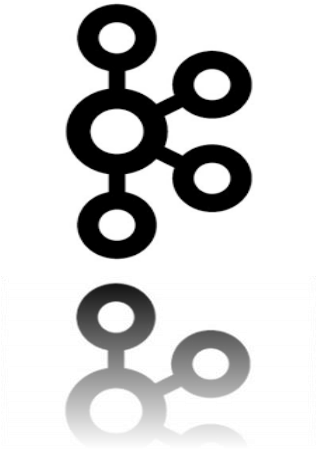
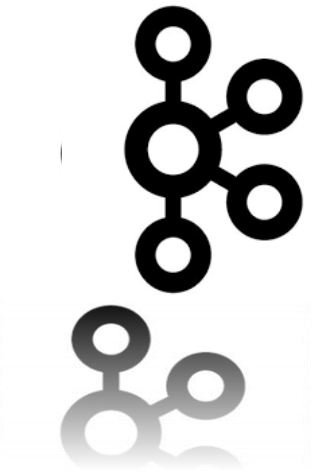
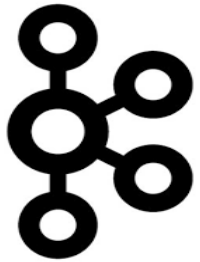


Apache Kafka Series



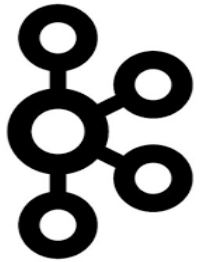
Kafka - Core

Course Objectives



- What is Kafka? Why is used? Problem it solved?
- Learn about Kafka Eco-system (High Level)
- Learn about Apache Kafka Core APIs
  - Topics, Partitions
  - Broker, Replication, Zoo Keeper
  - Producers, Consumer and Consumer Group
- Get started with multi broker setup on your own machine
- Learn about some tools to speed up the development process with Apache Kafka.

About me

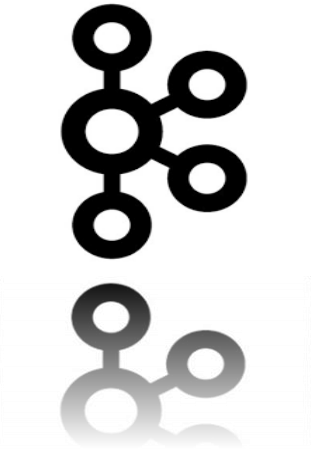


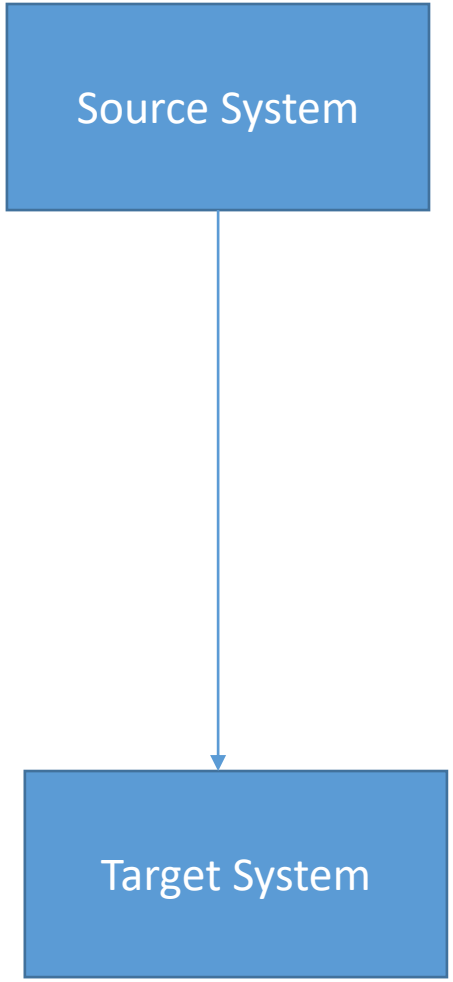
- I am Sanjoy Ganguly
- I worked at LTI (Bangalore) as Solution Architect
- Worked with Apache Kafka a bit
- Did full-deployments in various client

Pre-requisites before we get started

- Ability to use command line
  - We will use it to launch Kafka
  - We will use to publish data and consume data
  - Knowledge of java is strongly proffered if you know Python it is ok
    - We will write code in Java to produce and consume data from Apache Kafka
  - Linux and mac strongly preferred or Windows if possible
    - Ability to install Docker on you machine otherwise Java jdk should be installed in your machine

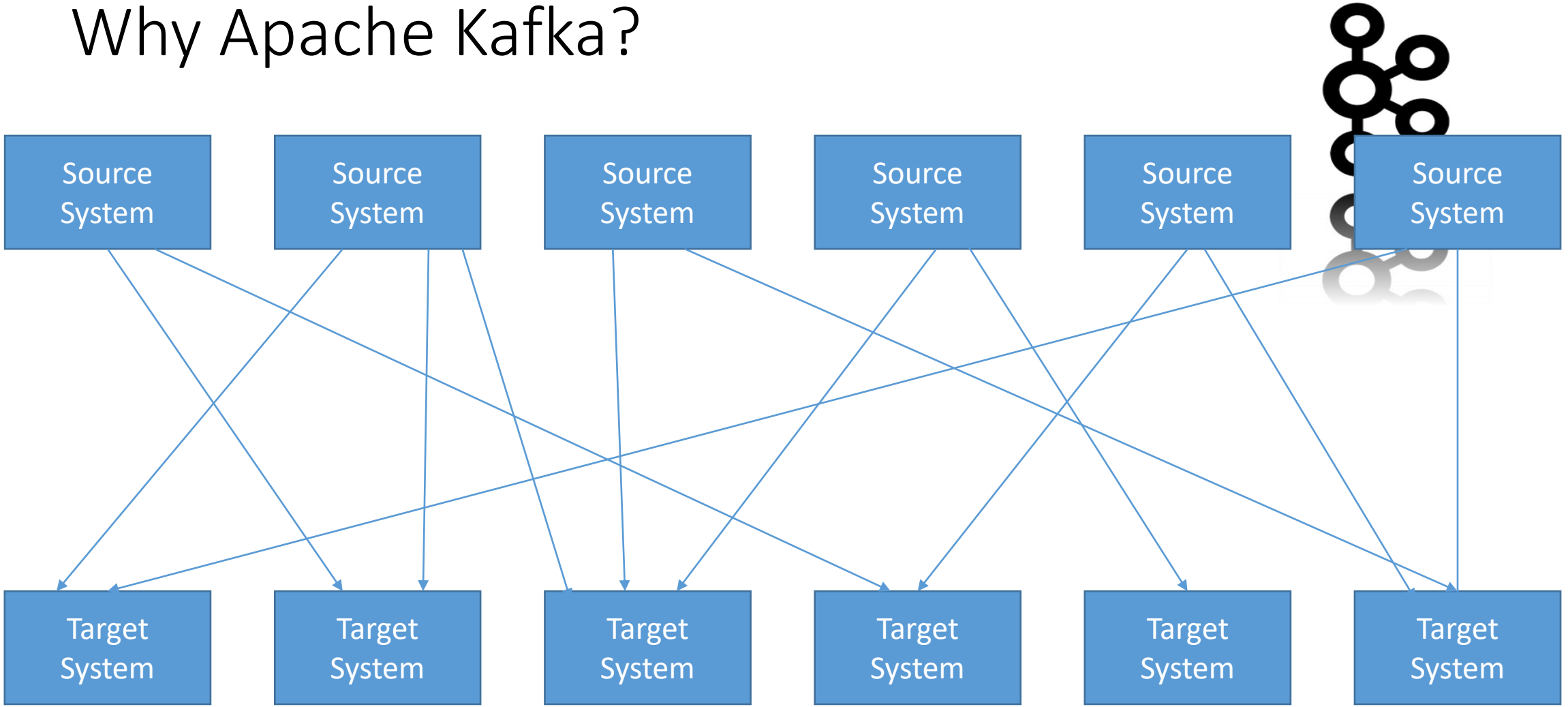
Why Apache Kafka?





