TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

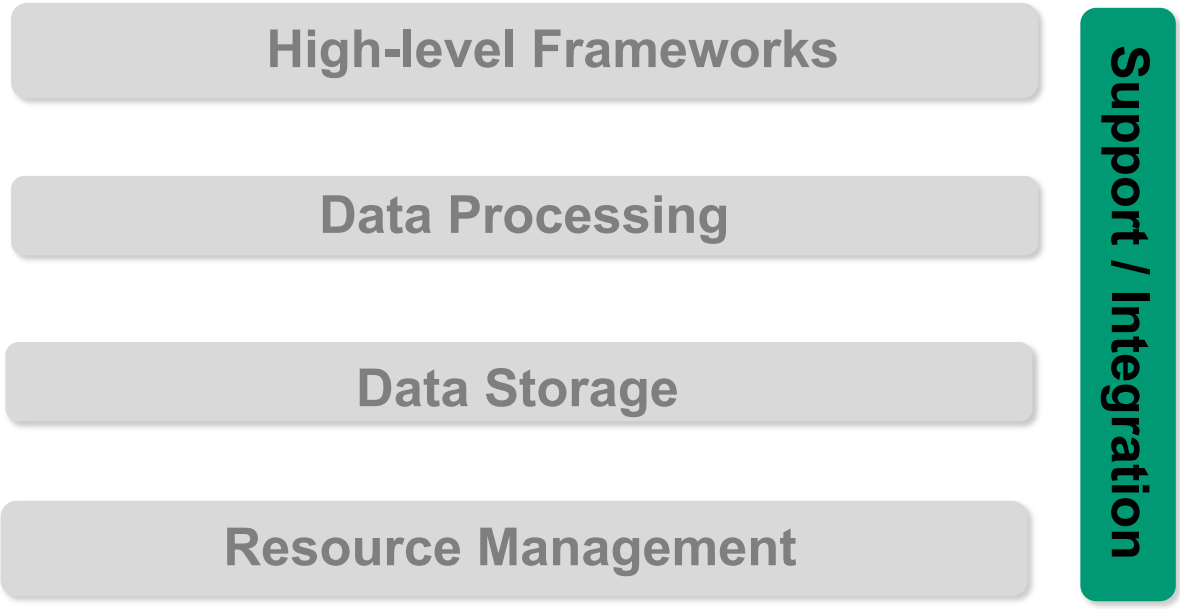
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Data Acquisition and Ingestion

Corso di Sistemi e Architetture per Big Data
A.A. 2021/22
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack



Data acquisition and ingestion

- How to collect data from external (and multiple) data sources and ingest it into a system where it can be stored and later analyzed?
  - Using distributed file systems, NoSQL data stores, batch processing frameworks
- How to connect external data sources to stream or in-memory processing systems for immediate use?
- How to perform some preprocessing (e.g., data transformation or conversion)?

Driving factors

- Source type and location
  - Batch data sources: files, logs, RDBMS, …
  - Real-time data sources: IoT sensors, social media feeds, stock market feeds, …
  - Source location
- Velocity
  - How fast data is generated?
  - How frequently data varies?
  - Real-time or streaming data require low latency and low overhead
- Ingestion mechanism
  - Depends on data consumer
  - Pull vs. push based approach

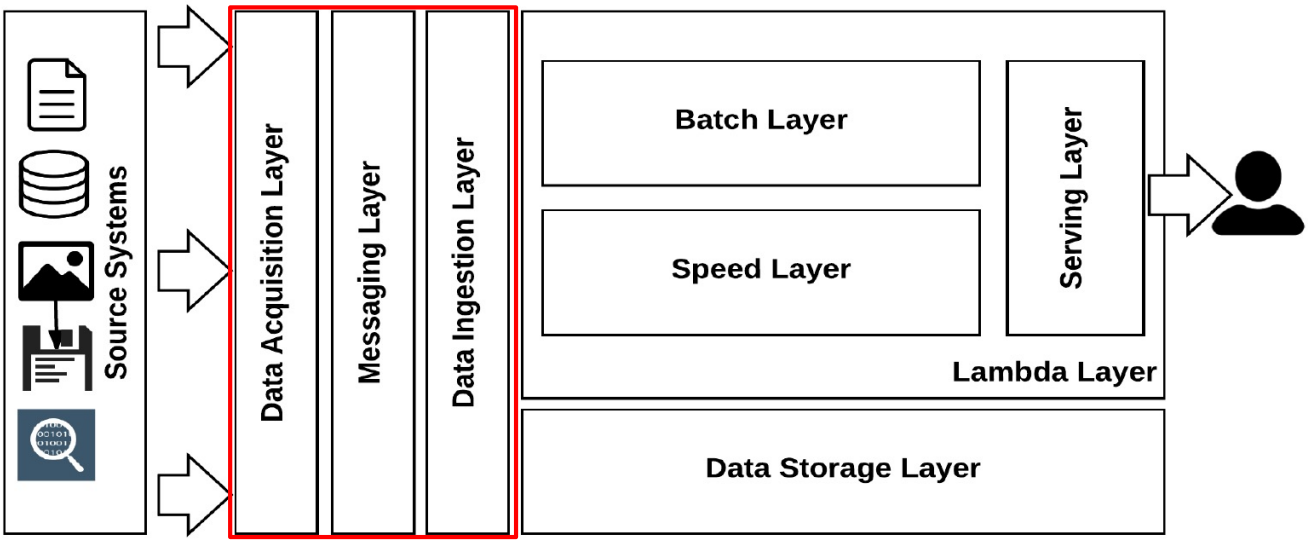
Requirements for data acquisition and ingestion

