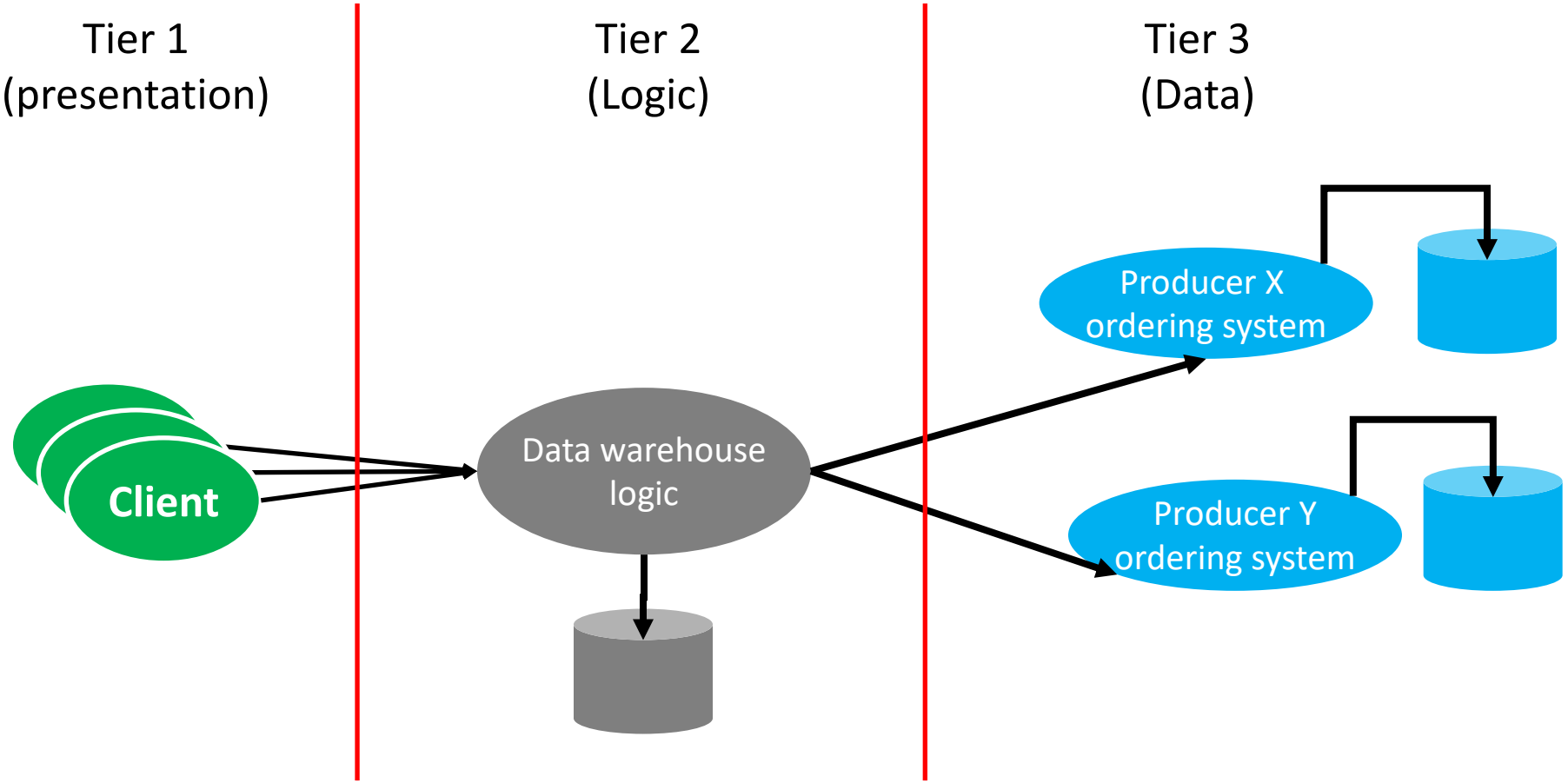
POLITECNICO
MILANO 1863

Architectural styles part 2

Three-tier Architecture – an example

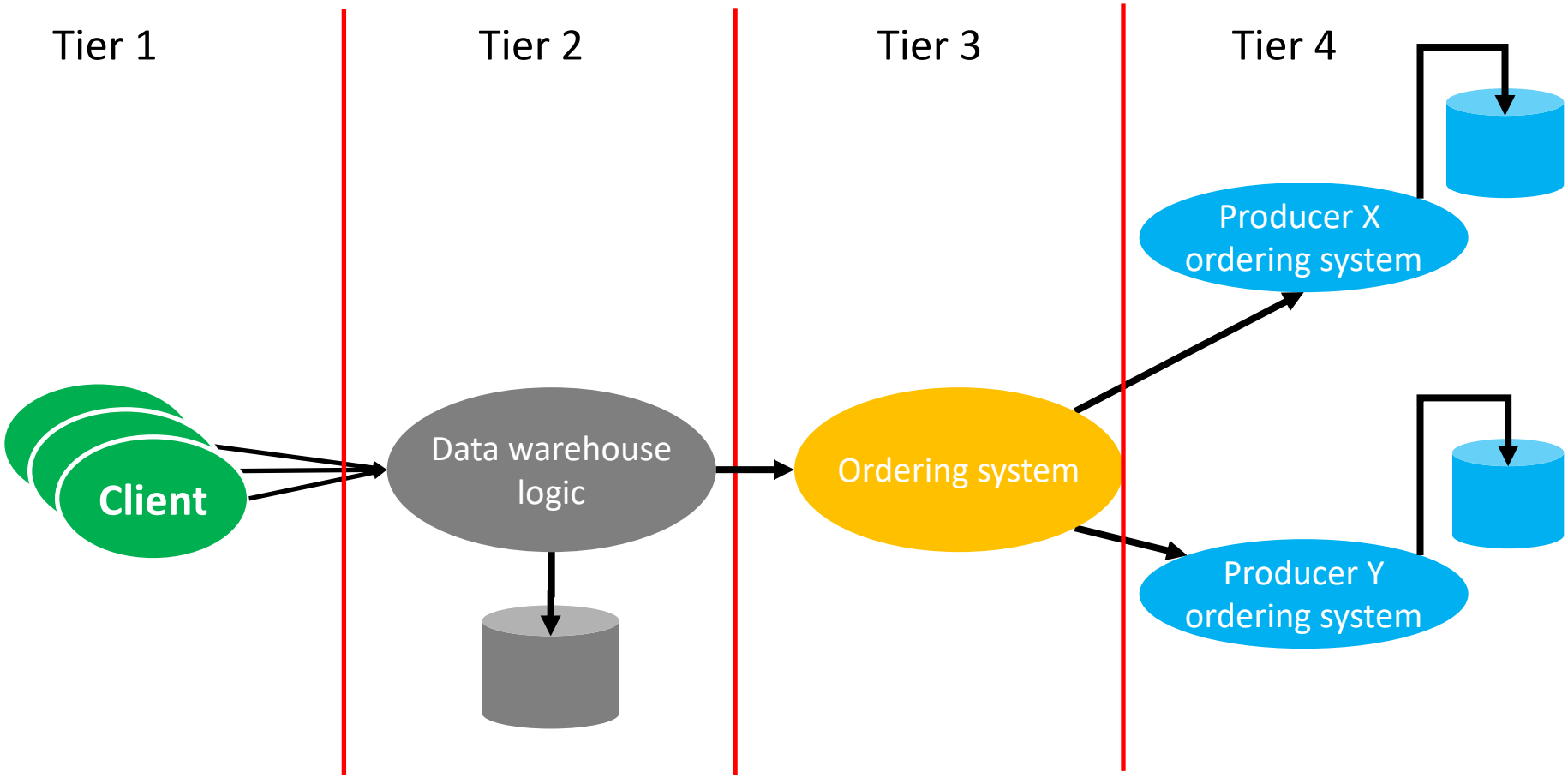
Data warehouse for a supermarket



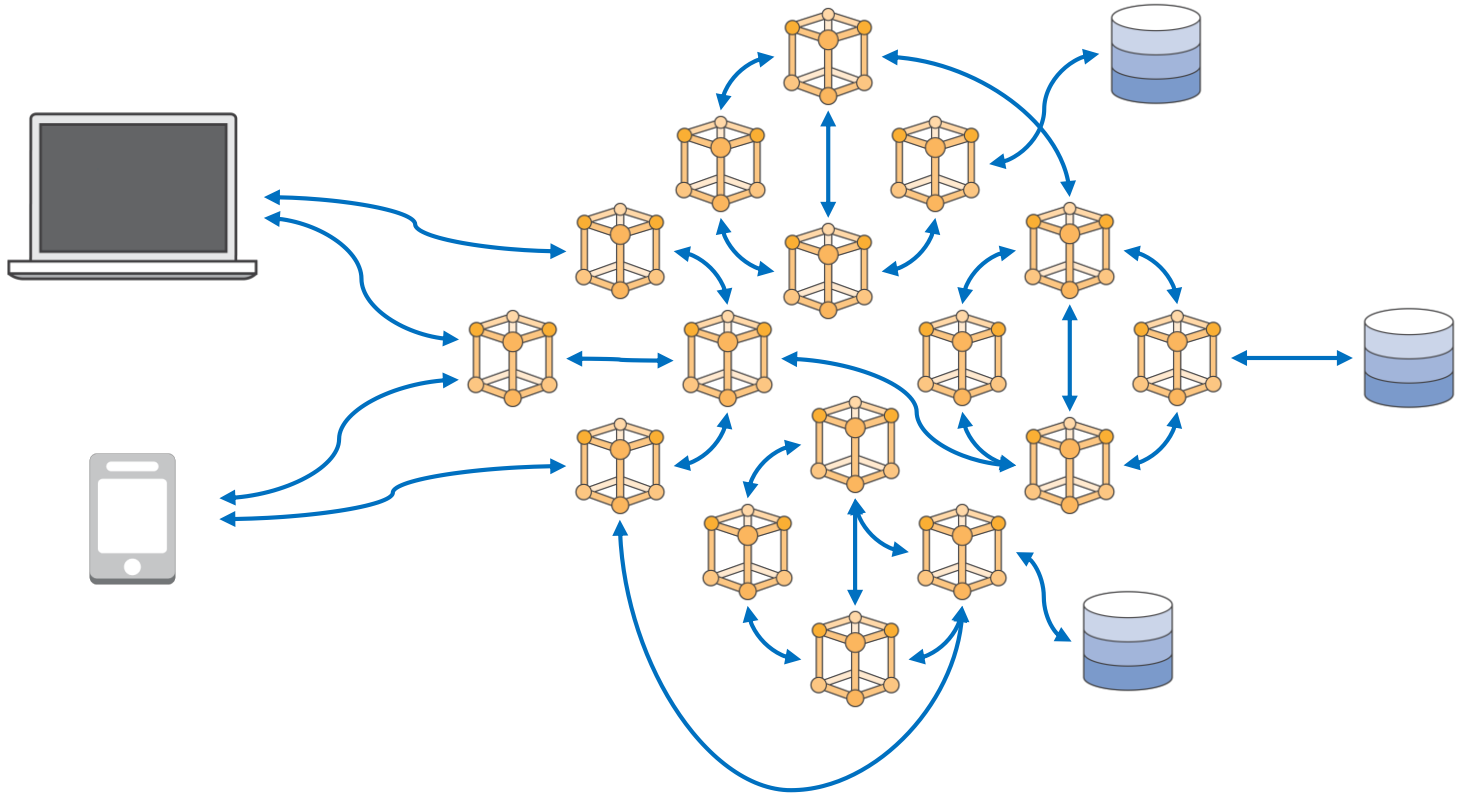
From 3 to N tiers

POLITECNICO MILANO 1863

Data warehouse for a supermarket



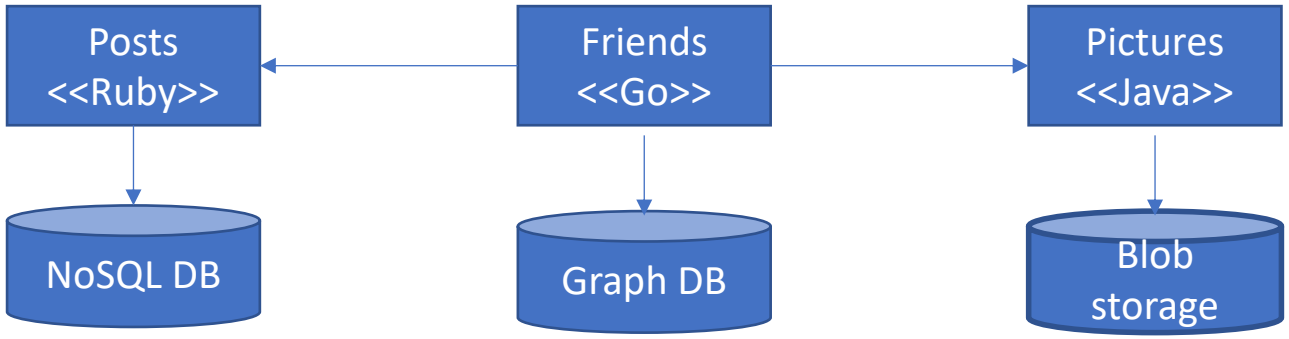
Microservice architectural style



*"...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

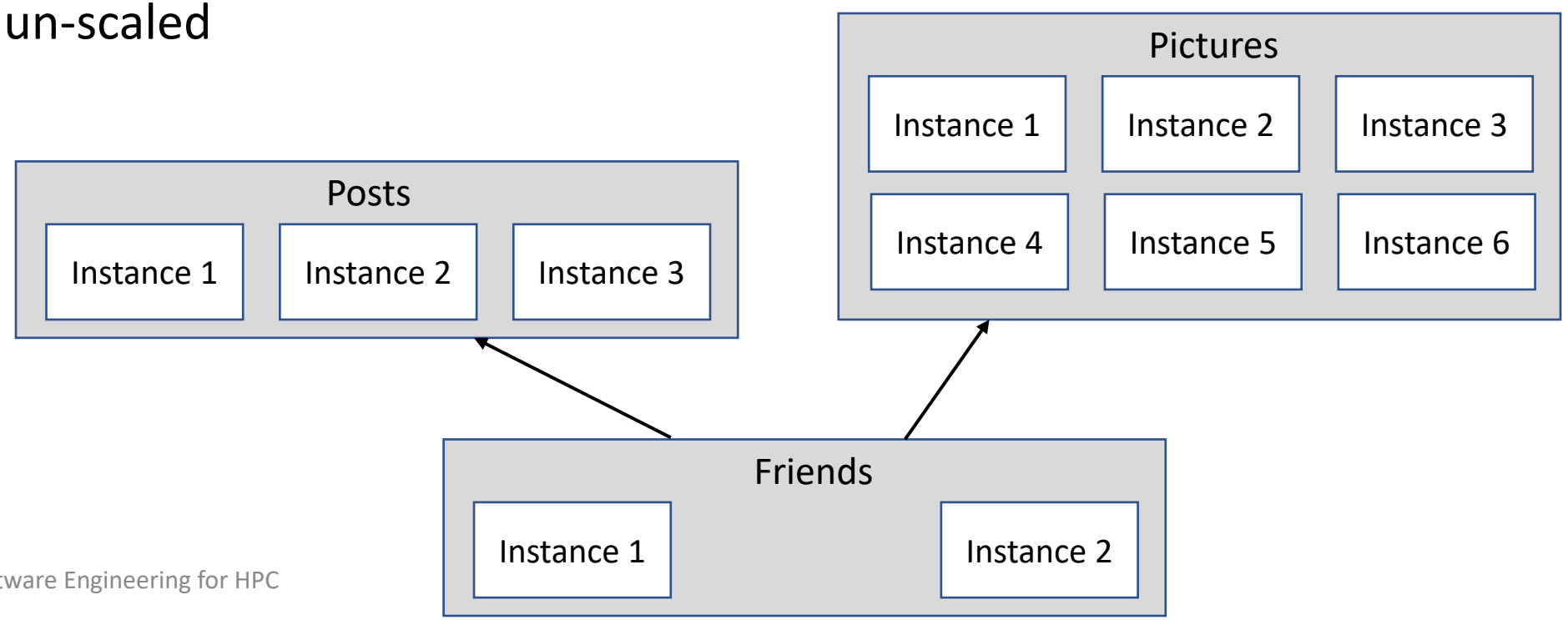
Microservices some key benefits: technology heterogeneity

- Each service uses its own technology stack
  - Technology stack can be selected to fit the task best
    - E.g., Data analysis vs Video streaming
  - Teams can experiment with new technologies within a single microservice
    - E.g., we can deploy the two versions and do A/B testing