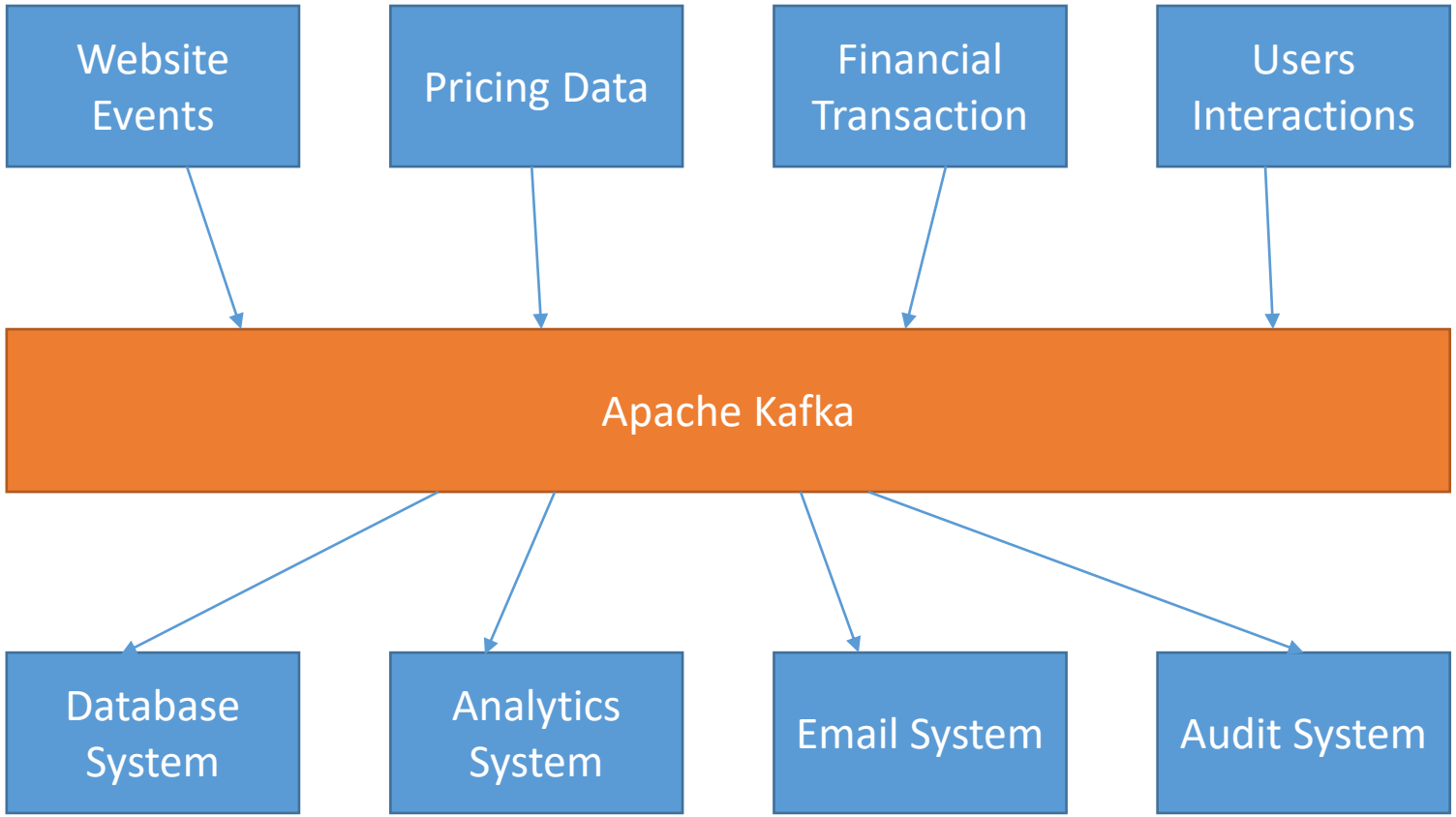
Simple in first

Why Apache Kafka?

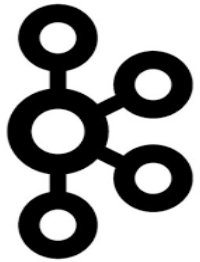


Very complicated – tedious – have syncing problen

Why Apache Kafka? Decoupling of Data streams

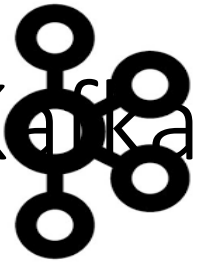


Why Kafka?



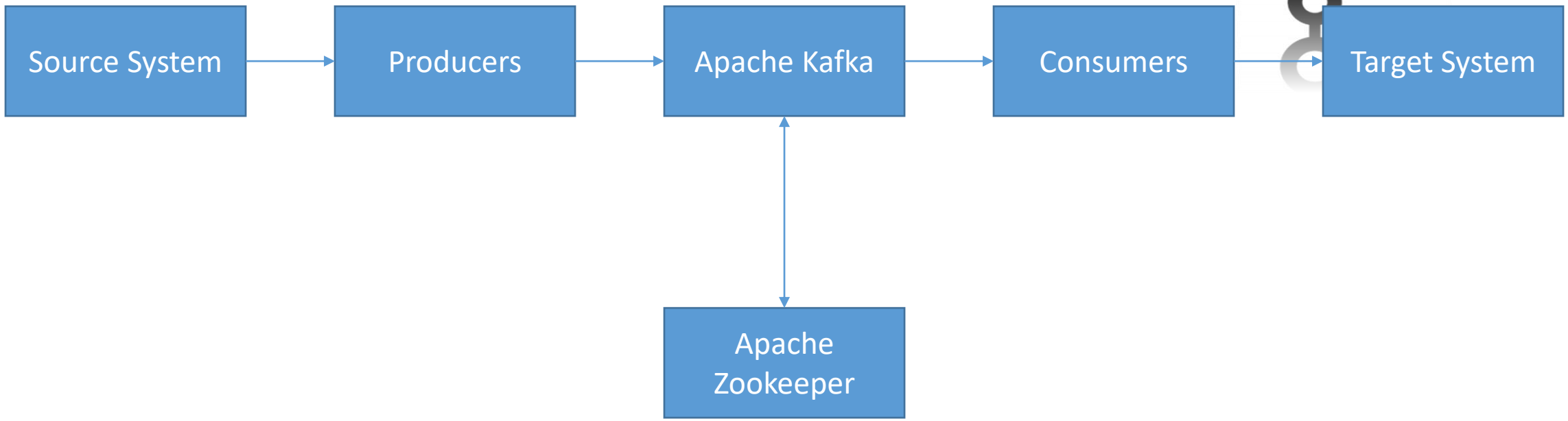
- Distributed, resilient architecture, fault tolerance
- Horizontal scalability
- High Performance (latency of less than 10 milliseconds) – real time
- Used by 2000 + firms
  - Netflix
  - LinkedIn (first created)
  - Airbnb
  - Walmart

