Message Distribution Systems: Apache Kafka

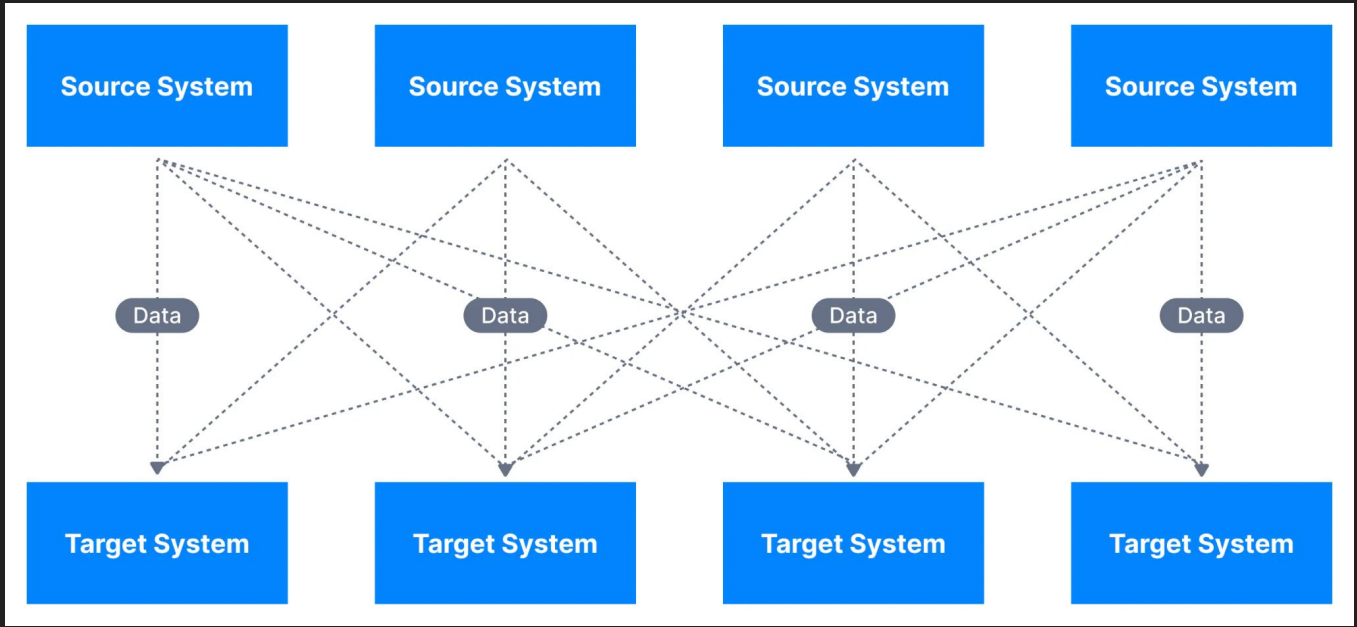
Programs are composed of objects and interactions between them.

In a web-based platform, numerous components interact with each other.

For the system on the right, we need to implement up to 16 integrations.

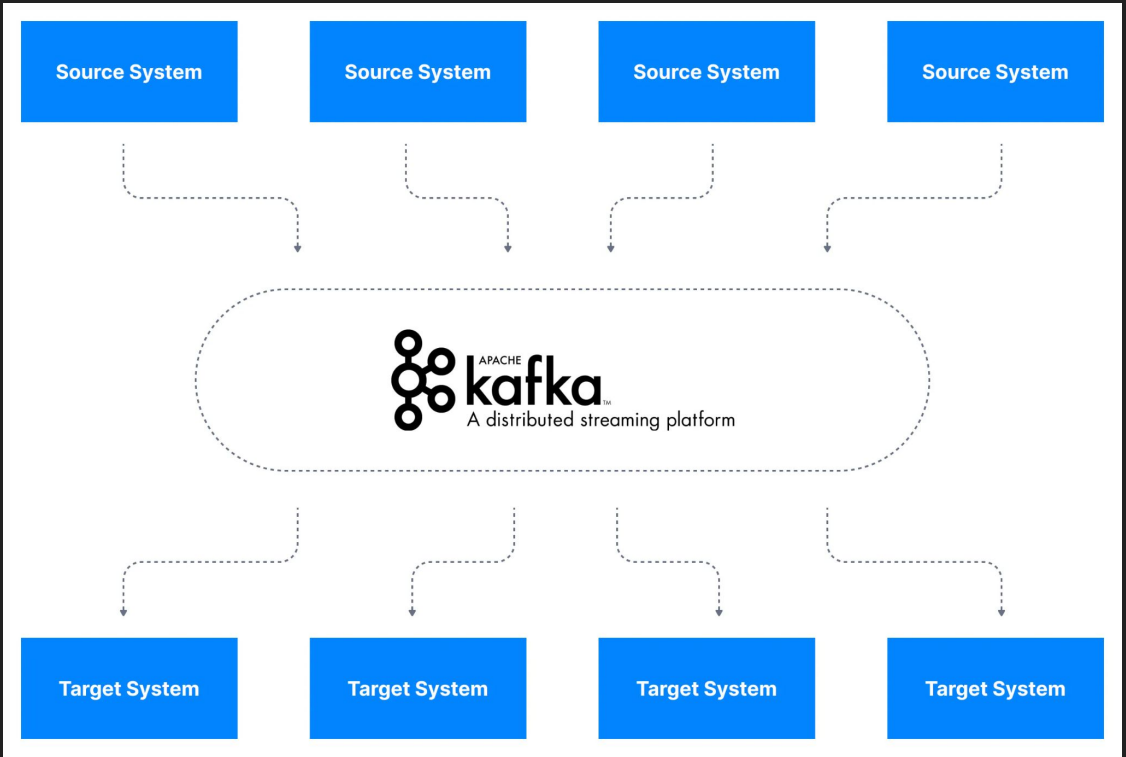
Components may use various protocols (HTTP, REST, SOAP, XML-RPC, etc.), data formats (JSON, XML, Avro, binary, CSV, etc.), and schemas.