- Ingestion
  - Batch data, streaming data
  - Easy writing to storage (e.g., HDFS)
- Decoupling
  - Data sources should not directly be coupled to processing framework
- High availability and fault tolerance
  - Data ingestion available 24x7
  - For streaming data: buffering (persistence) in case processing framework is not available
- Scalability and high throughput
  - Number of sources and consumers will increase, amount of data will increase

Requirements for data acquisition and ingestion

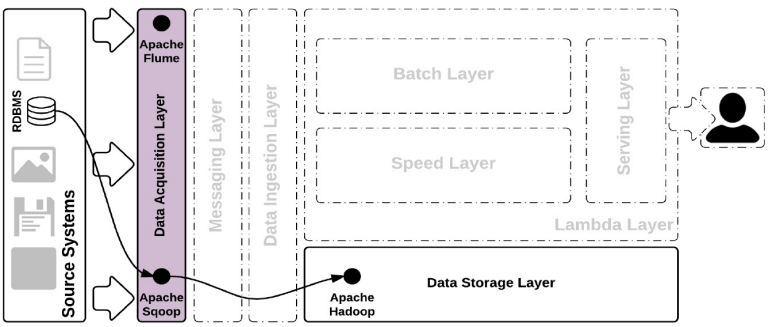
- Data provenance
- Security
  - Data authentication and encryption
- Data conversion
  - From multiple sources: transform data into common format
  - Also to speed up processing
- Data integration
  - From multiple flows to single flow
- Data compression
- Data preprocessing (e.g., filtering)
- Backpressure and routing
  - Buffer data in case of temporary spikes in workload and provide a mechanism to replay it later

A unifying view



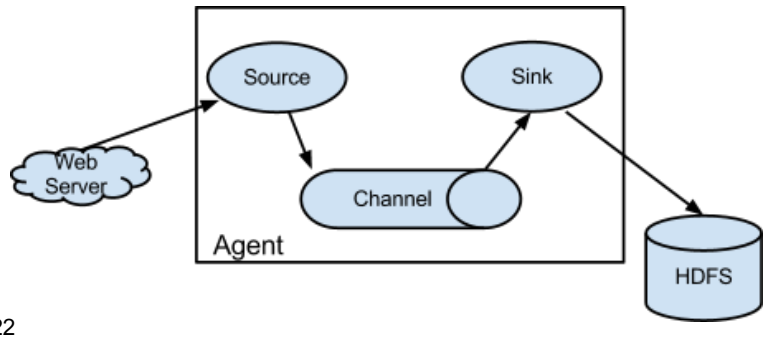
Data acquisition layer

- Allows collecting, aggregating and moving data
- From various sources (server logs, social media, streaming sensor data, …)
- To a data store (distributed file system, NoSQL data store, messaging system)
- We analyze
  - **Apache Flume**
  - **Apache Sqoop**
  - **Apache NiFi**

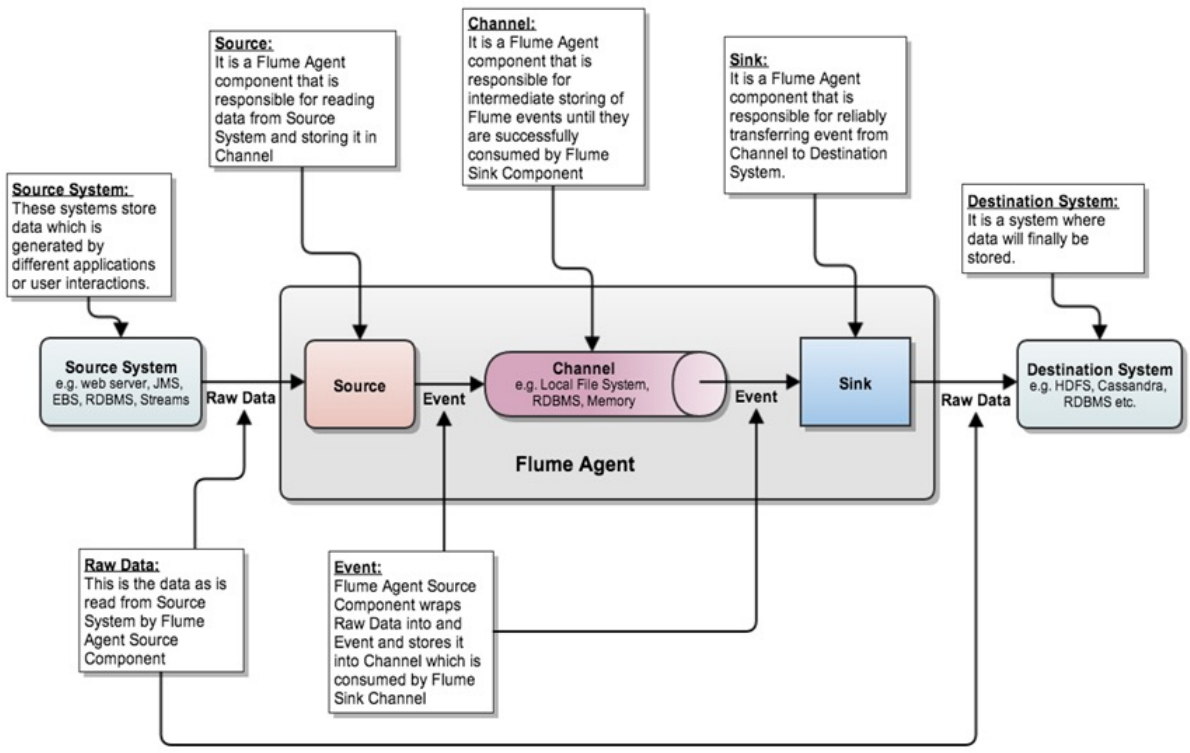


Apache Flume

- Distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of stream data (e.g., log data)
- Robust and fault tolerant with tunable reliability mechanisms and failover and recovery mechanisms
  - Tunable reliability levels
    - Best effort: “Fast and loose”
    - Guaranteed delivery: “Deliver no matter what”
- Suitable for streaming analytics