Few metrics and Use cases of Apache Kafka

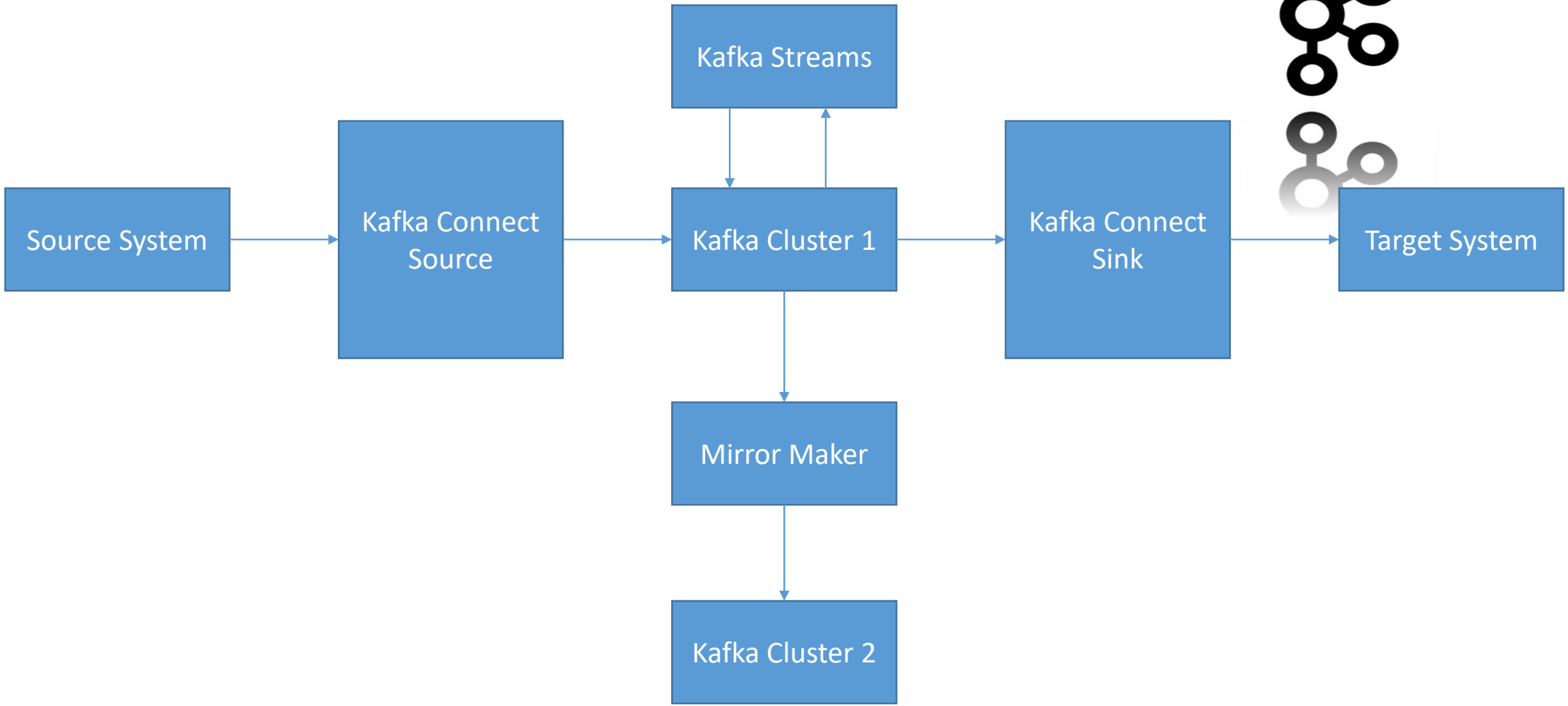


- Messaging system
- Activity Tracking
- Gathering metrics from different location
- Application logs gathering
- Stream processing (with Kafka streams apis and Spark for example)
- De-coupling of system dependencies
- Integration with Spark, Flink, Storm and Hadoop and many other Big Data Technologies

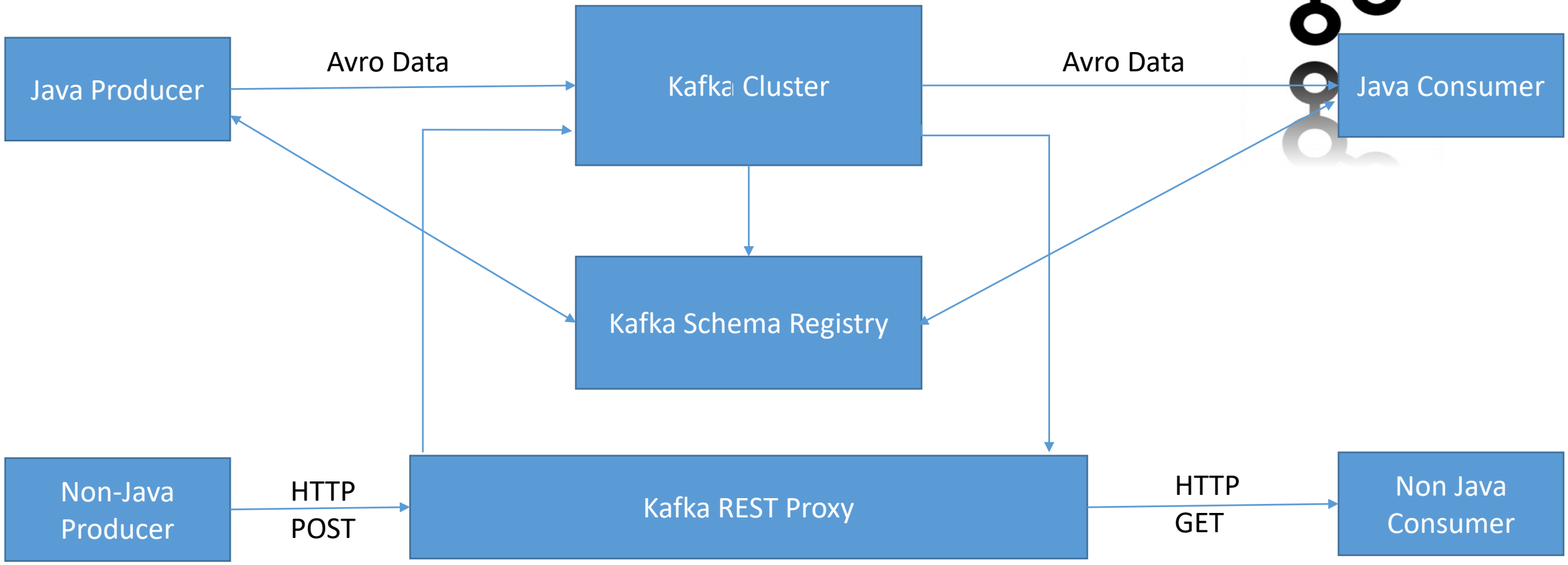
Apache Kafka – Eco-system: Kafka Core



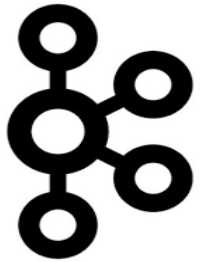
Kafka Ecosystem: Kafka extended API



Kafka Ecosystem: Confluent components, schema registry and REST Proxy extended API