These integrations increase system complexity and make maintenance challenging.



https://learn.conduktor.io/kafka/what-is-apache-kafka/

Message Distribution Systems: Apache Kafka



https://learn.conduktor.io/kafka/what-is-apache-kafka/

The solution is the integration of messaging systems, which involves separating messaging responsibilities from components. This approach promotes:

- Separation of concerns
- Single responsibility
- Dependency inversion

Apache Kafka is:

- A distributed messaging system
- Resilient with fault-tolerant architecture
- Scalable
- Low-latency, high-throughput
- Open source

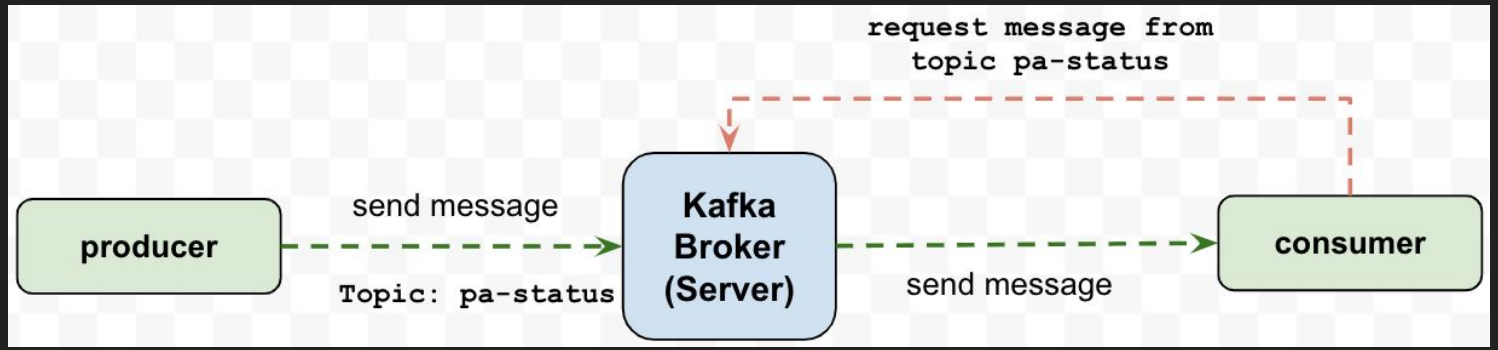
Many large companies, such as LinkedIn, Uber, Netflix, and Walmart, use it in their backbone systems as a messaging solution.

Apache Kafka: Terminology

In an Kafka messaging system, there are four key components:

Producer: A client that writes (publishes) messages to Kafka topics.
Consumer: A client that reads messages from Kafka topics.
Broker: A Kafka server that receives messages from producers, stores them, and serves them to consumers.
Zookeeper: Manages and coordinates the Kafka cluster, tracking brokers and ensuring scalability and fault tolerance.