Flume: architecture

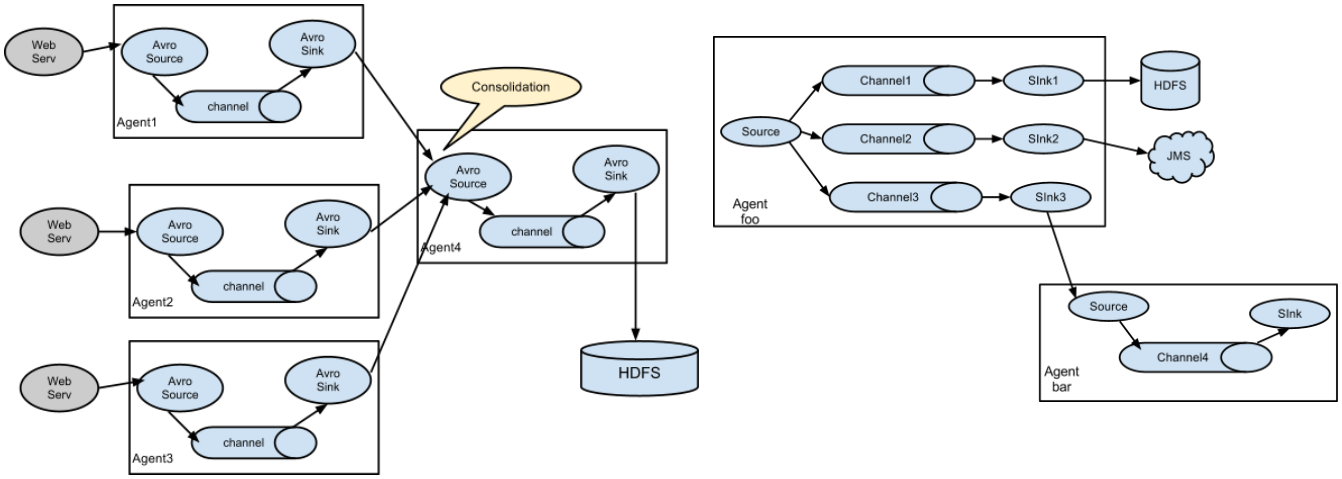


Flume: architecture

- Agent: JVM running Flume
  - One per machine
  - Can run many sources, sinks and channels
- Event
  - Basic unit of data that is moved using Flume (e.g., Avro event)
  - Normally ~4KB
- Source
  - Produces data in the form of events
- Channel
  - Connects source to sink (like a queue)
  - Implements the reliability semantics
- Sink
  - Removes an event from a channel and forwards it to either to a destination (e.g., HDFS) or to another agent

Flume: data flows

- Flume allows a user to build multi-hop flows where events travel through multiple agents before reaching the final destination
- Supports multiplexing the event flow to one or more destinations
- Multiple built-in sources and sinks (e.g., Avro, Kafka)

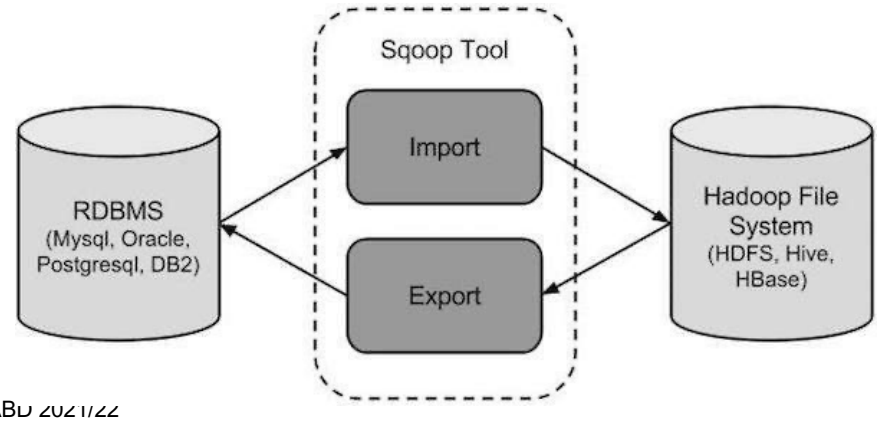


Flume: reliability

- Events are staged in a channel on each agent
  - Channel can be either durable (FILE, will persist data to disk) or non durable (MEMORY, will lose data if machine fails)
- Events are then delivered to next agent or final destination (e.g., HDFS) in the flow
- Events are removed from a channel *only after* they are stored in the channel of next agent or in the final destination
- Transactional approach to guarantee the reliable delivery of events
  - Sources and sinks encapsulate in a transaction the storage/retrieval of events

Apache Sqoop

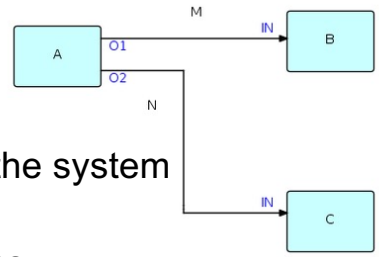
- A commonly used tool for SQL data transfer to Hadoop
  - SQL to Hadoop = SQOOP
- To import bulk data from structured data stores such as RDBMS into HDFS, HBase or Hive
- Also to export data from HDFS to RDBMS
- Supports a variety of file formats (e.g., Avro)