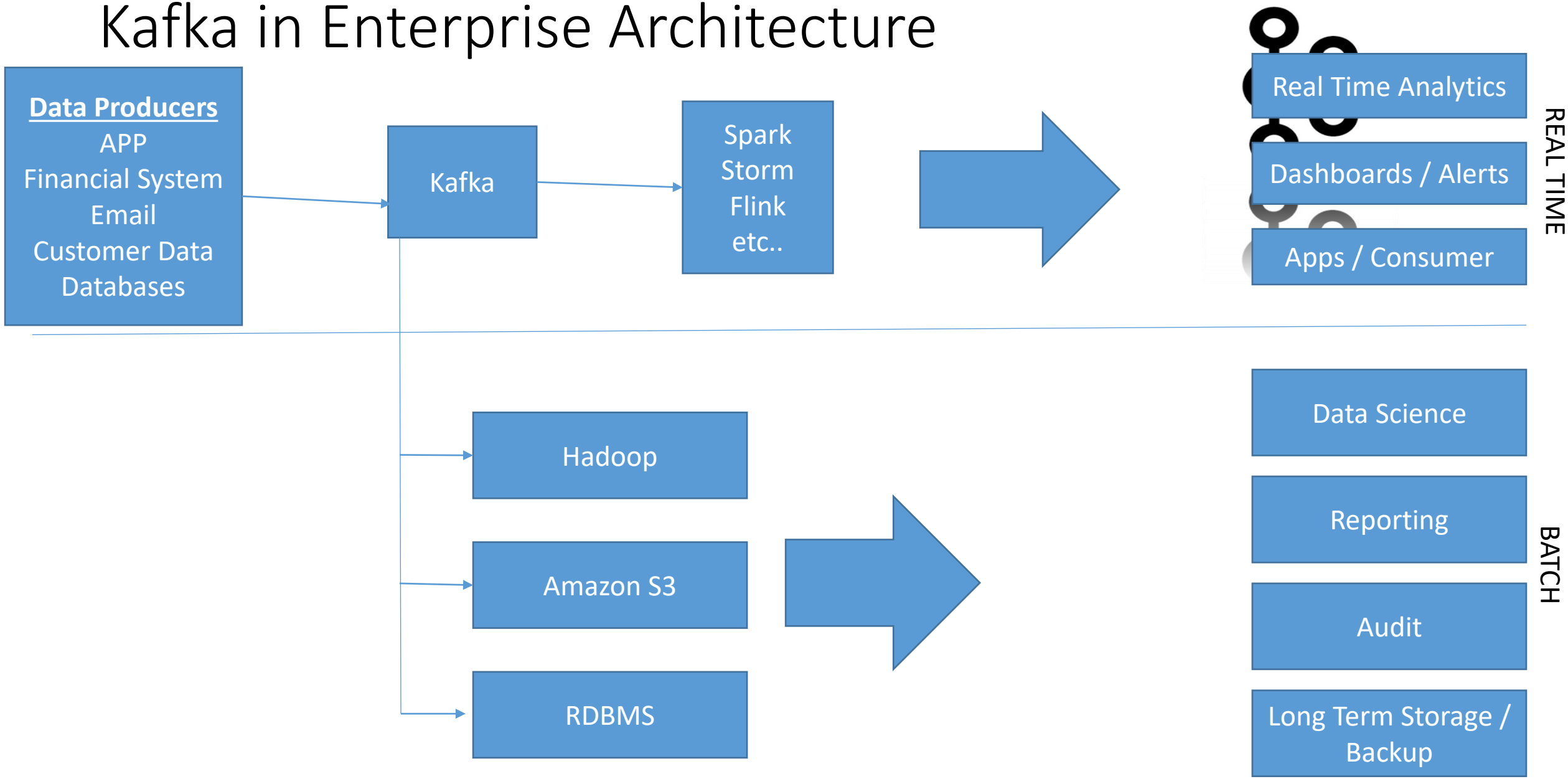


Kafka Ecosystem: Administrations and Monitoring Tools



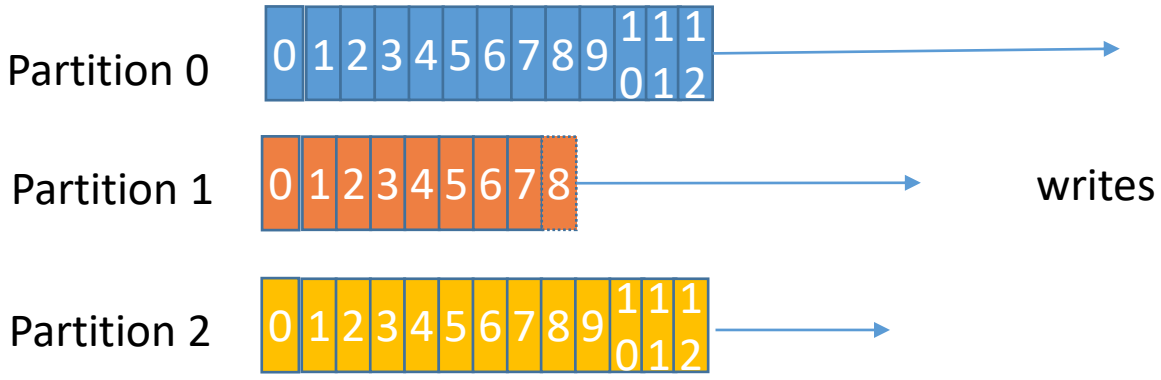
- Topics UI (Landoop): View the Topics content
- Schema UI(Landoop): Explore the schema registry
- Connect UI (Landoop): Create and monitor connect task
- Kafka Manager (Yahoo): Overall Kafka cluster management
- Burrow (LinkedIn): Apache Consumer Lag checking
- Exhibitor(LinkedIn): Zookeeper configuration, monitoring and backup.
- Kafka Tool (LinkedIn): Broker and Topics administration task simplified
- Kafkat (Airbnb): More broker and topics administration tasks simplified
- JMX Dump: JMX metrics from Broker
- Control Center/ Data Load balancer/ Replicator (Confluent): Paid Tools

Kafka in Enterprise Architecture



Topics and Partitions

- Topics: a particular streams of data
  - Similar to a table in database (without all the constraints)
  - You can have as many topics as you want
  - A topic is identified by it's name
  - Topics are split into partitions
    - Each partitions is ordered
    - Each message within a partition will get an incremental id, that number is called as offset called *offset*



Topics and Partitions

- Offset only have meaning for a specific partitions *e.g. offset 3 in partition 1 does not represents the same data in offset 3 in partition 2*
- Order is guaranteed only within partitions not across partition
- Data kept only for a limited time only for two weeks.
- Once the data is written into a partition it can not be change *(immutability)*
- *Data* is assigned randomly to a partition unless a key is provided (more details later)
- You can have as many partitions per topic as you want.

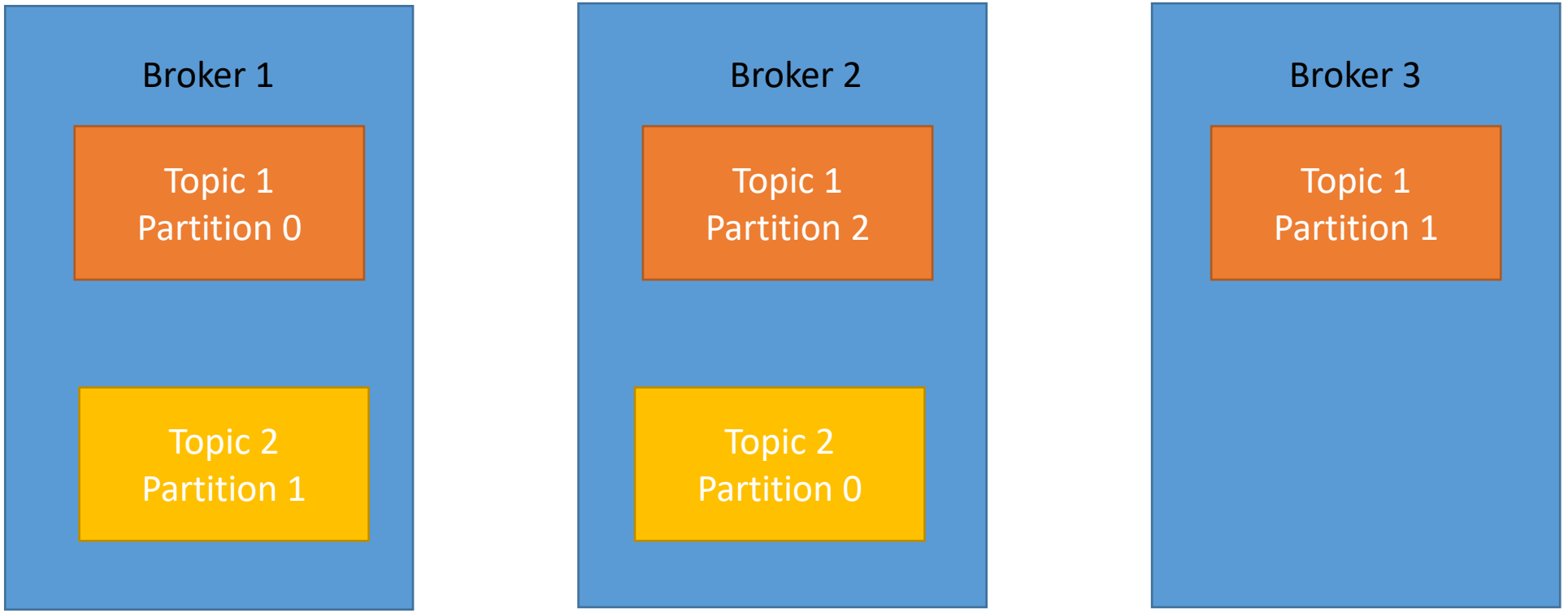
Brokers

- A Kafka clusters is composed of multiple broker (servers).
- Each broker is identified with its ID (integer)
- Each broker contains certain Topic partitions.
- After connecting to any broker (called as a bootstrap broker) you will be connected to entire cluster.
- A good number to started is 3 brokers, but a big cluster can have upto 100 brokers this show how much Kafka can scale horizontally