All messages between producers and consumers are transferred via **topics** managed by the broker.



kafka-consumer.js kafka-producer.js

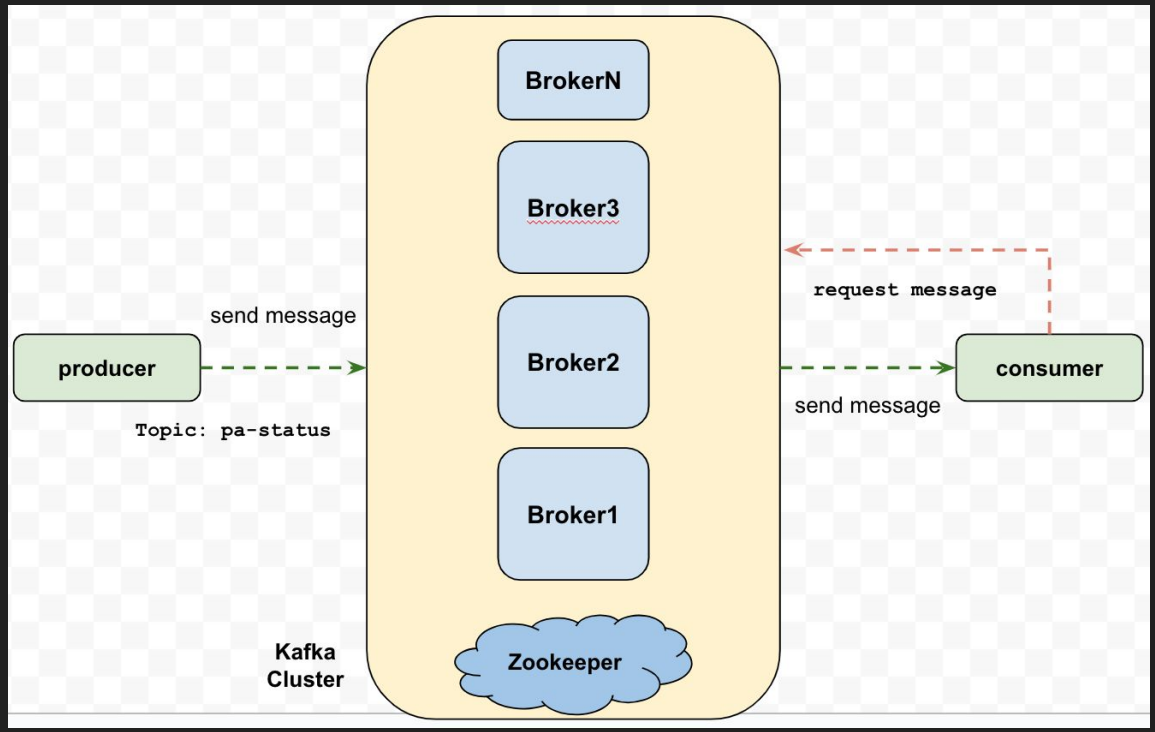
Apache Kafka: Cluster

A **Kafka cluster** is built to ensure **scalability, fault tolerance, and high availability**. By distributing data across multiple brokers, the cluster handles high volumes of data streams, provides redundancy in case of broker failures, and allows for seamless horizontal scaling to manage growing workloads efficiently. This makes Kafka ideal for handling large-scale, real-time data pipelines.

Assume our system currently supports data traffic for up to 100 clients. If we want to scale this to 1,000 clients, several challenges arise for non-scalable systems:

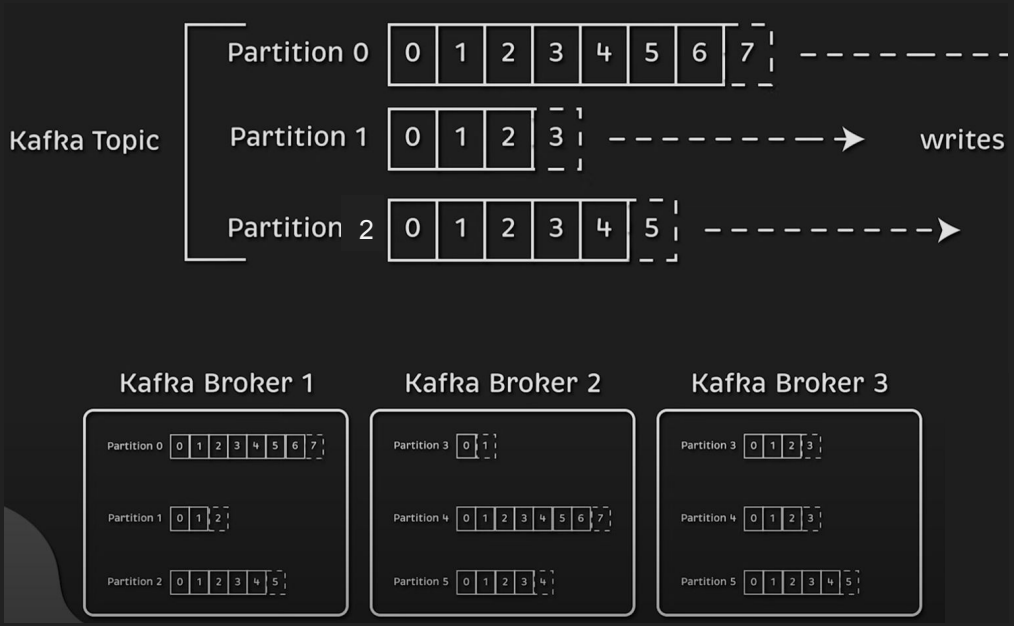
- higher latency
- increased response time
- system crashes due to resource liimitations