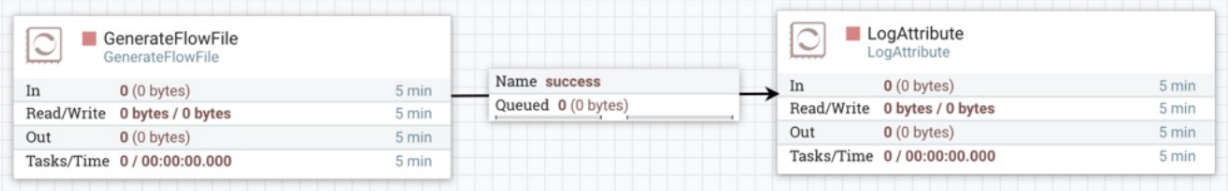
Apache NiFi

- Powerful and reliable system to automate the flow of data between systems, mainly used for data routing and transformation
- Highly configurable
  - Flow specific QoS: loss tolerant vs guaranteed delivery, low latency vs high throughput
  - Dynamic prioritization of queues
  - Flow can be modified at runtime: useful for preprocessing
  - Back pressure
- Data provenance and security (SSL, data encryption, …)
- Ease of use: web-based UI to create and manage the dataflow
  - Allows to define sources from where to collect data, processors for data conversion, destinations to store data
    https://nifi.apache.org/docs/nifi-docs/html/getting-started.html

NiFi: core concepts

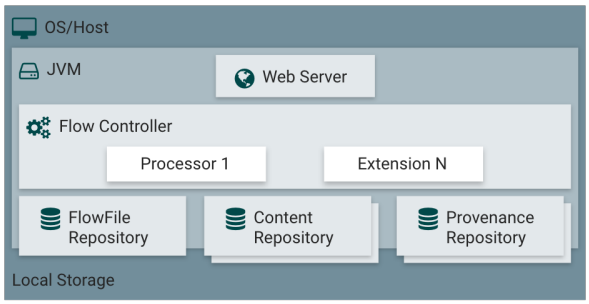


- Based on flow-based programming
- Main NiFi concepts:
  - FlowFile: each piece of user data moving in the system
  - FlowFile Processor: performs data routing, transformation, or mediation between systems
  - Connection: actual linkage between processors; acts as queue
  - Flow Controller: maintains the knowledge of how processes connect and manages threads and allocations
  - Process Group: specific set of processes and their connections

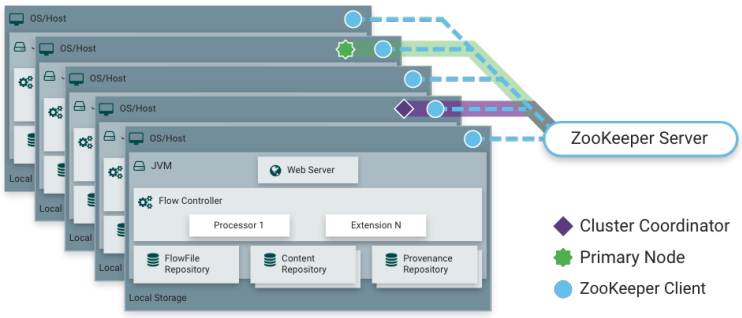


NiFi: architecture

- NiFi executes within a JVM



- Multiple NiFi servers can be clustered for scalability



NiFi: use case

- Use NiFi to fetch tweets by means of NiFi's processor 'GetTwitter'
  - It uses Twitter Streaming API for retrieving tweets
- Move data stream to Apache Kafka using NiFi's processor 'PublishKafka'

Data serialization formats for Big Data

- Serialization: process of converting structured data into a compact (binary) form
- Some data serialization formats you may already know
  - JSON
  - Protocol buffers
- Other serialization formats
  - Apache Avro (row-oriented)
  - Apache Parquet (column-oriented)
  - Apache Thrift https://thrift.apache.org/

Apache Avro

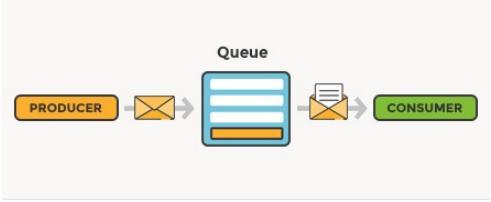
- Key features
  - Compact, fast, binary data format
  - Supports a number of data structures for serialization
  - Neutral to programming language
  - Simple integration with dynamic languages
  - Relies on *schema:* data+schema is fully self-describing
    - JSON-based schema segregated from data
  - Can be used in RPC
  - Both Hadoop and Spark can access Avro as data source
    https://spark.apache.org/docs/latest/sql-data-sources-avro.html
- Comparing performance of serialization formats
  - Avro should not be used from small objects (high serialization and deserialization times)
  - Interesting for large objects

Messaging layer: use cases

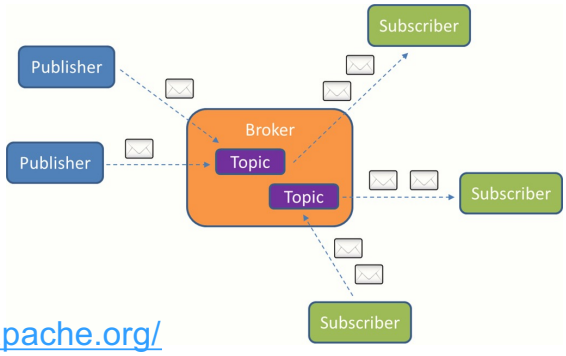
- Mainly used in the data processing pipelines for data ingestion or aggregation
- Envisioned mainly to be used at the beginning or end of a data processing pipeline
- Example
  - Incoming data from various sensors: ingest data into a streaming system for real-time analytics or a distributed file system for batch analytics

Messaging layer: architectural choices

- Message queue
  - ActiveMQ
  - RabbitMQ
  - ZeroMQ
  - Amazon SQS



- Publish/subscribe
  - Kafka
  - NATS http://www.nats.io
  - Apache Pulsar https://pulsar.apache.org/
  - Redis