  - No unnecessary dependencies or libraries for each service

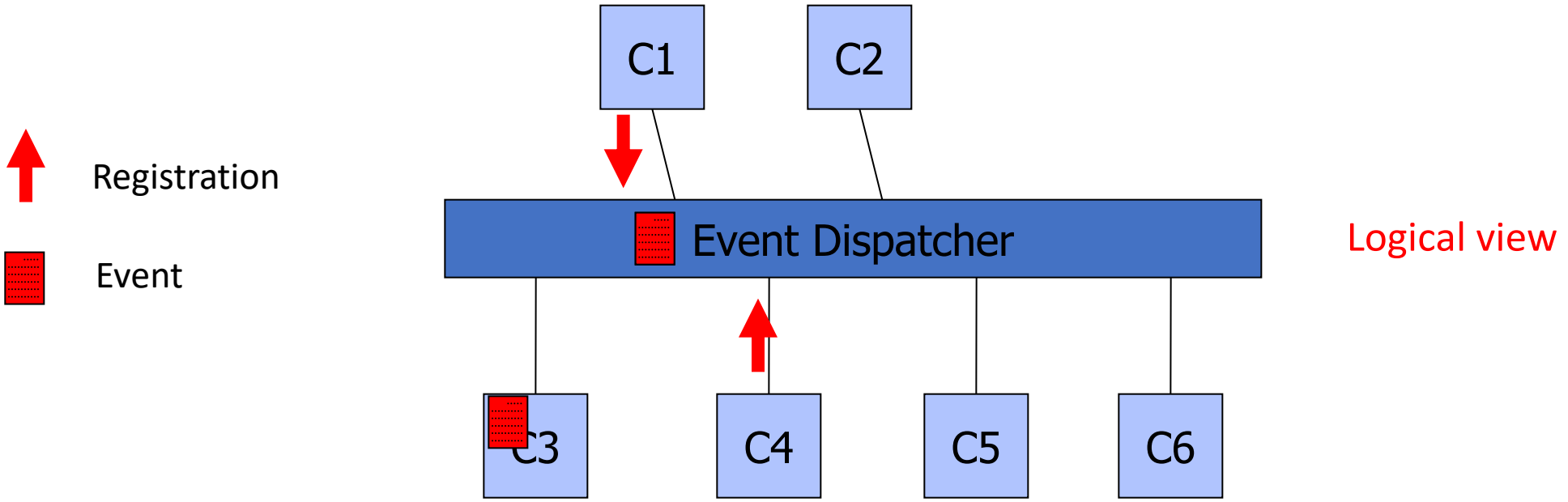


Microservices some key benefits: scaling

- Each microservice can be scaled **independently**
- Identified bottlenecks can be addressed directly
  - Parts of the system that do not represent bottlenecks can remain simple and un-scaled



Event-driven Architecture



Logical view

- Components can **register** to / **send events**
- Events are **sent to all registered components**
- **No explicit naming** of target component

Event-driven paradigm

POLITECNICO MILANO 1863

- Often called publish-subscribe
  - **Publish** = event generation
  - **Subscribe** = declaration of interest
- Different kinds of event languages possible

Event-driven systems Pros & Cons

POLITECNICO MILANO 1863

+ **Very common in modern development practices**
  + E.g., continuous integration / deployment (such as GitHub Actions)
+ **Easy addition/deletion of components**
  + Publishers/subscribers are decoupled
  + The event dispatcher handles this dynamic set
– **Potential scalability problems**
  – The event dispatcher may become a bottleneck (under high workload)
– **Ordering of events**
  – Not guaranteed, not straightforward

Other characteristics

POLITECNICO MILANO 1863

- Messages/events are asynchronous (send and forget)
- Computation is reactive (driven by receipt of message)
- Destination of messages determined by receiver, not sender (location/identity abstraction)
- Loose coupling (senders/receivers added without reconfiguration)
- Flexible communication means (one-to-many, many-to-one, many-to-many)