Brokers and Topic

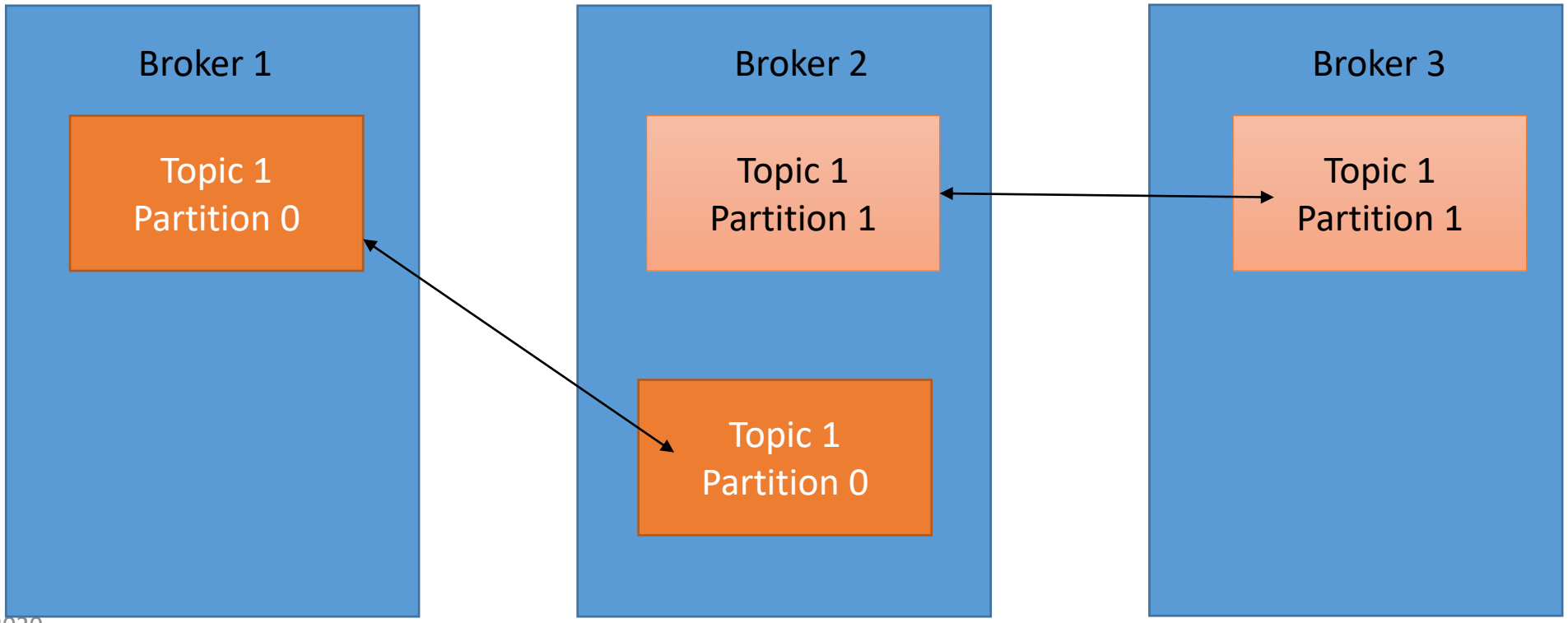
Examples of 2 topics (3 partitions and 2 partitions)

Data is distributed Broker 3 does not have nay Topic 2 data



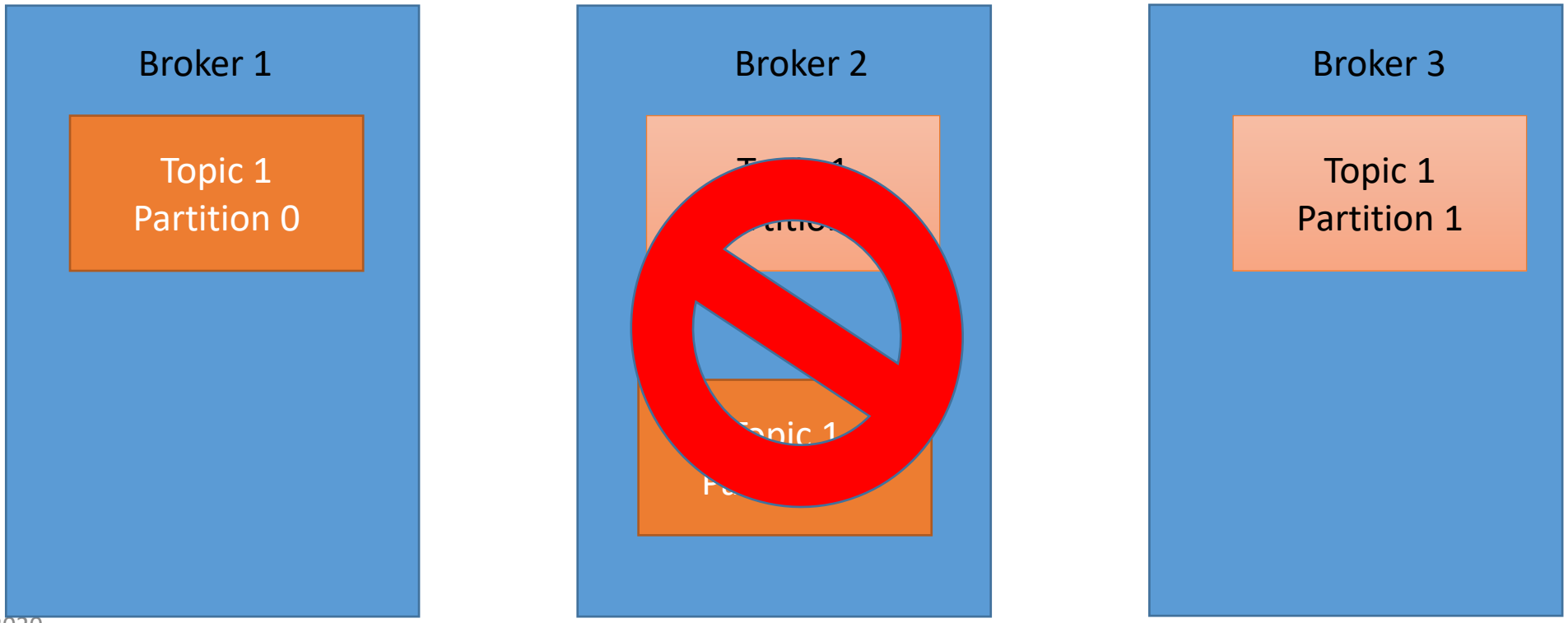
Topic replication factors

- Topic should have replication factor > 1 (usually between 2 and 3)
- This way if any broker is down then another broker can serve the data
- Example: Topic with 2 partitions and replication factor of 2.



