Why Kafka ?
Kafka Motivation

1https://engineering.linkedin.com/blog/2016/04/kafka-ecosystem-at-linkedin
Kafka at Linkedin

1https://engineering.linkedin.com/blog/2016/04/kafka-ecosystem-at-linkedin
Kafka at Linkedin

- 1400 brokers
- 1.4 trillion messages/day
- AVRO encoded records
- Multiple uses:
  - Message queue/data bus
  - Database replication
  - Metrics
  - Logging
  - Web app tracking data
  - Real-time Aggregation

¹https://engineering.linkedin.com/blog/2016/04/kafka-ecosystem-at-linkedin
Microservices Advantages

- Fast value delivery
  - Fixes
  - New features
  - Experiments
  - Increased confidence

- Language independent

- Fault isolation

- Pair well with containers

- Scalability

- Flexibility
Event Driven Architecture

- Microservice Ordering
- Microservice Billing
- Microservice Inventory
- Microservice Shipping

Kafka

Event Streams Platform
What Is Kafka?
Kafka Fundamentals

- **Records/Messages** → have a key, value, and timestamp
- **Topic** → a stream of records (“/orders”, “/user-signups”), feed name
  - Log → topic storage on disk
  - Partition → topics are divided into a fixed number of partitions, amongst which the records are divided.
- **Producer** → produces a stream of records
- **Consumer** → consumes a stream of records
- **Broker** → Kafka server that runs in a Cluster
- **Cluster**: A group of Brokers. This is also called a Kafka Cluster
- **ZooKeeper**: Coordinates brokers within the cluster.
Kafka Components: Producers, Consumers, Topics, etc

The diagram illustrates the components of a Kafka Cluster, including Producers, Topics, and Consumers. Each Topic is divided into multiple Partitions, and the Producers publish messages to these Topics, which are then consumed by the Consumers.
Kafka Messages

• The basic unit of data in Kafka is a message also known as record

• A message is a key-value pair
  o All data is stored in Kafka as byte arrays
  o Producer provides serializers to convert the key and value to byte arrays
  o Key and value can be any data type
Data Order in Kafka
What Kafka Guarantees

Messages sent by a producer to a particular topic partition will be appended in the order they are sent.

A consumer instance sees records in the order they are stored in the log.

For a topic with replication factor $N$, we will tolerate up to $N-1$ server failures without losing any records committed to the log.
What Kafka Guarantees

Producers writes
reads

Consumer A (offset=9)
Consumer B (offset=11)
What Kafka Guarantees

Anatomy of a Topic

Partition 0
0 1 2 3 4 5 6 7 8 9 1 1 1 0 1 2

Partition 1
0 1 2 3 4 5 6 7 8 9

Partition 2
0 1 2 3 4 5 6 7 8 9 1 1 1 0 1 2

Writes

Old

New
What happens if we send data to many partitions?
10 Partitions

```
./kafka-topics.sh --create \
--zookeeper localhost:2181/kafka \
--replication-factor 1 --partitions 10 \
--topic my-topic
```
/kafka-console-producer.sh \ 
--broker-list localhost:9092 \ 
--topic my-topic 

> 1
> 2
> 3
> 4
> 5
> 6
> 7
> 8
> 9
> 10
/kafka-console-consumer.sh \
--bootstrap-server localhost:9092 \
--topic my-topic \
--from-beginning

> 2
> 5
> 9
> 3
> 4
> 7
> 6
> 10
> 8
> 1
How did this happen?

<table>
<thead>
<tr>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 &gt;1</td>
</tr>
<tr>
<td>P1 &gt;5</td>
</tr>
<tr>
<td>P1 &gt;8</td>
</tr>
<tr>
<td>P1 &gt;10</td>
</tr>
<tr>
<td>P1 &gt;4</td>
</tr>
</tbody>
</table>

| P1 >2 |
| P1 >7 |
| P1 >9 |
| P1 >3 |
| P1 >6 |

Data Sent Round Robin

Consumer
What happens if we send data to one partition?
1 Partition

./kafka-topics.sh --create \
--zookeeper localhost:2181/kafka \
--replication-factor 1 --partitions 1 \
--topic my-topic
Send Some Data

./kafka-console-producer.sh \ 
--broker-list localhost:9092 \ 
--topic my-topic

> 1
> 2
> 3
> 4
> 5
> 6
> 7
> 8
> 9
> 10
Receive the Same Data

./kafka-console-consumer.sh \ 
--bootstrap-server localhost:9092 \ 
--topic my-topic \ 
--from-beginning

> 1
> 2
> 3
> 4
> 5
> 6
> 7
> 8
> 9
> 10
How did this happen?