Examples of relevant technologies

POLITECNICO MILANO 1863

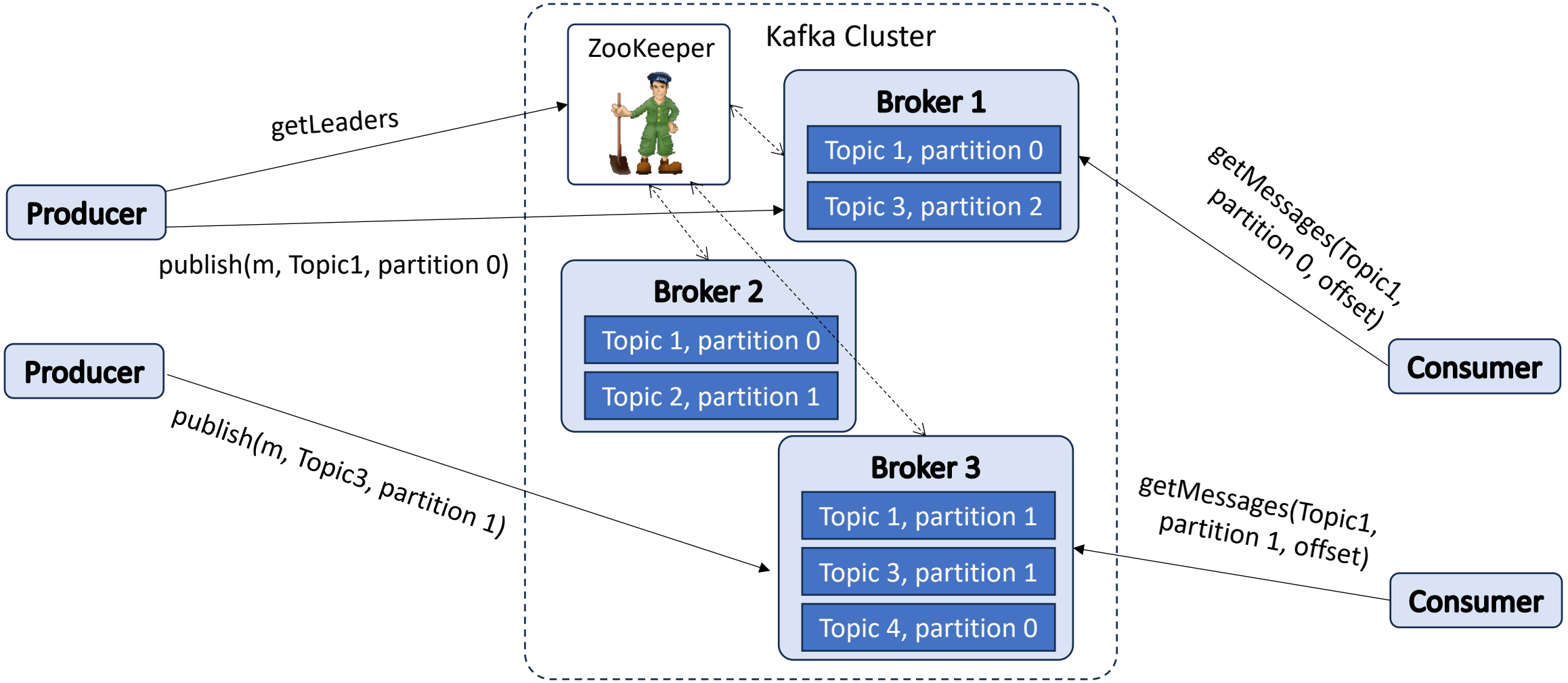
- Apache Kafka
  - https://kafka.apache.org/
- RabbitMQ
  - https://www.rabbitmq.com/

Apache Kafka

POLITECNICO MILANO 1863

- Kafka is a framework for the event-driven paradigm:
  - Includes primitives to create **event producers** and **consumers** and a runtime infrastructure to handle **event transfer** from producers to consumers
  - **Stores events** durably and reliably
  - Allows consumers to **process events** as they occur or retrospectively
- These services are offered in a distributed, highly scalable, elastic, fault-tolerant, and secure manner

Kafka architecture



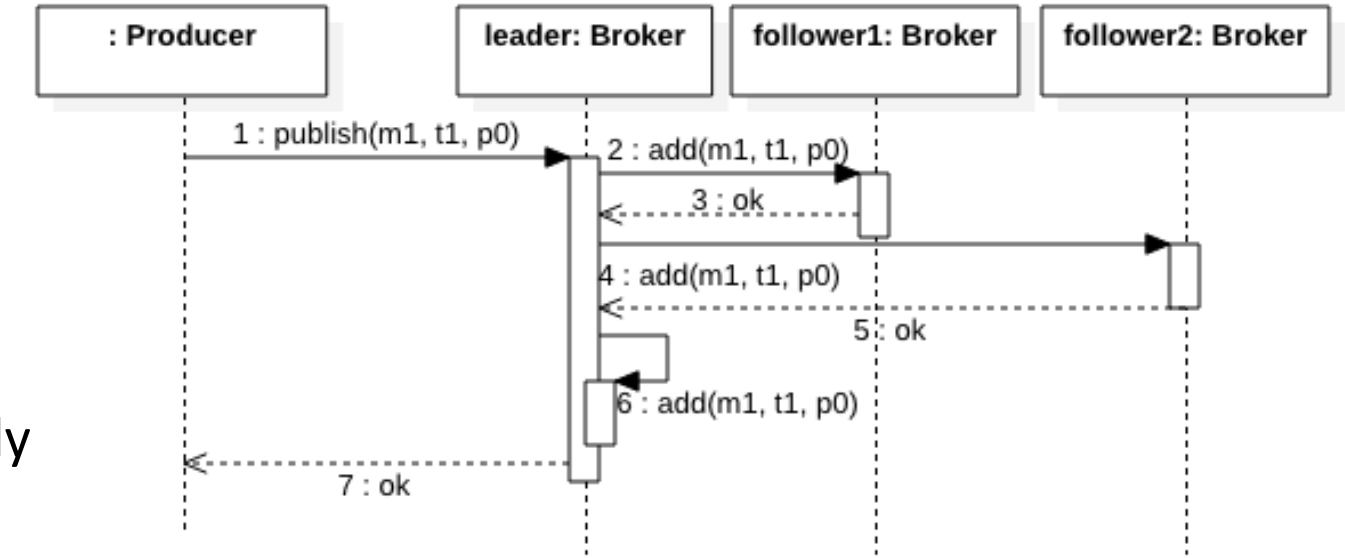
Kafka main characteristics

POLITECNICO MILANO 1863

- Each broker handles a set of topics and topic partitions, parts including sets of messages on the topic
- Partitions are independent from each other and can be replicated on multiple brokers for fault tolerance
- There is one leading broker per partition. The other brokers containing the same partition are followers
- Producers know the available leading brokers and send messages to them
- Messages in the same topic are organized in batches at the producers' side and then sent to the broker when the batch size overcomes a certain threshold
- Consumers adopt a pull approach. They receive in a single batch all messages belonging to a certain partition starting from a specified offset
- Messages remain available at the brokers' side for a specified period and can be read multiple times in this period
- The leader keeps track of the in-synch followers
- ZooKeeper is used to oversee the correct operation of the cluster. All brokers send heartbeat to ZooKeeper. ZooKeeper will replace a failing broker by electing a new leader for all the partitions the failing broker was leading. It may also start/restart brokers

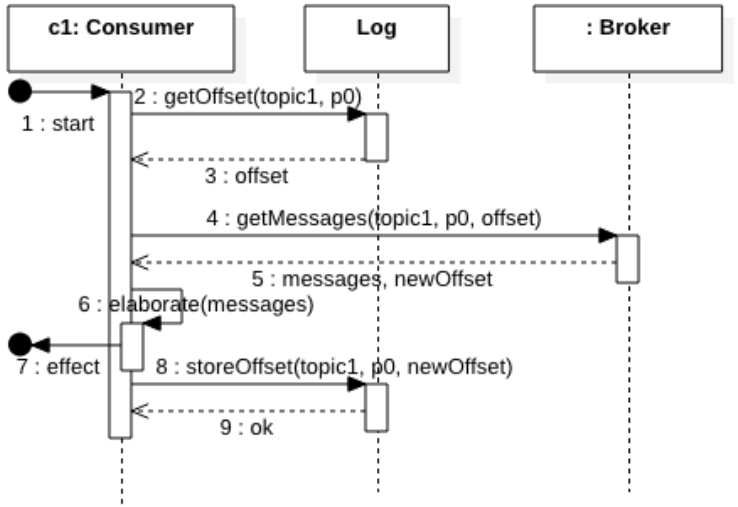
Message delivery semantics — Producer

- Brokers commit messages by storing them in the corresponding partition
- Leader adds the message to followers (replicas) if available
- **Issue**: in case of failure, the producer may not get the response (msg #7)
  - The producer has to resend the message
  - Kafka brokers can identify and eliminate duplicates
- Synchronization with replicas can be transactional
  - Exactly-once semantics is possible but long waiting time
  - At-least-once can be chosen by excluding duplicates' management
  - At-most-once can be chosen by publishing messages asynchronously