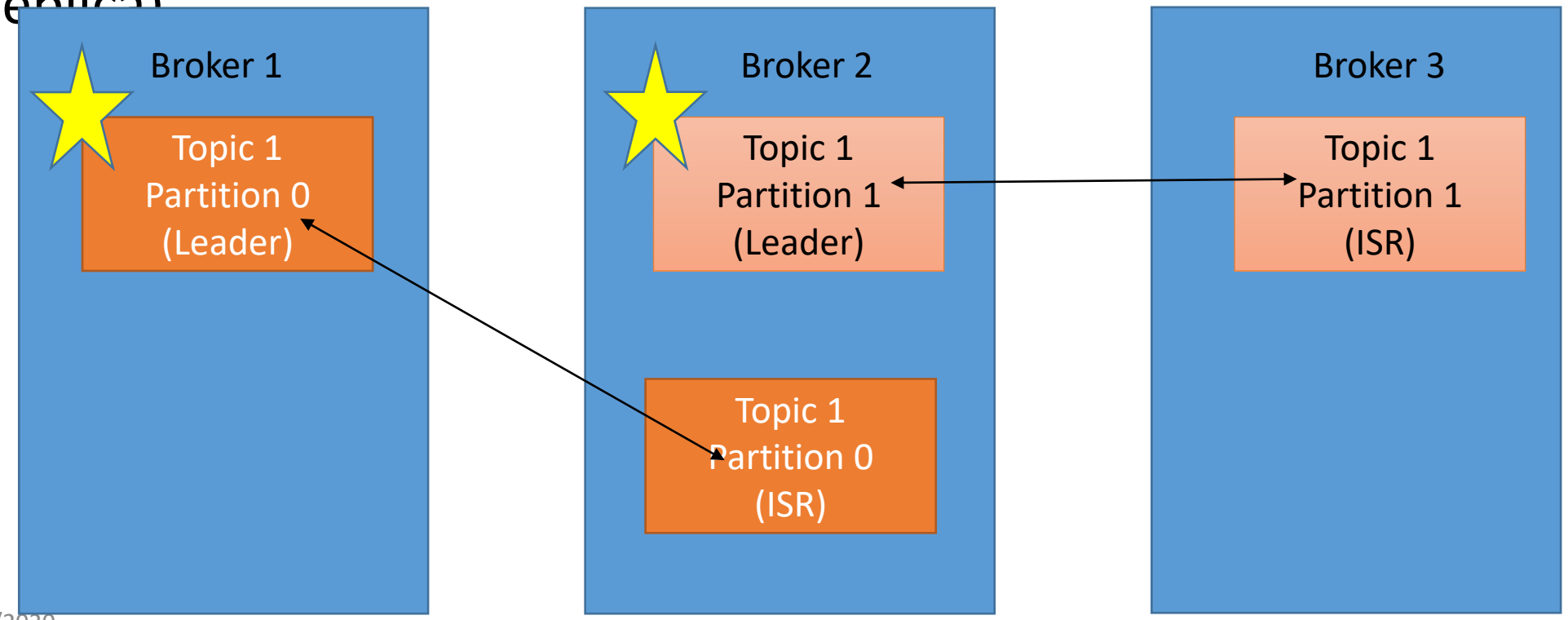
Topic replication factors

- Example: we lost the broker 2
- Result: broker 1 and broker 3 still serve the data



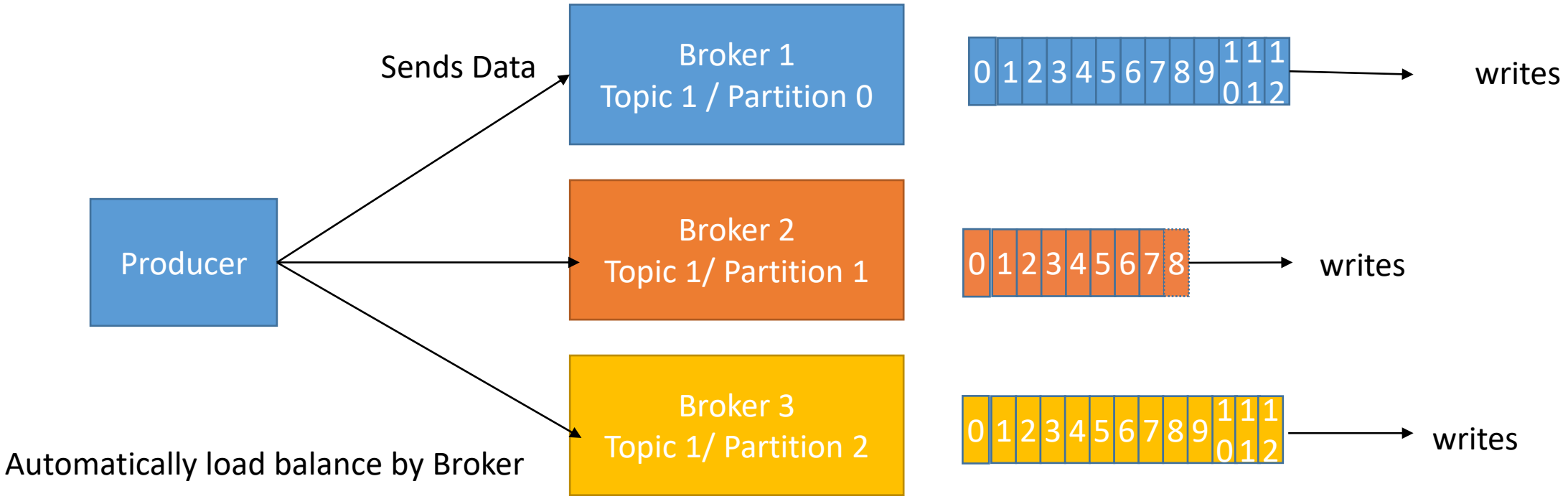
Concept of Leader for a partition

- At any time only 1 broker can be a leader for a given partition.
- Only that leader can receive and serve data for a partition.
- The other broker will synchronize and data
- There each partition has: only one leader and multiple ISR (in-sync replica)



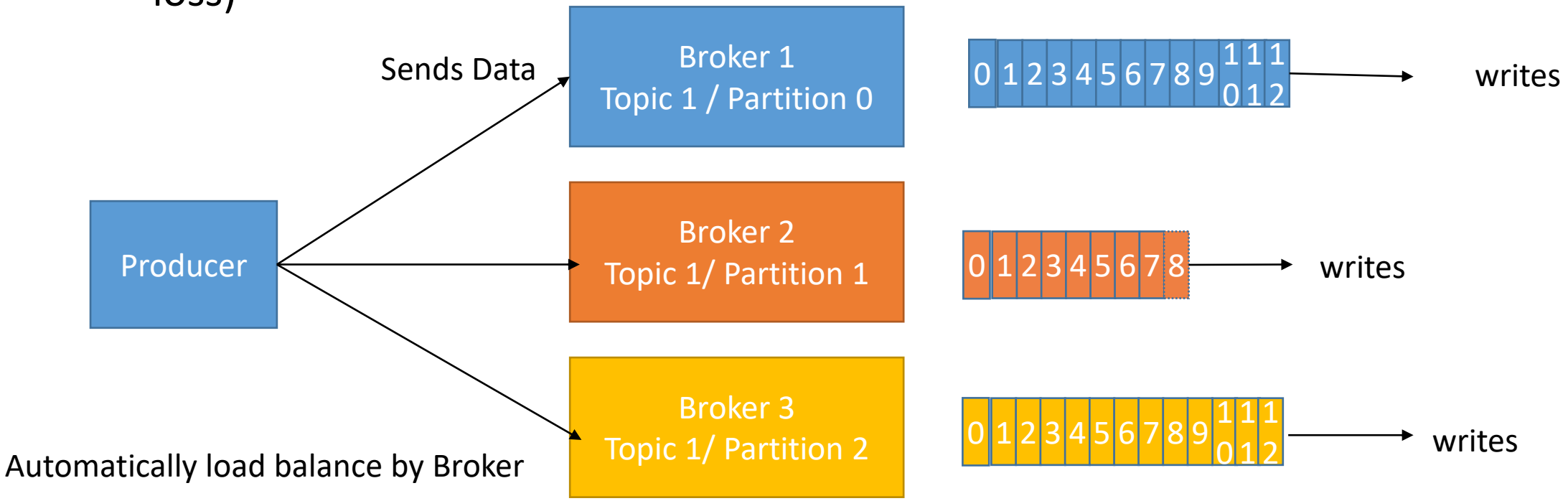
Producers

- Producers: writes data to the Topic
- ***They only have to specify the topic name and one broker to connect to and Kafka will automatically take care of routing the data to the right brokers.***



