Apache Kafka

What is event streaming?

- Event streaming:
  - capturing data in real-time from event sources, such as
    - sensors/IoT, mobile devices, social media, cloud services, databases, other software applications
  - storing these event streams durably for later retrieval
  - manipulating, processing, and reacting to event streams
    - in real-time or retrospectively
  - routing the event streams to different destinations
  - ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time
  - «technological foundation for the 'always-on' world»

Uses of event streaming

- Most industries and organizations
  - process payments and financial transactions in real-time
  - track and monitor vehicles and shipments in real-time
  - continuously capture and analyze sensor data from IoT devices and other types of equipment
  - collect and immediately react to customer interactions
  - monitor patients in hospital care
  - predict changes in condition to ensure timely treatment
  - connect, store, and make available data produced by different divisions of a company
  - serve as the foundation for data platforms, event-driven architectures, and microservices

Messaging systems

- A messaging system:
  - manages communication between the components of a distributed system
  - supports the sending and receiving of messages
- Main task:
  - reliably moving messages from sending to receiving machines
- Concerns:
  - performance, synchronisation, queueing
  - reliability (exactly-one, at-least-once, best-effort)
  - scalability, fault-tolerance (distribution, redundancy...)
  - ...and a lot more

Apache Kafka

- Apache Kafka is
  - a scalable, distributed system of *servers* and *clients* that communicate events (data) via a network protocol.
  - an *event streaming platform*
    - in Kafka, «event» means «data that describes an event»
- Combines three key capabilities:
  - publish (write) and subscribe to (read) streams of events
    - including continuous import from/export to other systems
  - store streams of events durably and reliably as wanted
  - process streams of events in real time or retrospectively

Kafka goals

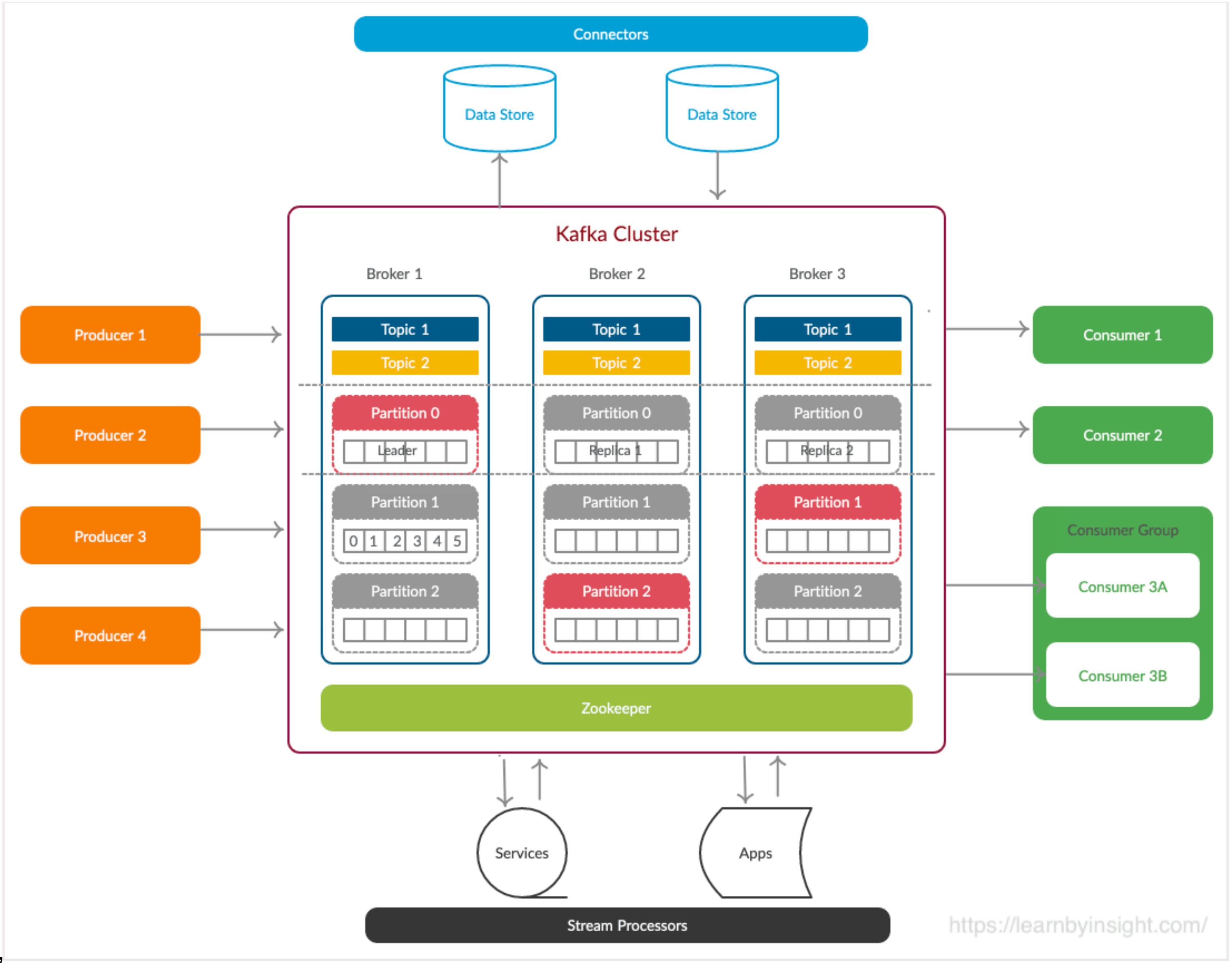
- Design goals:
  - distributed, highly scalable, elastic, and secure
  - fault-tolerant:
    - machine failure or network problems
    - other servers take over for failed/unresponsive servers to ensure continuous operations without data loss
  - deployable on bare-metal hardware, virtual machines, and in containers
  - local, self-managed, or fully-managed cloud services
- Complete and field-tested end-to-end solution