Message delivery semantics — Consumer

- Each **consumer** can rely on a **persistent log** to keep track of the **offset** so that it is not lost in case of failure
- If the consumer fails after having elaborated messages and before storing the new offset in the log, the same messages will be retrieved again
  - → at-least-once semantics
- The delivery semantics can be changed if the new offset is stored before the elaboration
  - → at-most-once semantics because, if failing after storing the offset, the effect of the received messages does not materialize
- Transactional management of log allows for exactly-once semantics



Kafka architectural tactics

POLITECNICO MILANO 1863

- Scalability
  - **Multiple partitions and multiple brokers** → possibility to distribute producers/consumers on different partitions handled by different brokers
  - **Scale of operation**: Kafka supports the creation of clusters of brokers
    - Each clusters includes up to 1K brokers able to handle trillions of messages per day
- Fault tolerance
  - Partitions are **stored persistently**
  - **Replication** significantly reduces the chances of losing data
  - Cluster management takes care of restarting brokers and setting leaders as needed

POLITECNICO
MILANO 1863

Architectural styles for data-intensive applications

Batch vs stream processing

POLITECNICO MILANO 1863

- Batch [webster dictionary]
  - A quantity of things/persons considered as a group
  - An amount of something that is made at one time
- Stream [webster dictionary]
  - A continuous flow of people or things
  - Any flow of liquid or gas
- Batch processing
  - Works on batches of data that are retrieved from some data store
  - Generates another batch of data
  - The batch processor terminates when the batch has been processed
- Stream processing
  - Works on continuous flowing data focusing on each single data item
  - Outputs a new stream of data
  - The stream processor is still alive after the processing of a data item

Batch vs streaming processing

Batch

- Has access to all data
- Might compute something big and complex
- Is generally more concerned with throughput than latency of individual components of the computation
- Has latency measured in minutes or more

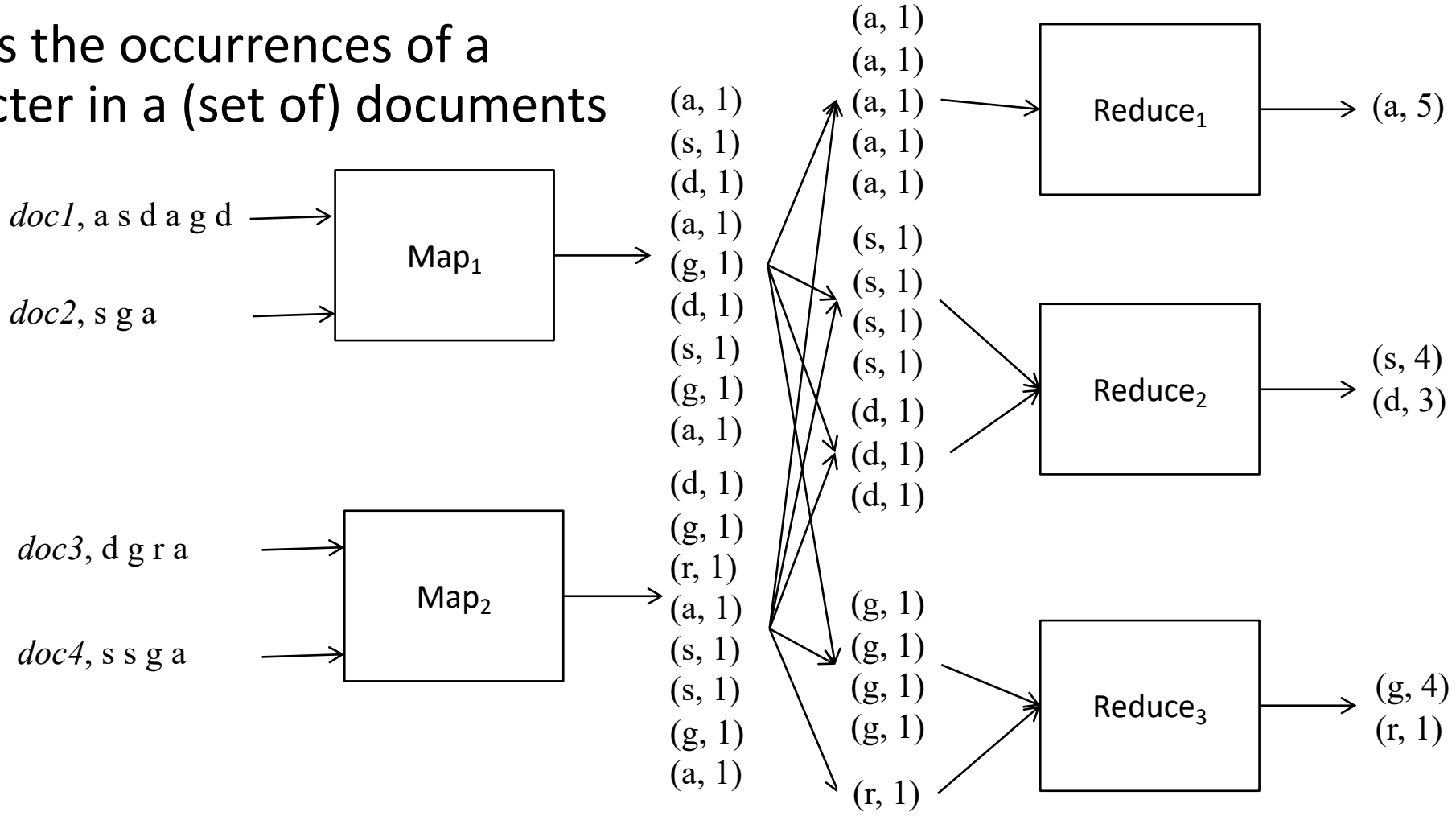
Stream

- Computes a function of one data element, or a smallish window of recent data
- Computes something relatively simple
- Needs to complete each computation in near-real-time – probably seconds at most
- Computations are generally independent
- Asynchronous – source of data doesn't interact with the stream processing directly, like by waiting for an answer

http://softwareengineeringdaily.com/2015/08/03/batch-vs-streaming-the-differences/