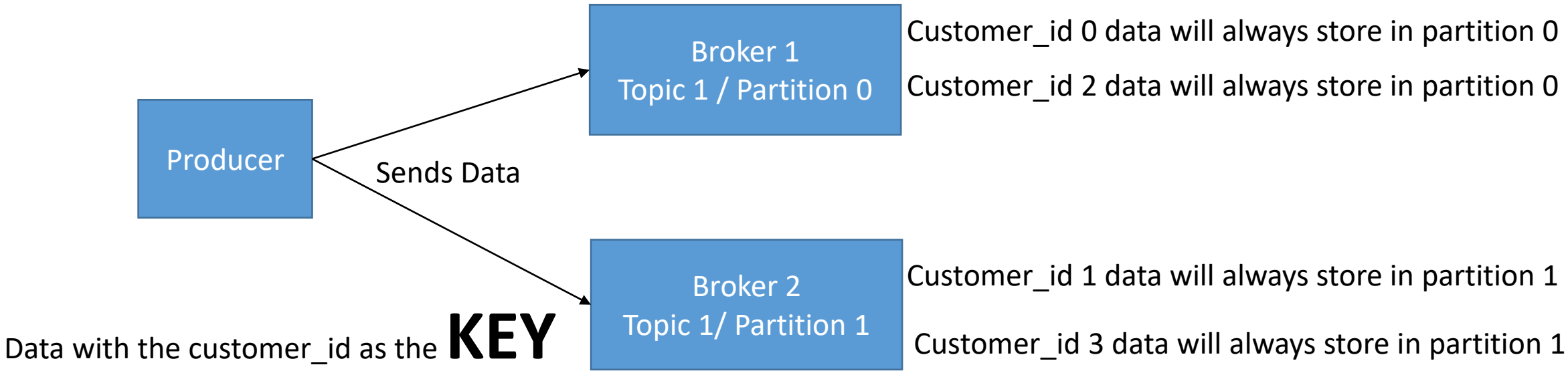
Producers

- Producer can choose to receive the acknowledgement of data writes:
  - Acks=0: Producer won't wait for acknowledgment (possible data loss)
  - Acks=1: Producer will wait for leader acknowledgment (limited data loss)
  - Acks=All: Producer will wait for Leader + replicas acknowledgement (no data loss)



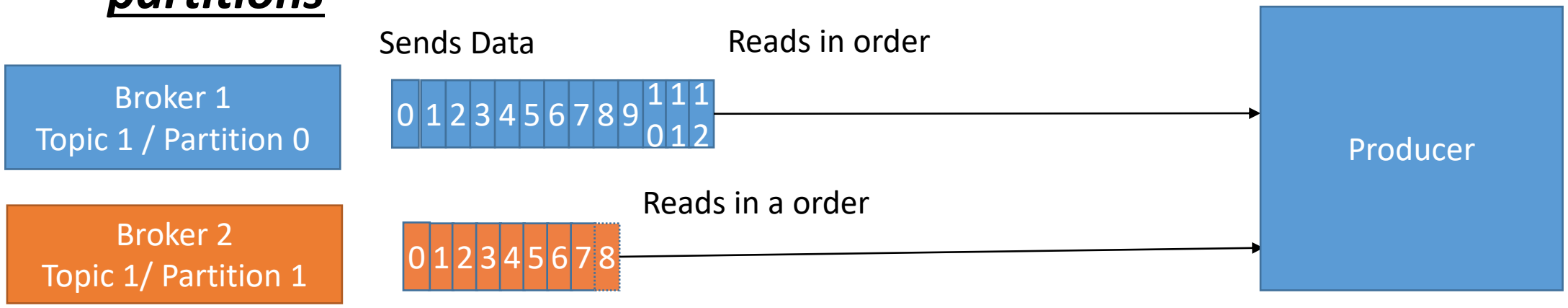
Producers: Messages Key

- Producer can choose to send the key with the message
  - If the key is send then the Producer has the guaranteed that all the messages for that key will always got to same partition *that allows you guaranteed ordering*
  - This enables guaranteed ordering for a specific key



Consumers

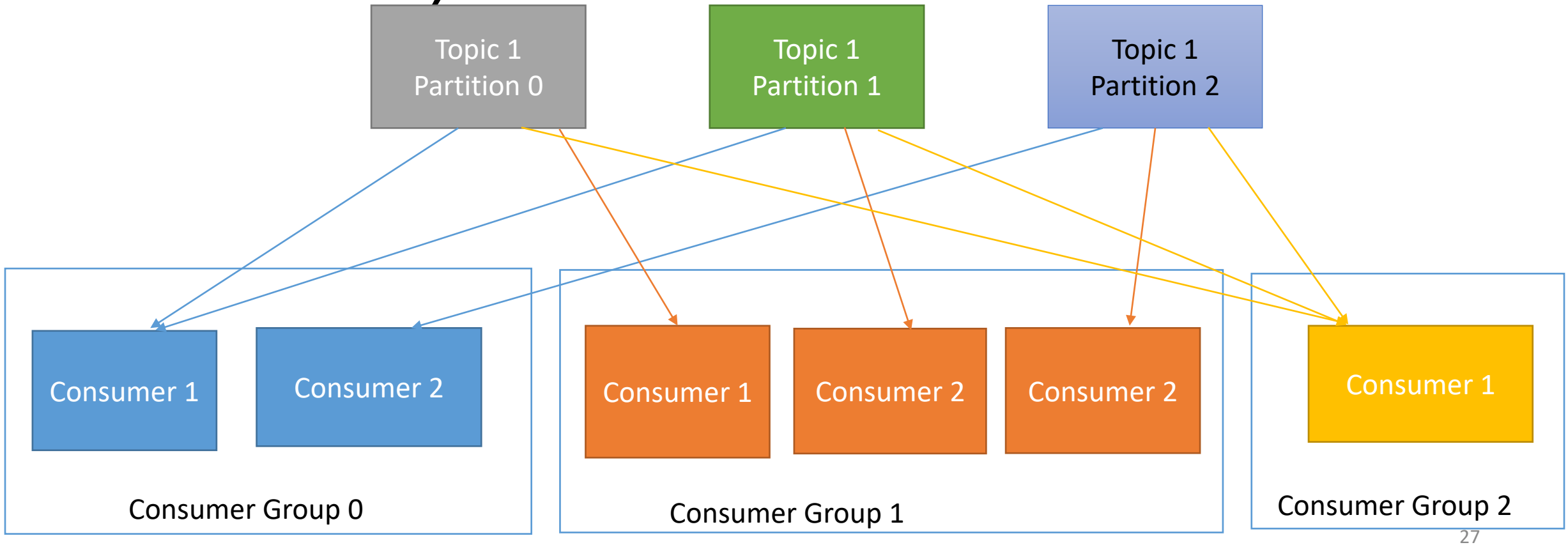
- Consumers: reads data from a topic
- ***They only have to specify the topic name and one broker to connect to and Kafka will automatically take care of pulling of data from the right brokers.***
- ***Date read in a order from each partitions and in parallel across the partitions***