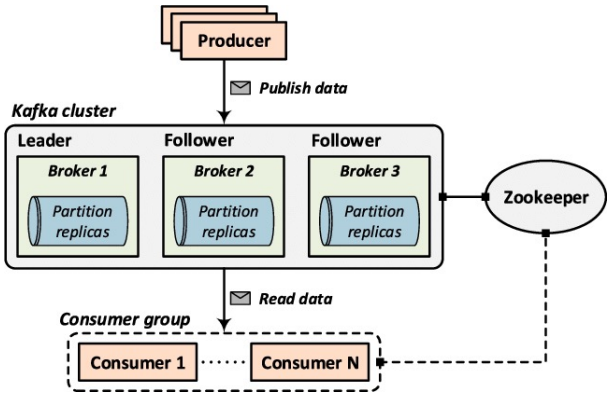


Apache Kafka

- Analyzed in SDCC course
- In a nutshell https://kafka.apache.org/
  - Open-source, distributed pub/sub and event streaming platform
  - Designed as a replicated, distributed, persistent commit log
  - Clients produce or consume events directly to/from a cluster of brokers, which read/write events durably to the underlying local file system and also automatically replicate the events synchronously or asynchronously within the cluster for fault tolerance and high availability
- Let's recall the main points

Kafka: architecture

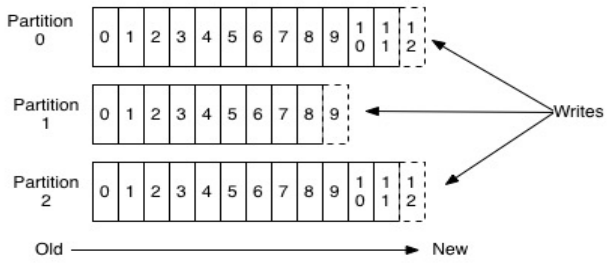
- Kafka maintains feeds of messages in categories called topics
- Producers publish messages to a Kafka topic, while consumers subscribe to topics and process published messages



- Kafka cluster: distributed and replicated commit log of data over servers known as *brokers*
  - Brokers rely on Apache Zookeeper for coordination

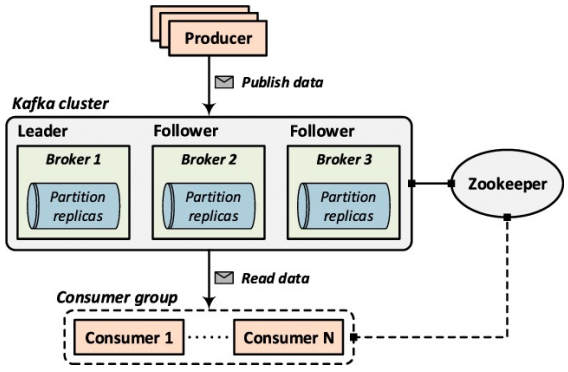
Kafka: topics and partitions

- For each topic, Kafka cluster maintains a partitioned log: topic is split into a fixed number of partitions
- Each partition is an ordered, numbered, immutable sequence of records that is continually appended to
- Each partition is replicated for fault tolerance across a configurable number of brokers
- Partitions are distributed across brokers for scalability



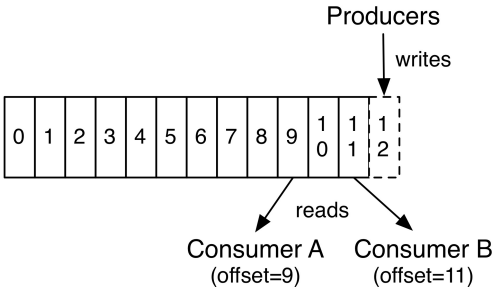
Kafka: partition replication

- Each partition has one leader broker and 0 or more followers
- Leader handles read and write requests
- A follower replicates the leader and acts as a backup
- Each broker is a leader for some of its partitions and a follower for others to distribute load



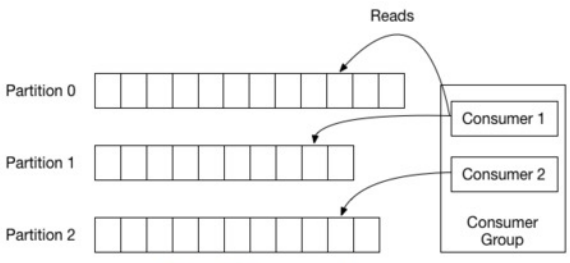
Kafka: partitions

- Producers publish their records to partitions of a topic (round-robin or partitioned by keys), and consumers consume published records of that topic
- Each record is associated with a monotonically increasing sequence number, called offset
  - Kafka provides the topic `__consumer_offsets` for storing the offsets
- Consumers must manage their *offset*



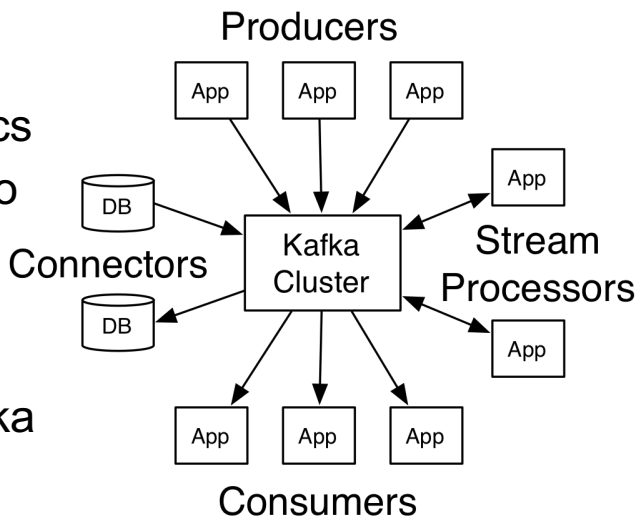
Kafka: consumers

- In Kafka design, pull approach for consumers
  http://kafka.apache.org/documentation.html#design_pull
- Consumers use offset to track which messages have been consumed
  - Replay messages using offset
- Consumers can be grouped into a Consumer Group: set of consumers sharing a common group ID
  - A Consumer Group maps to a logical subscriber
  - Each group consists of multiple consumers for scalability and fault tolerance