An example of batch approach - MapReduce

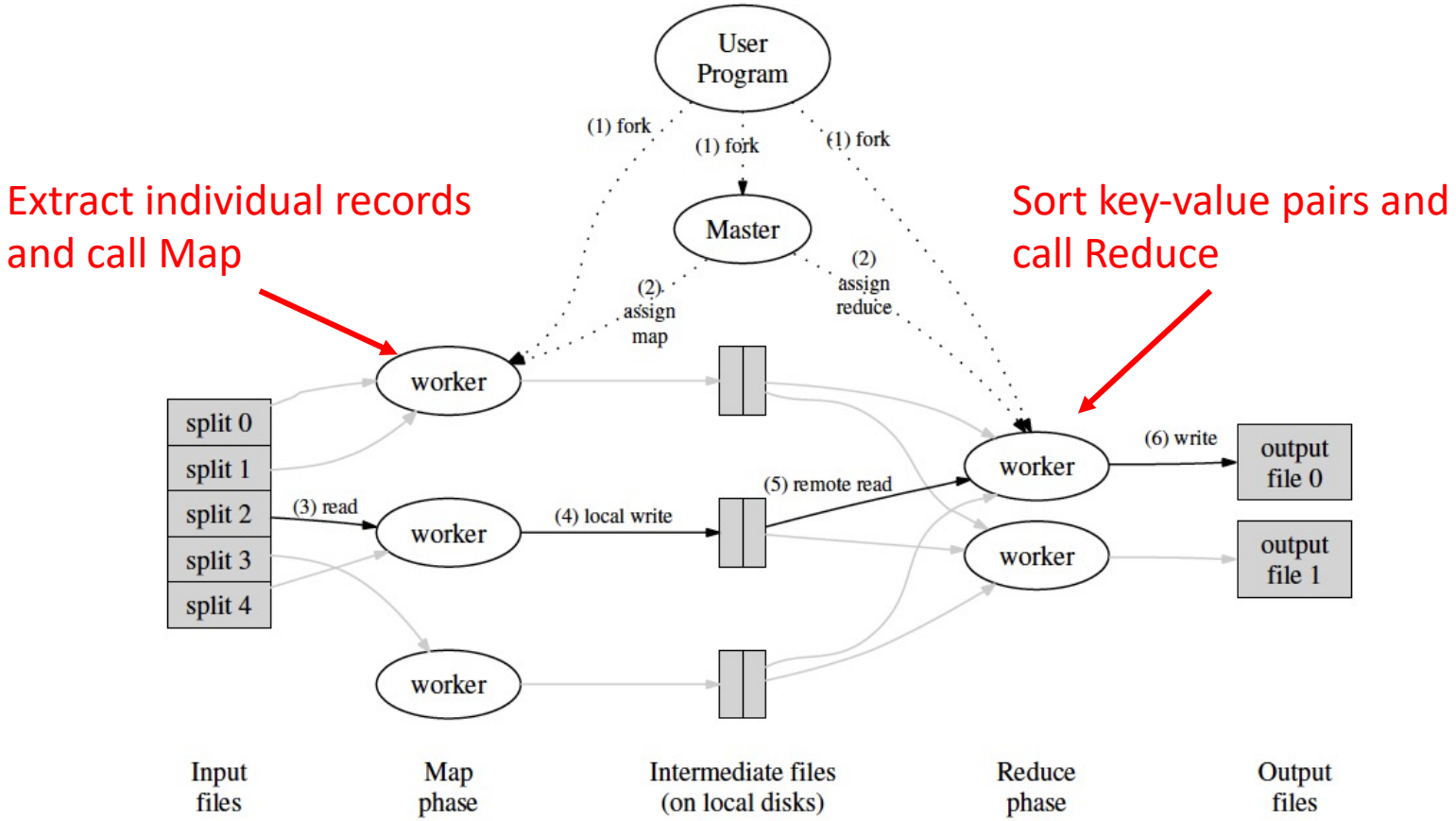
- Counts the occurrences of a character in a (set of) documents



MapReduce workflow

1. Read a set of input files and break it into records
2. Call the map function. It extracts a key and a value from each record (the assigned value is application-dependent)
3. Sort all the key-value pairs by key
4. Call the reduce function. It iterates over the ordered sets of key-value pairs and combines the values (the combination logic is application-dependent)

MapReduce architecture



Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters. In the proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2004

Why MapReduce?

POLITECNICO MILANO 1863

- Works well on commodity hardware
- From the OSDI paper the setting is as follows:
  - Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.
  - Commodity networking hardware is used, typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.
  - A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.
  - Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system (HDFS) is used. It uses replication to provide availability and reliability on top of unreliable hardware.

MapReduce disadvantages

POLITECNICO MILANO 1863

- Implementing a complex processing job is not simple
  - High level programming models have been built on top of it (Hive, Pig, …)
- Reducers have to wait until the preceding Mappers have concluded their job
- Materialization of intermediate states can be overkilling
- Sometimes it is not necessary to sort the results of mappers
- -> New batch computation approaches supported by frameworks as Spark, Tez, Flink, …
  - They offer a less rigid programming model

Stream processing and unbounded data

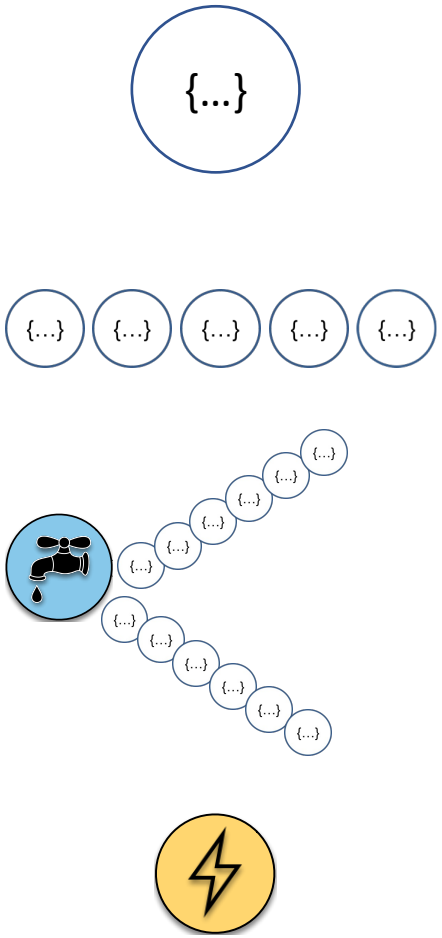
POLITECNICO MILANO 1863

- Ever growing infinite dataset received from some source
  - Mouse clicks
  - Temperature sensors
  - Fixed cameras on the street
  - …
- Often we interested in processing while they arrive and within a specific time-window

An example of streaming approach – Storm

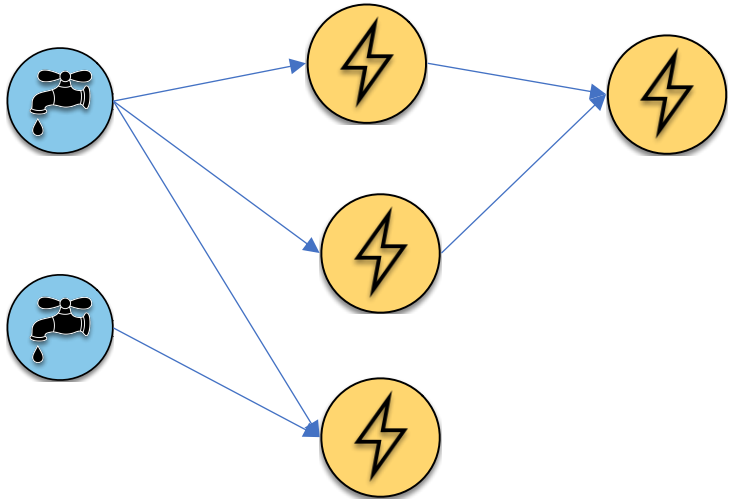
- Support stream processing
- More than 1 million messages per second per node
- Can scale up to thousands of nodes per cluster
- Expects and manages failures (fully fault tolerant)
- Provides guaranteed message delivery with exactly once semantics (reliable)

Storm conceptual model



- Tuple
  - Core unit of data (message)
  - Set of immutable key-value pairs
- Stream
  - Unbounded sequence of tuples
- Spout
  - Source of streams
  - Emits tuple
- Bolt
  - Receives tuples
  - Does computation, reads from and writes to a store
  - Optionally, emits other tuples

Topologies



- Topology = DAG of spouts and bolts
- Each spout/bolt executed as an individual task
- Run in parallel on multiple machines