in scalable systems, these challenges can be mitigated by distributing the load efficiently and dynamically allocating resources to handle the increased demand.

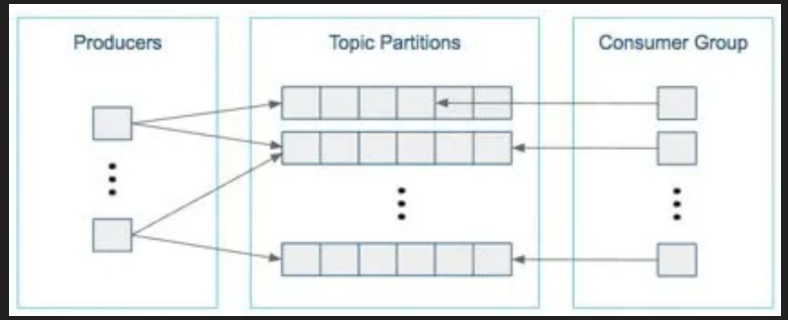


Apache Kafka: Topic Partitions And Parallelism

In Kafka, topics can split into partitions, which allow messages to be distributed across multiple brokers. This partitioning enables parallelism by allowing producers to write and consumers to read from multiple partitions simultaneously. Each partition can be processed independently, improving scalability and throughput by enabling concurrent processing of data. More partitions mean more parallelism, enhancing performance in distributed systems.