Consumer

Data Sent Round Robin to all partitions

Topic

P1
>1
>2
>3
>4
>5
>6
>7
>8
>9
>10
What happens if we introduce keying?
Kafka messages with a key

All Key messages with a Key will go to the same partition
Suppose we want to send 4 messages to a Kafka topic with 2 partitions.

- **Key: Costco**
  - Value: 400

- **Key: Walmart**
  - Value: 400

- **Key: Target**
  - Value: 400

- **Key: BestBuy**
  - Value: 400
Keys are hashed and distributed across the cluster

Orders Topic

Partition 1

Key: Costco
Value: 400

Key: Walmart
Value: 400

Partition 2

Key: Target
Value: 400

Key: BestBuy
Value: 400
Suppose we want to send 4 MORE messages to a Kafka topic with 2 partitions.

Orders Topic

Partition 1

Key: Costco
Value: 100

Key: Walmart
Value: 200

Key: Target
Value: 100

Key: BestBuy
Value: 200

Partition 2

Key: Costco
Value: 400

Key: Walmart
Value: 400

Key: Target
Value: 400

Key: BestBuy
Value: 400
Messages are sent to the same partition using the existing key
Suppose we add more partitions to the cluster

Partition 1
Key: Costco
Value: 400
Key: Walmart
Value: 400
Key: Costco
Value: 100
Key: Walmart
Value: 200

Partition 2
Key: Target
Value: 400
Key: BestBuy
Value: 400

Partition 3
Key: Target
Value: 100
Key: BestBuy
Value: 200

Orders Topic
Suppose we then decide to rebalance the partitions.
Suppose we want to send **4 MORE messages** to a kafka topic with **3 partitions**

Orders Topic

Partition 1

Partition 2

Partition 3

Key: Costco  
Value: 500

Key: Walmart  
Value: 500

Key: Target  
Value: 500

Key: BestBuy  
Value: 500
Suppose we then decide to rebalance the partitions again.
The result looks like Balanced Partitions

Orders Topic

Partition 1
Key: Costco Value: 400
Key: Costco Value: 100
Key: Costco Value: 500

Partition 2
Key: Target Value: 400
Key: Target Value: 100
Key: Target Value: 500

Partition 3
Key: BestBuy Value: 400
Key: BestBuy Value: 200
Key: BestBuy Value: 500

Partition 4
Key: Walmart Value: 400
Key: Walmart Value: 200
Key: Walmart Value: 500
How do make sure the data is always sent in order?
Kafka Producer Overview

1. **ProducerRecord**
   - Topic
   - (Partition)
   - (Key)
   - Value

2. **Serializer**
3. **Partitioner**
4. **Topic A Partition 1**
   - Batch 1
   - Batch 2
   - Batch 3
5. **Topic B Partition 2**
   - Batch 1
   - Batch 2
   - Batch 3

- **If Fail on Retry, throw exception**
- **If Success, return information**
- **Retry?**
  - Yes
  - No
- **Fail?**
max.in.flight.requests.per.connection

Setting the retries parameter to nonzero and the `max.in.flight.requests.per.connection` to more than one means that it is possible that the broker will fail to write the first batch of messages, succeed to write the second (which was already in-flight), and then retry the first batch and succeed, thereby reversing the order.
Usually, setting the number of retries to zero is not an option in a reliable system, so if guaranteeing order is critical, we recommend setting `in.flight.requests.per.session = 1` to make sure that while a batch of messages is retrying, additional messages will not be sent (because this has the potential to reverse the correct order).

This will severely limit the throughput of the producer, so only use this when order is important.
**Kafka Delivery Guarantees**

<table>
<thead>
<tr>
<th>Guaranee</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>At Once</strong></td>
<td>Guarantees that a particular message will always be delivered.</td>
</tr>
<tr>
<td><strong>At Least Once</strong></td>
<td>Guarantees that a particular message will always be delivered.</td>
</tr>
<tr>
<td><strong>Exactly Once</strong></td>
<td>Guarantees that all messages will always be delivered exactly once.</td>
</tr>
</tbody>
</table>
Exactly Once Delivery

- Idempotent Producer
- Transactions Across Partitions
- Transactional Consumer
Kafka Transaction Example Workflow

**Transaction Log**
- **init**
  - 2) bump epoch

- **ongoing**
  - 4) add partitions

- **prepare**
  - 7) prepare

- **committed**
  - 9) committed

**Transaction Coordinator**
- 1) register
- 3) add partitions to transaction
- 6) commit transaction

**Producer**
- 5) send msg to partitions
- 8) write markers
- 10) success

**Topics**
- Topic 1
- Topic 2
- Topic 3
Exactly Once Semantics Diagram

![Diagram](image-url)