Reads in a order within a partitions but read data par alley across the partitions

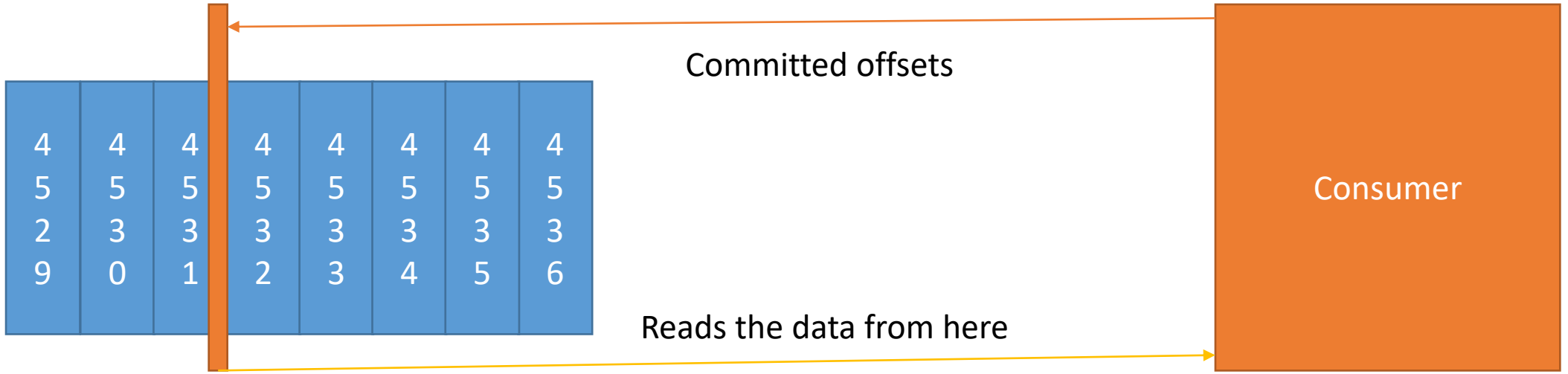
Consumer Groups

- Consumers reads data in a consumer groups
- ***Each consumer within a group reads from exclusive partitions***
- ***You can not have more consumers than partitions (otherwise some will be inactive)***



Consumer Offsets

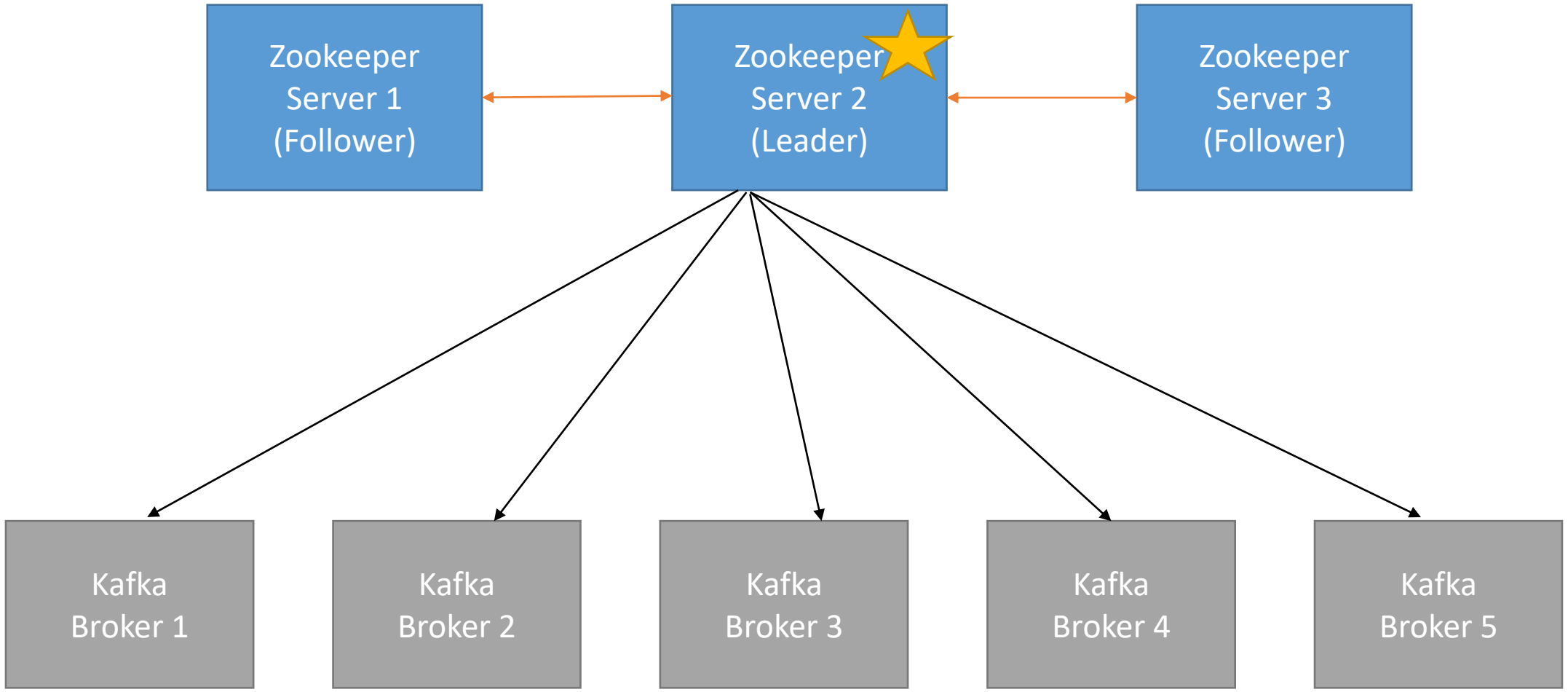
- Consumers stores the offsets at which a consumer group has been reading.
- The offsets commits live in Kafka topic names “__consumer_offsets”
- When a consumer has processed data received from KAFKA, it should be committing the offset
- If consumer process dies, it will able to read back the data from where it left off, thanks to consumer offset!



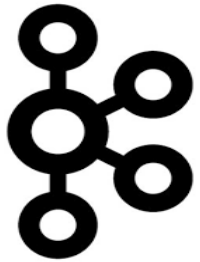
Zookeeper

- Zookeeper manages brokers (keeps a list of them)
- Zookeeper helps is in performing the leader election for partitions
- Zookeeper sends notification to Kafka in case on changes (e.g. new topic, brokers dies, broker comes up, topic deletes, etc..)
- **Kafka can’t work without Zookeeper**
- Zookeeper usually operates in a odd quorum of clusters of servers (3, 5 and 7)
- Zookeeper has a leader, rest of the servers are followers

Zookeeper and Kafka Brokers orchestration



Kafka guarantees



- Messages are appended to a topic-partition in the order they sent
- Consumers read the messages in the order stored in the partition
- With the replication factor of N, producers and consumers can tolerate up to N-1 brokers being down
- This is why replication factors is 3 is a good idea
  - Allows for 1 broker to be taken down for maintenance
  - Allow for another broker to be take down unexpectedly
- As long a number of partitions remain constant for topic (no new partitions) the same key will go to same partitions

Delivery semantics for consumers

- As learned before, consumers choose when to commit offsets (there is at least 3 situations)
- **At most once:** offsets are committed as soon as a message is received, if processing goes wrong, the message will be lost (it can't be read again)
- **At least once:** offsets are committed after the message is processed. If processing goes wrong, the message will be read again, this can result in duplicate processing of the messages. Make sure your processing is **idempotent** (i.e. processing again the messages won't impact your system) e.g. insert and upsert example first time inserted and second time it will be updated.
- **Exactly once:** Very difficult to achieve, require very strong in engineering
- **Bottom line:** Most often you should use at least once processing and ensure your transformation / processing is idempotent