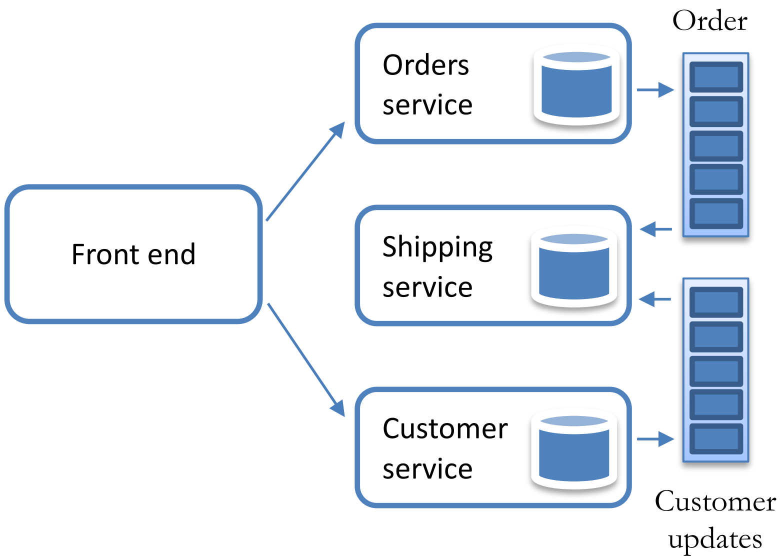
- Now, let us focus on the interaction between Shipping and Customer
  - Shipping queries Customer for the address of customers before shipping an order
  - This is a synchronous query-response interaction …
  - … but synchronous interactions may negatively affect latency

Case study: microservices architectures



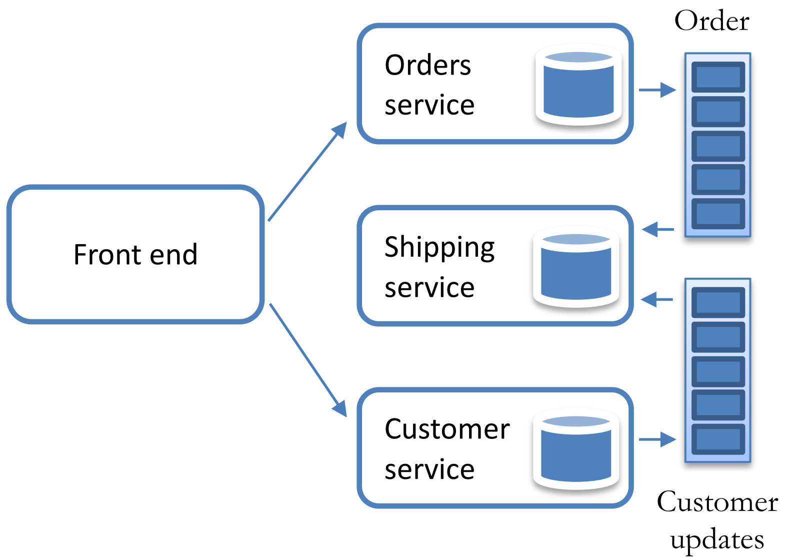
- Using an event-based paradigm, we can solve this problem by reversing the interaction between the services
  - Shipping does not query Customer
  - But it observes notifications of state (customer) changes from the Customer service

Case study: microservices architectures



- We can replicate the address of customers in the local database of the Shipping service
- When customers change their detail, the Customer service places a notification of change in the Customer updates topic
- Every interested service can read from Customer update
  - In particular, the Shipping service can update its local view of customers (address info)

Case study: microservices architectures



- Final design
- Benefits
  - Services are fully decoupled
  - No synchronous interaction
  - No problem in the case of unavailability of a service
  - Promotes evolution: add, remove, change services without other services even noticing
- Problem
  - Possible temporary inconsistency of Customer DB and Shipping DB

Case study: microservices architectures

- The above example introduces two key concepts
  - Event sourcing: events become the core element of the system
    - "Making events the source of truth" (rather than state)
    - "Turning the database inside-out"
  - Command query responsibility segregation (CQRS)
    - Separate the write path (commands) from the read path (queries)
    - Connects them through an asynchronous communication channel
      - An event log such as Apache Kafka