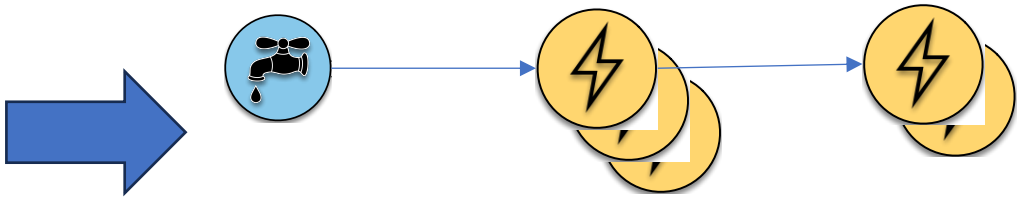
Example

```
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("words", new TestWordSpout(), 10);

builder.setBolt("exclaim1", new ExclamationBolt(), 3)
        .shuffleGrouping("words");

builder.setBolt("exclaim2", new ExclamationBolt(), 2)
        .shuffleGrouping("exclaim1");
```



https://storm.apache.org/releases/2.6.1/Tutorial.html

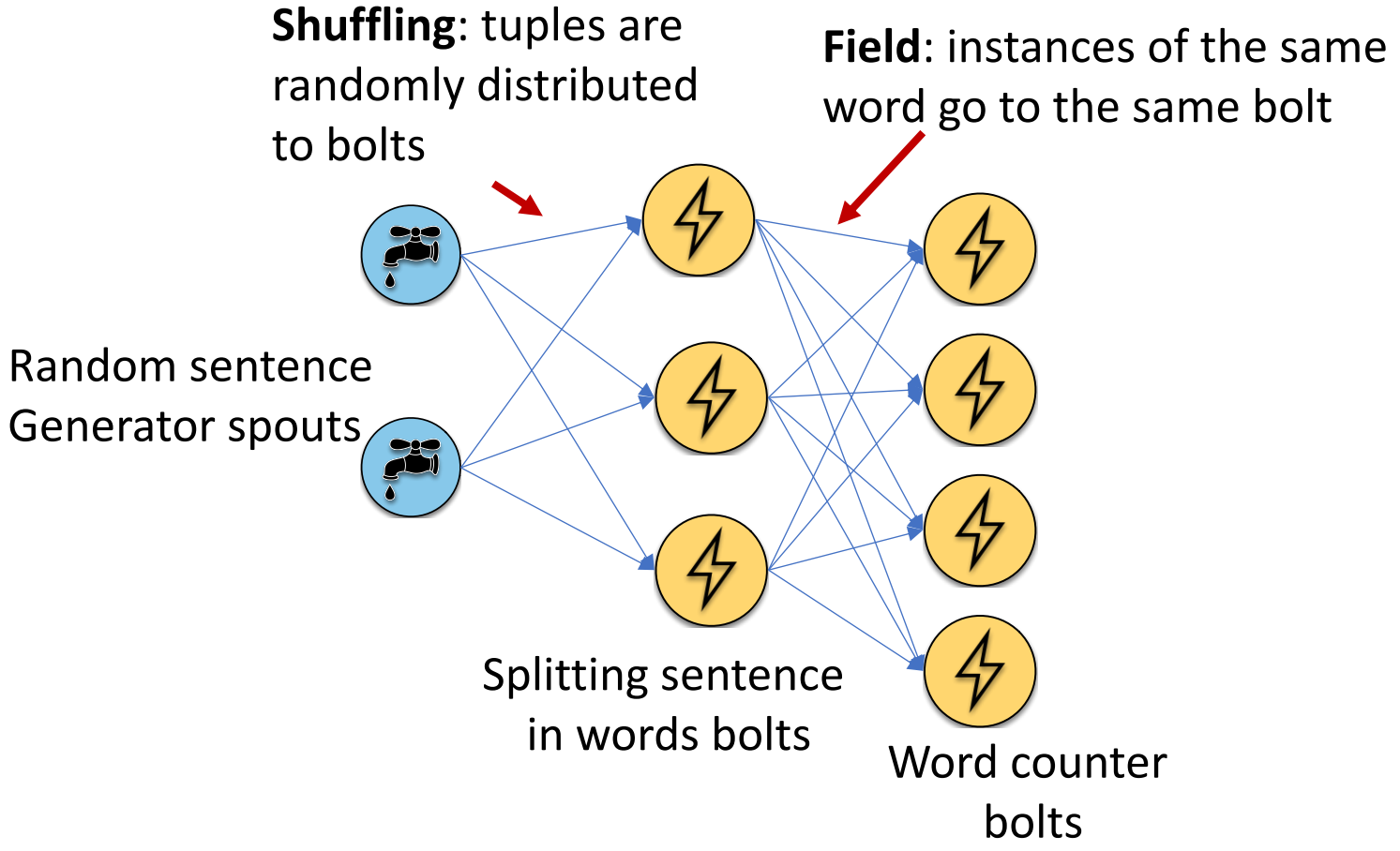
Stream grouping

POLITECNICO MILANO 1863

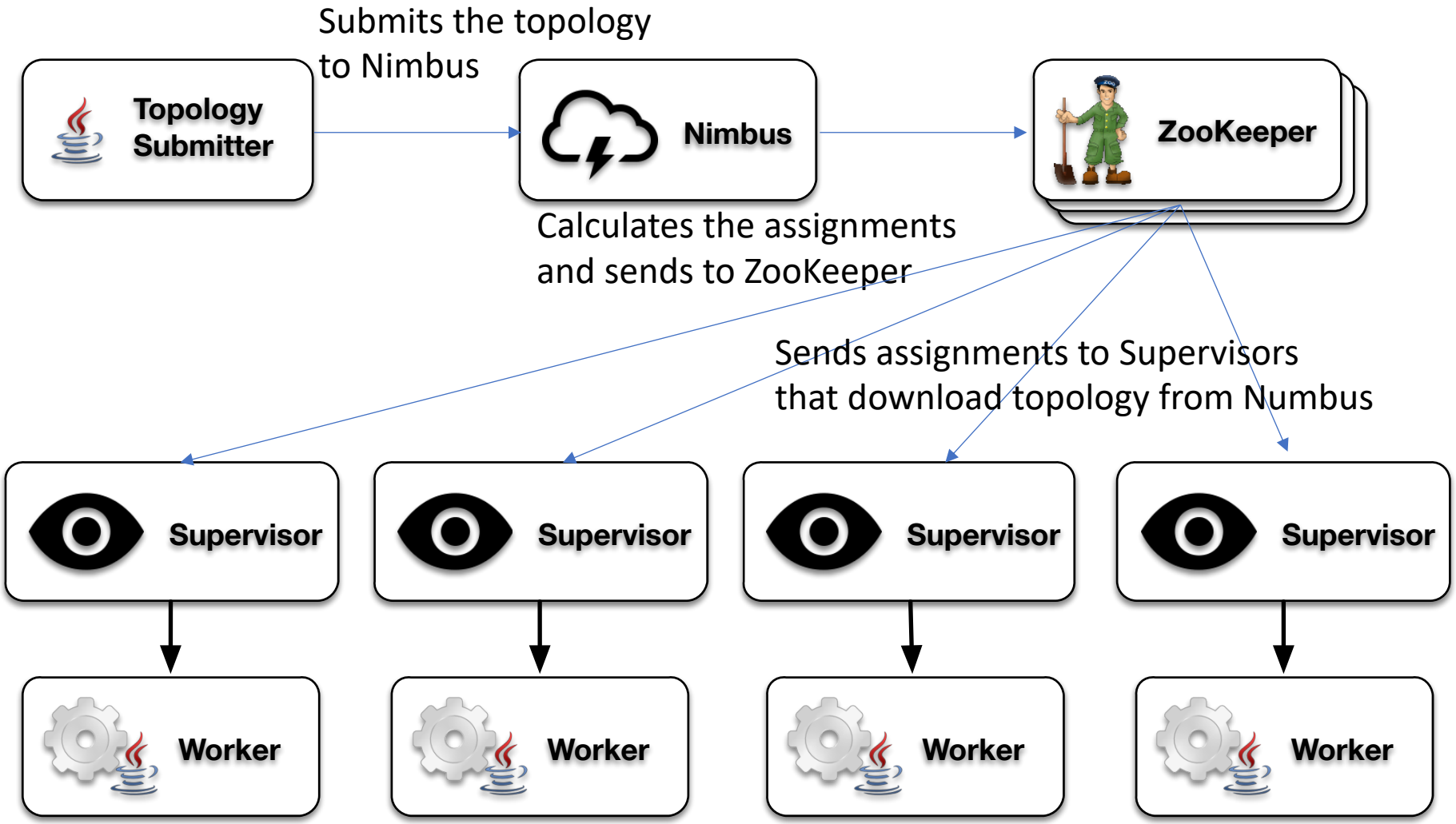
- Goal To control how tuples are routed to bolts in a topology