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

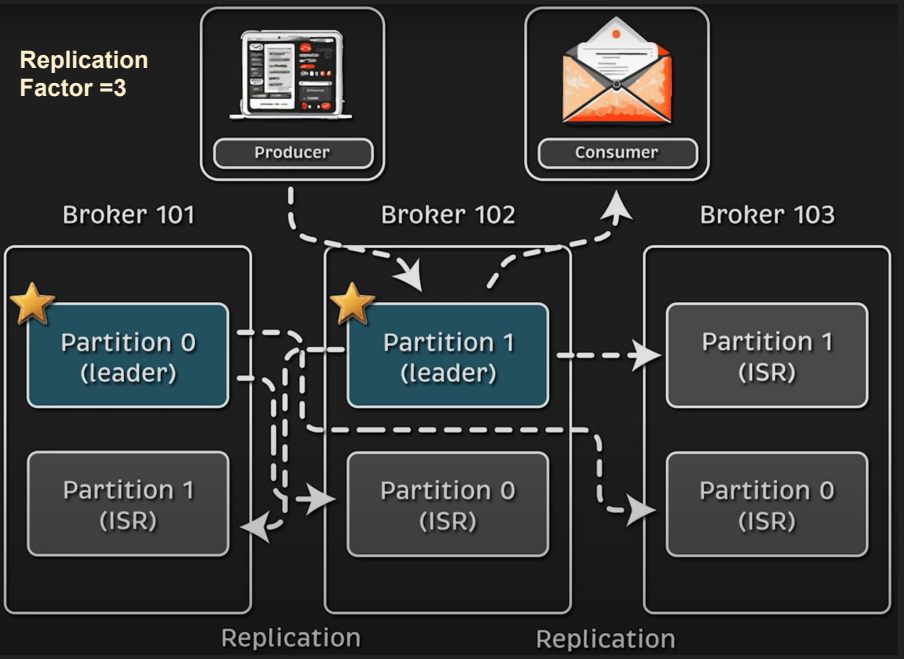


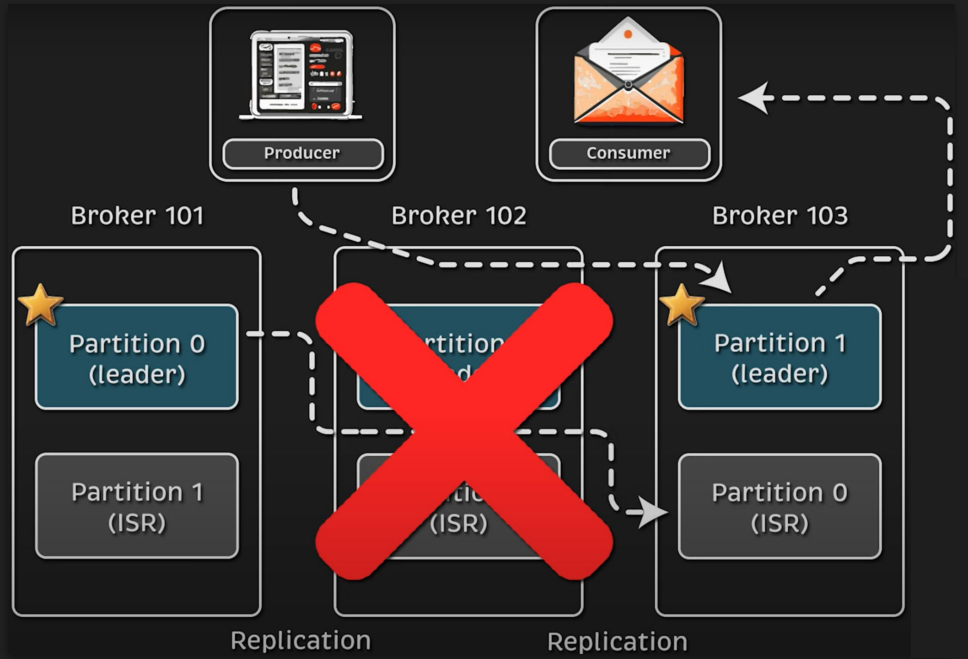
https://nostdahl.com/2022/02/12/administration-of-kafka/

Apache Kafka: Partition Replication And Fault Tolerance

Kafka ensures fault tolerance through partition replication. Each partition is replicated across multiple brokers, creating copies called replicas. If a broker fails, one of the replicas automatically takes over as the leader, ensuring no data is lost and the system remains operational. This replication mechanism is key to Kafka's ability to provide high reliability and data durability in distributed environments.

Replication Factor ≤ Number of Brokers





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

Setting up a Cluster

Installation of Apache Kafka Cluster with 3 brokers

Broker 1 (server-1.properties)

```
broker.id=1
log.dirs=/tmp/kafka-logs-1
zookeeper.connect=localhost:2181
listeners=PLAINTEXT://localhost:9092
```

Broker 2 (server-2.properties)

```
broker.id=2
log.dirs=/tmp/kafka-logs-2
zookeeper.connect=localhost:2181
listeners=PLAINTEXT://localhost:9093
```

Broker 3 (server-3.properties)

```
broker.id=3
log.dirs=/tmp/kafka-logs-3
zookeeper.connect=localhost:2181
listeners=PLAINTEXT://localhost:9094
```

Starting the Kafka Cluster

```
bin/zookeeper-server-start.sh config/zookeeper.properties &
bin/kafka-server-start.sh config/server-1.properties &
bin/kafka-server-start.sh config/server-2.properties &
bin/kafka-server-start.sh config/server-3.properties &
```

Topic partition and replication factor

```
bin/kafka-topics.sh --create --topic
dss-module3-processing --bootstrap-server
localhost:9092 --replication-factor 3 --partitions 3
```

Initializing a Consumer Group

Installation of Apache Kafka Cluster with 3 brokers

Consumer 1.1 - A
./kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic dss-module2-exercise --**group** employee-consumer-group1
Consumer 1.2 - A
./kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic dss-module2-exercise --**group** employee-consumer-group1

Consumer 2.1 - B
./kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic dss-module2-exercise --**group** employee-consumer-group2
Consumer 2.2 - B
./kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic dss-module2-exercise --**group** employee-consumer-group2

There are two consumer groups, A and B, both subscribed to the same topic. Since they belong to different groups, each message is delivered to both groups, effectively creating two replicas of the data.

Within a group, Kafka ensures that each partition's messages are processed by only one consumer at a time, enabling parallelism. This means that while all consumers in Group A share the workload, no two consumers in A receive the same message. The same applies to Group B, ensuring efficient message distribution.

If there are more consumers than partitions, some consumers will remain idle.

- **Different groups get the same data (replication).**
- **Within a group, only one consumer gets a message from a given partition (parallelism).**
- **The Consumer Group Coordinator decides partition assignments, ensuring balanced distribution.**