Kafka: APIs

- Four core APIs https://kafka.apache.org/documentation/#api
- Producer API: allows apps to publish records (e.g., clickstream, logs, IoT) to topics
- Consumer API: allows apps to read records from topics
- Connect API: reusable connectors (producers or consumers) that connect Kafka topics to existing applications or data systems so to move large collections of data into and out of Kafka
  - Connectors for AWS S3, HDFS, RabbitMQ, MySQL, Postgres, AWS Lambda, MongoDB, Twitter, …



Kafka: APIs

- Streams API: allows transforming streams of data from input topics to output topics
  - Kafka as also a real-time streaming platform
- Hands-on course: you will use Kafka Streams to process data in pipelines consisting of multiple stages

Kafka: limitations

- No complete set of monitoring tools
- No support for wildcard topic selection
- Limited support for geo-replication
  - Single Apache Kafka cluster can run across multiple geo-regions but suffers from latency
  - Kafka's MirrorMaker tool allows to replicate data (topics, consumer groups and their offset) among different clusters located in different geographic locations

Hands-on Kafka

- Preliminary steps:
  - Download and install Kafka http://kafka.apache.org/downloads
    - Configure Kafka properties in `server.properties` (e.g., `listeners` and `advertised.listeners`)
    - As alternative, see Bitnami Docker image for Kafka
  - Start Kafka environment
    - Start ZooKeeper (default port: 2181)

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

    - Start Kafka broker (default port: 9092)

```
$ bin/kafka-server-start.sh config/server.properties
```

- To list existing topics

```
$ kafka-topics --list --zookeeper localhost:2181
```

- To delete a given topic

```
$ kafka-topics --delete --zookeeper localhost:2181 --topic name
```

Hands-on Kafka

- Let's use CLI tools to create a topic, write some events into the topic and read events from the topic
- Create a topic named test with 1 partition and 1 replica

```
$ bin/kafka-topics.sh --create --bootstrap-server
localhost:9092 --replication-factor 1 --partitions 1 --
topic test
```

- Write some events into the topic

```
$ bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic test

> This is the first message

> This is another message
```

- Read the events

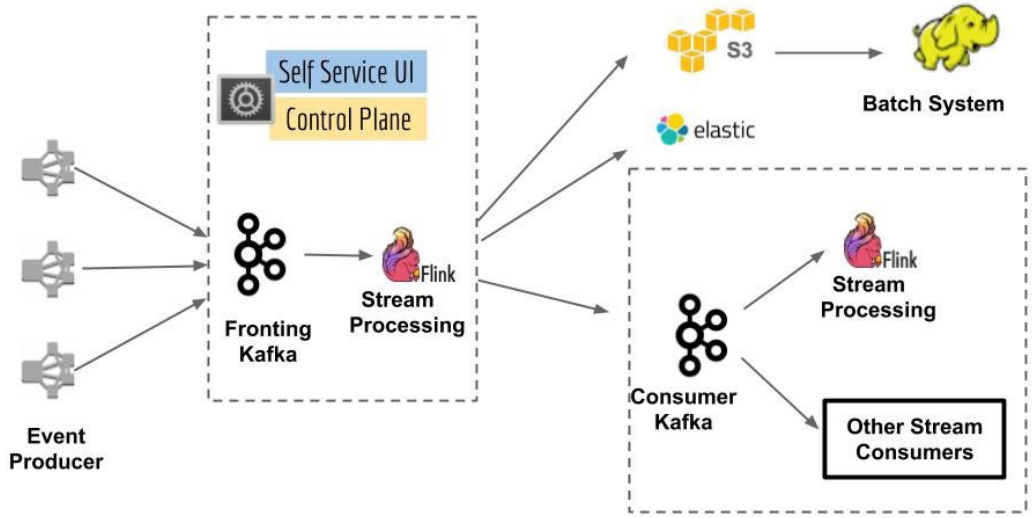
```
$ bin/kafka-console-consumer.sh --bootstrap-server
localhost:9092 --topic test --from-beginning
```


Kafka and Python client library

- Multiple options, let's consider Kafka-Python
  https://pypi.org/project/kafka-python/
- Preliminary steps
  - Install, configure and start Kafka and Zookeeper
  - Install Python client library
- See `kafka-python_example.py` on course site