Kafka events

- Event:
  - records the fact that "something happened" in the world or in your business
    - also called record or message
  - reading/writing data from/to Kafka is in the form of events
  - an event has
    - a key ("Alice")
    - a value ("Made a payment of $200 to Bob")
    - a timestamp ("Jun. 25, 2020 at 2:06 p.m.")
    - optional metadata headers

Kafka servers

- Kafka cluster:
  - one or more servers
  - can span multiple datacenters or cloud regions
  - command line tools for management and administration tasks
  - programming APIs: native and third party
- Brokers: storage layer for events (in-memory and on disk)
- Connectors:
  - continuously import and export event streams
  - integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters.
  - Kafka Connect API
- Zookeeper: keeps track of everything


Connectors
Data Store
Data Store
Kafka Cluster
Broker 1
Broker 2
Broker 3
Topic 1
Topic 2
Partition 0
Leader
Replica 1
Replica 2
Partition 1
0 1 2 3 4 5
Partition 2
Zookeeper
Producer 1
Producer 2
Producer 3
Producer 4
Consumer 1
Consumer 2
Consumer Group
Consumer 3A
Consumer 3B
Services
Apps
Stream Processors
https://learnbyinsight.com/

Kafka clients

- Clients:
  - implement distributed applications and microservices
  - read, process, and write streams of events (data) in parallel
  - run on the same or on different machines from the Kafka cluster
  - *producers:* send messages to the Kafka Connectors
  - *consumers:* receive messages from the Kafka Connectors
- Kafka ships with client APIs:
  - REST APIs
  - Java, Scala, the higher-level Kafka Streams library...
- Lots of third-party clients
  - including for Python and Spark

Client decoupling

- Kafka clients (producers and consumers) are:
  - – fully decoupled from each other
  - – agnostic of each other
  - – communication
    - over TCP
    - using Kafka protocols
    - to exchange events (data)
  - – contributes to high scalability
    - e.g., producers never need to wait for consumers

Topics

- Events are organized and durably stored in topics
  - similar to a folder in a filesystem
  - the events (data) are the files in that folder, e.g., "payments", "tweets", "sentiments"...
- Kafka Topics are:
  - multi-producer: a topic can have zero, one, or many producers that write (produce) events to it
  - multi-subscriber: a topic can have zero, one, or many consumers that read (subscribe to) its events