Stream Grouping	Description
Shuffle	Sends tuples to bolts in random, round robin sequence. Use for atomic operations, such as math.
Fields	Sends tuples to a bolt based on one or more fields in the tuple. Used to segment an incoming stream and to count tuples of a specified type with a specified value.
All	Sends a single copy of each tuple to all instances of a receiving bolt. Use to send a signal, such as clear cache or refresh state, to all bolts.
Custom	Customized processing sequence. Use to get maximum flexibility of topology processing based on factors such as data types, load, and seasonality.
Direct	Source decides which bolt receives a tuple.
Global	Sends tuples generated by all instances of a source to a single target instance. Use for global counting operations.

An example of topology with different groupings



Storm physical view: a cluster

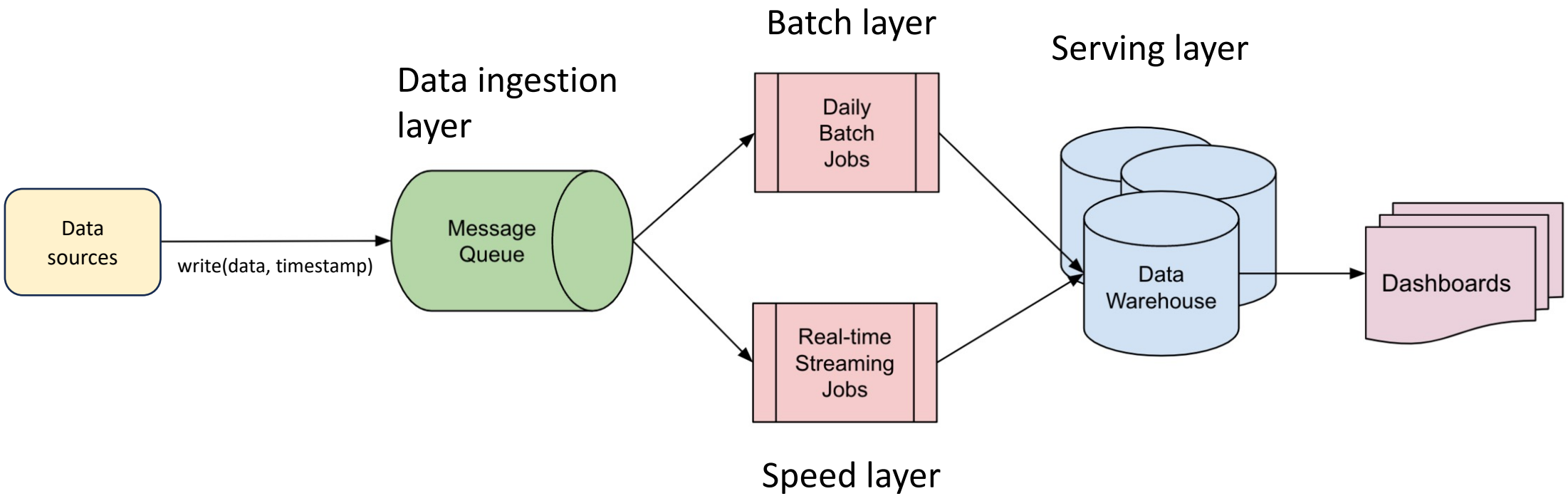


Combining batch and stream processing: λ (Lambda) architecture

POLITECNICO MILANO 1863

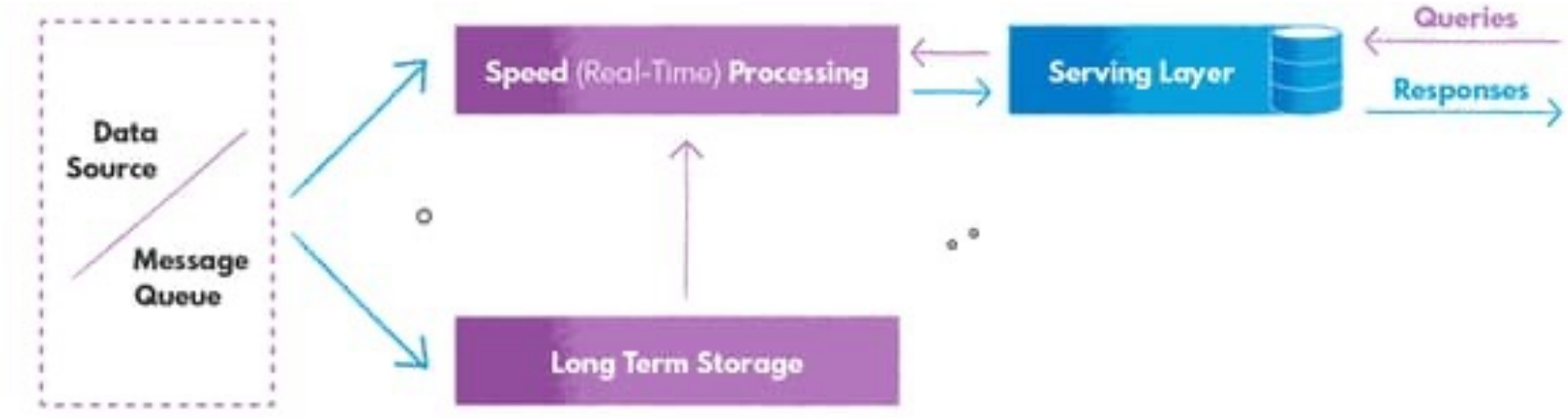
- Problem addressed by the λ architecture
  - To compute certain values we need to analyze a whole dataset -> batch processing
  - BUT, at the same time, we would like to have some hints on intermediate data -> stream processing
- Nice example is available here
  - https://www.linkedin.com/pulse/lambda-architecture-practical-example-richard-sun/

λ architecture



κ architecture

- Eliminates the batch layer
- Computation is always performed in streaming
- The availability of the long term storage allows us to compute precise results



https://nexocode.com/blog/posts/lambda-vs-kappa-architecture/