Kafka @ Netflix

- Netflix uses Kafka for data collection and buffering

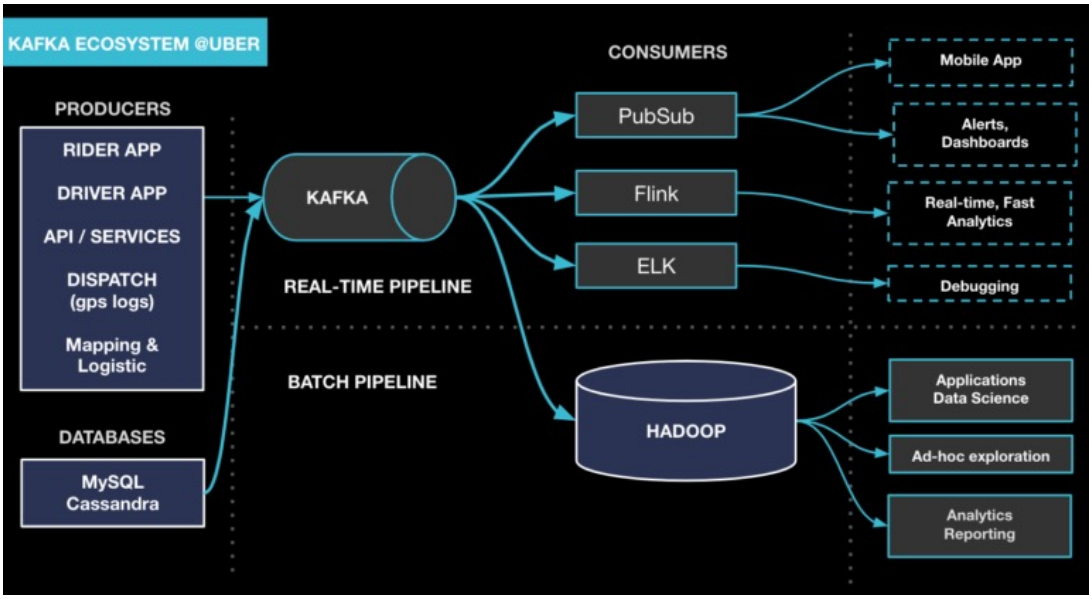


See http://techblog.netflix.com/2016/04/kafka-inside-keystone-pipeline.html

- Another example from Netflix https://www.confluent.io/blog/how-kafka-is-used-by-netflix/

Kafka @ Uber

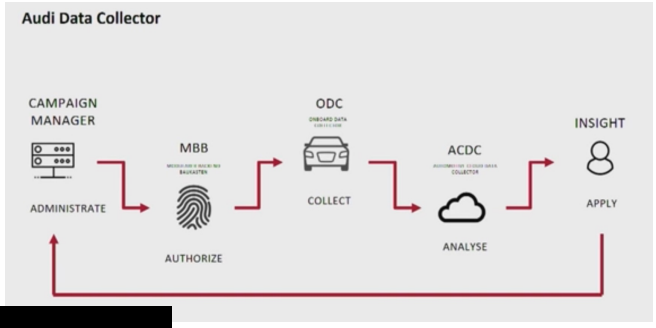
- Uber has one of the largest Kafka deployment in the industry

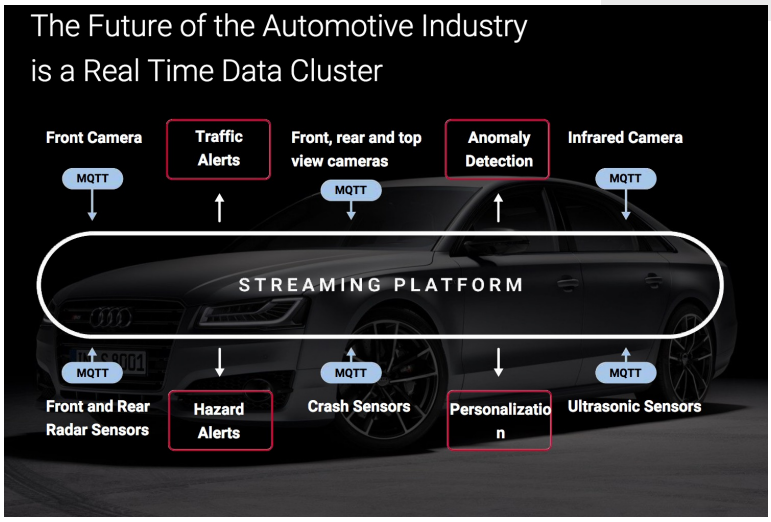


https://eng.uber.com/ureplicator/

Kafka @ Audi

- Audi uses Kafka for real-time data processing
  - 850 sensors in each car

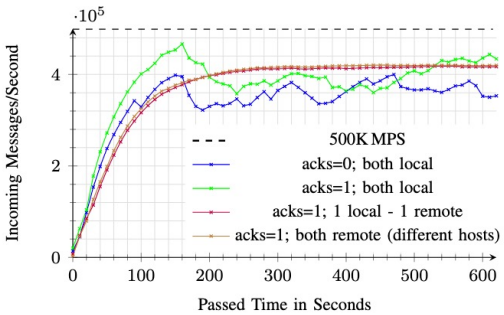




https://www.youtube.com/watch?v=yGLKi3TMJv8

Kafka performance

- A performance evaluation study of Apache Kafka
  How Fast Can We Insert? An Empirical Performance Evaluation of Apache Kafka, ICPADS 2020
  - Kafka can achieve an ingestion rate of about 421K messages/second or 92 MB/s (single topic with 1 partition and replication factor of 1) on commodity hardware and using the developed data sender tool (2 senders)
  - Influence on performance of the chosen ack level: configurations with enabled acks showed better performance



Kafka: cloud services

- Fully-managed services based on Kafka

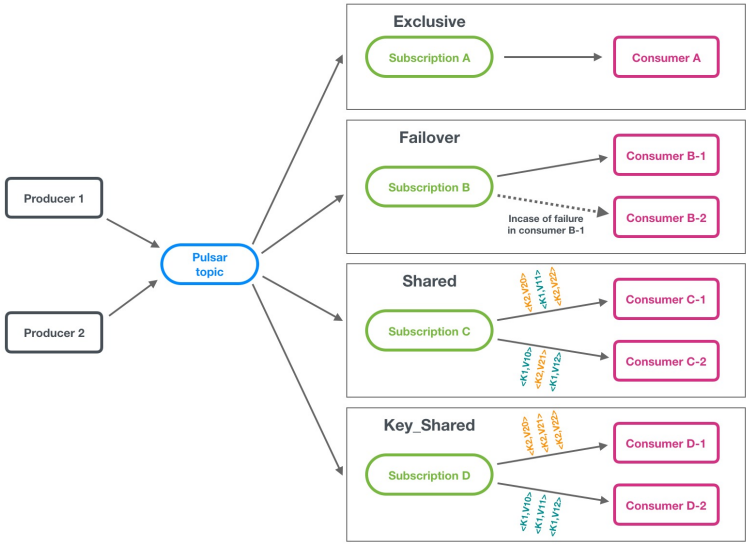
- Amazon MSK (Managed Streaming for Apache Kafka) https://aws.amazon.com/msk
- Confluent Cloud https://www.confluent.io/confluent-cloud
  - Led by the creators of Kafka
- CloudKarafka https://www.cloudkarafka.com