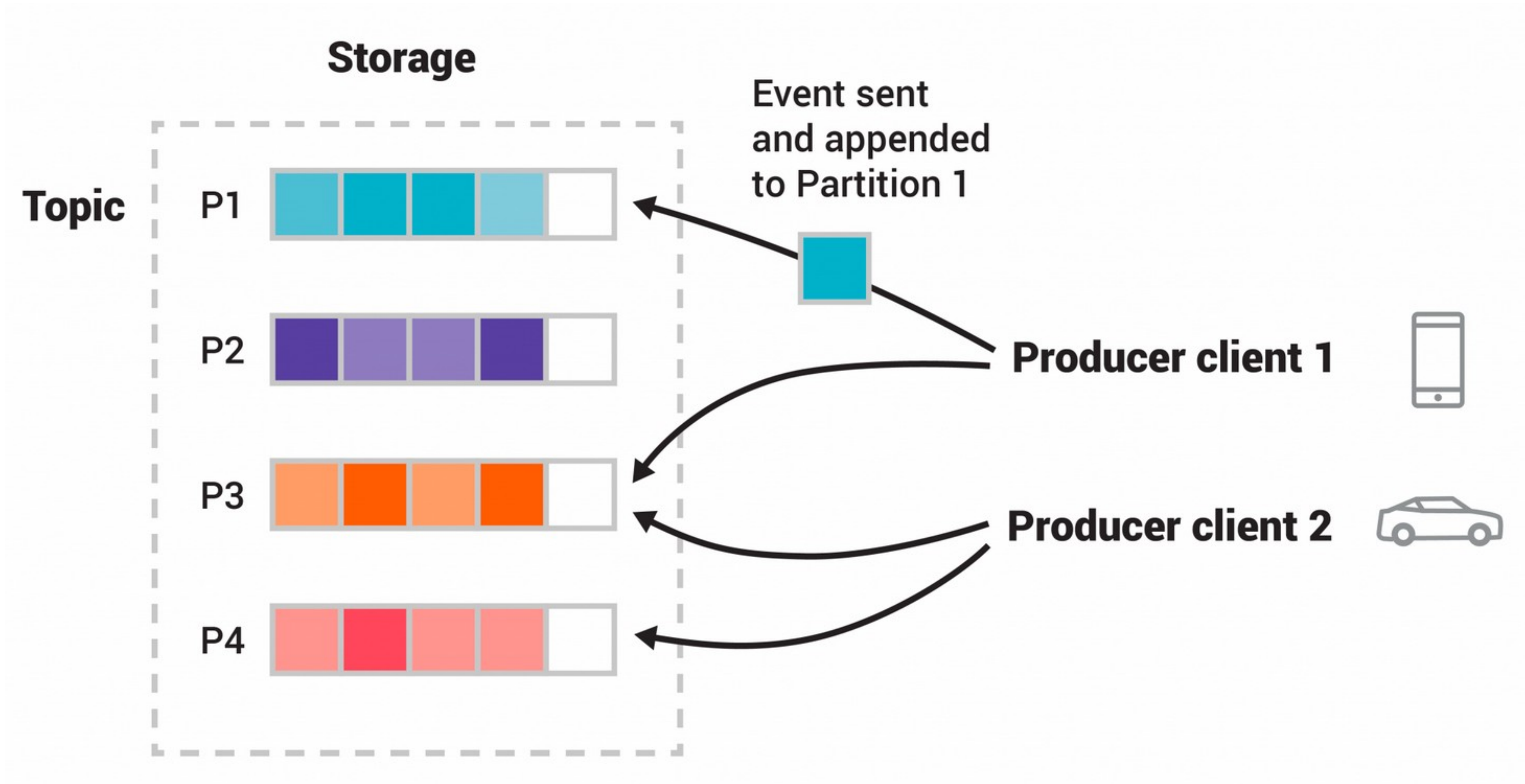
Topic durability

- Events in a topic
  - can be read as often as needed
  - *not* deleted by consumption
  - *the user-defines for how long* Kafka should retain events
  - a *per-topic* configuration setting
  - performance is effectively constant with respect to data size
  - storing data for a long time is perfectly fine.

Partitions

- A topic is spread over a number of «buckets»
- Buckets are located on different Kafka brokers
- When a new event is published to a topic, it is actually appended to one of the topic's partitions
- Events with the same event key (e.g., a customer or tweet ID) are written to the same partition
- Distributed placement of your data is essential for scalability
  - clients can read/write data from/to many brokers in parallel
- Kafka guarantees topic-partition sequence
  - but not overall topic sequence

Partitions


Storage
Event sent and appended to Partition 1
Topic
P1
P2
P3
P4
Producer client 1
Producer client 2

Topic replication

- Replication:
  - across cluster machines, datacenters, or geo-regions
  - every topic can be replicated
  - performed at the level of topic-partitions
  - provides fault-tolerance and high-availability
    - always multiple brokers that have a copy of the data
    - in case of broker failure, maintenance...
  - replication factor of 3 is common

Kafka APIs

- Core Kafka APIs for Java and Scala:
  - Admin API: manage and inspect topics, brokers, other objects
  - Producer API: to publish (write) streams of events topics
  - Consumer API: subscribe to (read) one or more topics and process streams of events
  - Streams API: for streaming applications and microservices
    - some overlap with Spark...
    - transformations, aggregations, joins, windowing, event times
  - Connect API: build and run reusable data importers/exporters
    - consume (read) or produce (write) streams of events from or to external systems and applications
    - Kafka community provides lots of ready-to-use connectors.

Running Kafka

- Standalone: Exercise 3
- Standalone with Streaming Spark: Exercise 3
- On a cluster in the next exercises

Programming with Kafka

- Start Kafka and create topics from command line
- From plain Python:
  - Exercise 3 will use python-kafka:
  - !pip install python-kafka
  - from kafka import KafkaProducer, KafkaConsumer, etc.
- From streaming Spark:
  - Exercise 3 will use spark-sql-kafka
  - imports python-kafka «under the hood»
  - important to get the Scala and Spark versions right