Apache Pulsar

- Cloud-native, distributed messaging and streaming platform, originally developed by Yahoo
- Scalable, low-latency and durable messaging based on pub-sub pattern, with support for multi-tenancy and geographical replication
- Multiple subscription modes for topics
- Guaranteed message delivery with persistent message storage provided by Apache BookKeeper
- Enables also stream-native data processing through a lightweight function-based computing framework, named Pulsar Functions

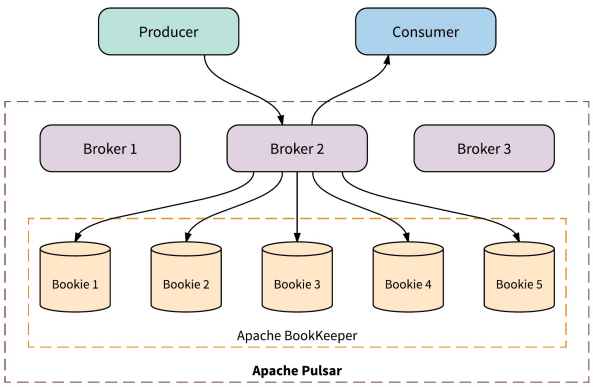
Pulsar: subscription modes

- Multiple subscription modes: exclusive, shared, failover, and key_shared



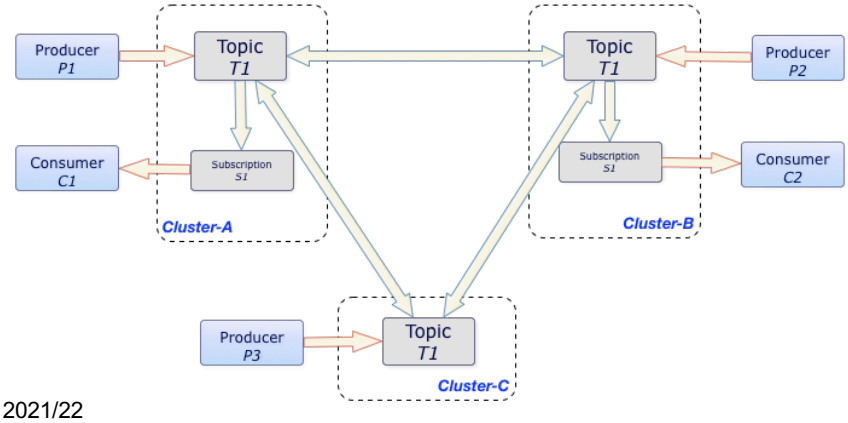
Pulsar: architecture

- Layered architecture designed to provide scalability and flexibility
  - Stateless serving layer and stateful persistence layer
  - Serving layer comprised of **brokers** that receive and deliver messages
  - Persistence layer comprised of Apache BookKeeper storage nodes called **bookies** that durably store messages
    - BookKeeper is a distributed write-ahead log



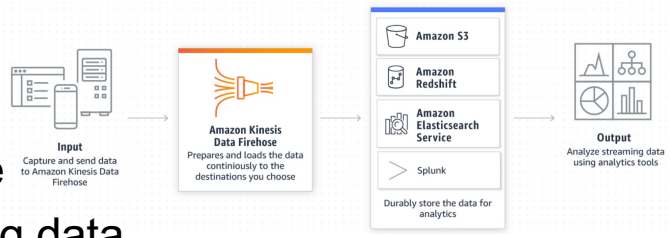
Pulsar: architecture

- Pulsar instance of Pulsar composed of one or more Pulsar clusters
  - Clusters may be geographically distributed and data can be geo-replicated among different clusters
  - Each cluster consiste of one or more brokers, an ensemble of bookies, and a ZooKeeper quorum
  - ZooKeeper is used for cluster-level configuration and coordination

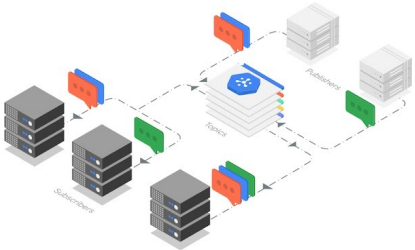


Cloud services for data ingestion

- Amazon Kinesis Data Firehose
  - Fully managed service for delivering streaming data directly to S3, used as data lake
  - Can transform and compress streaming data before storing it
  - Can invoke Lambda functions to transform incoming source data and deliver it to S3



- Google Cloud Pub/Sub https://cloud.google.com/pubsub/
  - Fully-managed real-time pub/sub messaging service



References

- Apache Flume documentation, https://flume.apache.org/FlumeUserGuide.html
- Apache NiFi documentation, https://nifi.apache.org/docs.html
- Apache Kafka documentation, https://kafka.apache.org/documentation/
- Apache Pulsar documentation, https://pulsar.apache.org/docs/en/standalone/
- Kreps et al., Kafka: A distributed messaging system for log processing, *NetDB 2011*.