A Distributed Message Broker for High-Scale Data Streaming

Big Data Technologies
Hannes Bähr

Content

1. Message Brokers & Log Systems

Message Brokers Definition

Middleware that facilitates asynchronous communication between different parts of a distributed system by receiving, storing, and routing messages from producers to consumers.

Key Features

- Decouples producers and consumers, allowing them to operate independently
- Supports various messaging patterns like publish-subscribe and point-to-point
- Ensures reliable message delivery, persistence, and routing

Benefits

- Enhances scalability and fault tolerance in distributed architectures
- Enables integration of heterogeneous systems across different platforms
- Improves system resilience by handling message queuing and delivery retries

1. Message Brokers & Log Systems

Log Systems Definition

An append-only, time-ordered sequence of records used to track events or changes within a system.

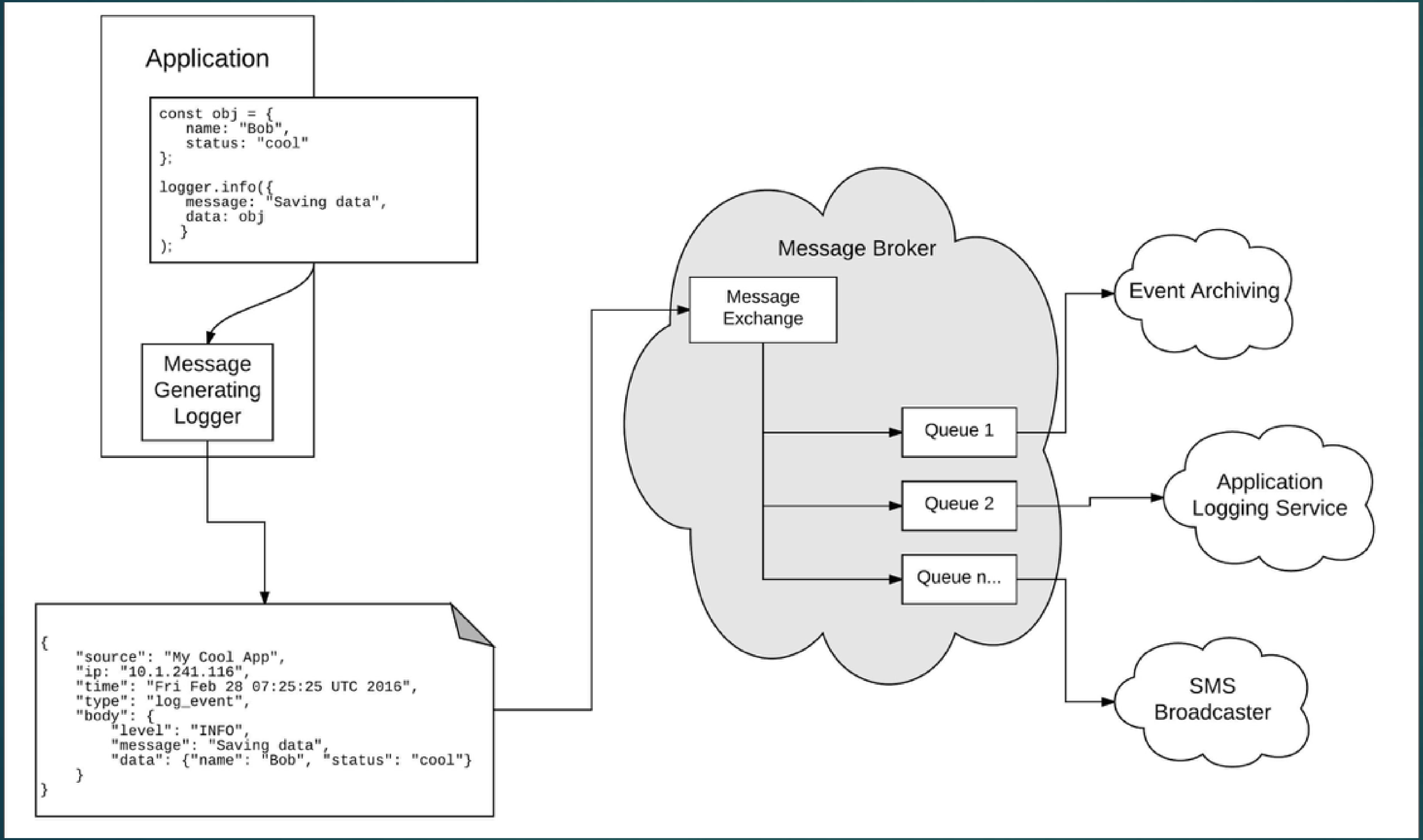
Purpose in Distributed Systems

- Provides a reliable record of events for auditing and debugging
- Facilitates data replication and synchronization across services
- Enables recovery and state reconstruction after failures

Common Implementations

- Tools like Apache Kafka use log structures to manage and store streams of records efficiently
- Log systems underpin event sourcing and stream processing architectures

1. Message Brokers & Log Systems



Log System

2. About Apache Kafka

Definition & Purpose

- Apache Kafka is an open-source distributed event streaming platform
- Used for high-performance data pipelines, streaming analytics, data integration, etc.
- Designed to handle real-time data feeds with high throughput and low latency

History

- Developed at LinkedIn to handle real-time data feeds (open-sourced in early 2011)
- Named after author Franz Kafka, because it is "a system optimized for writing"

Key Characteristics

- Kafka can handle trillions of messages per day: high-volume data processing
- Upscaling to thousands of brokers: handling petabytes of data and > 100.000 partitions
- Stores streams of records in a fault-tolerant manner, ensuring data is not lost
- Real-time processing of streaming data, enabling applications to respond to data as it occurs

3. Kafka Architecture Overview (1)

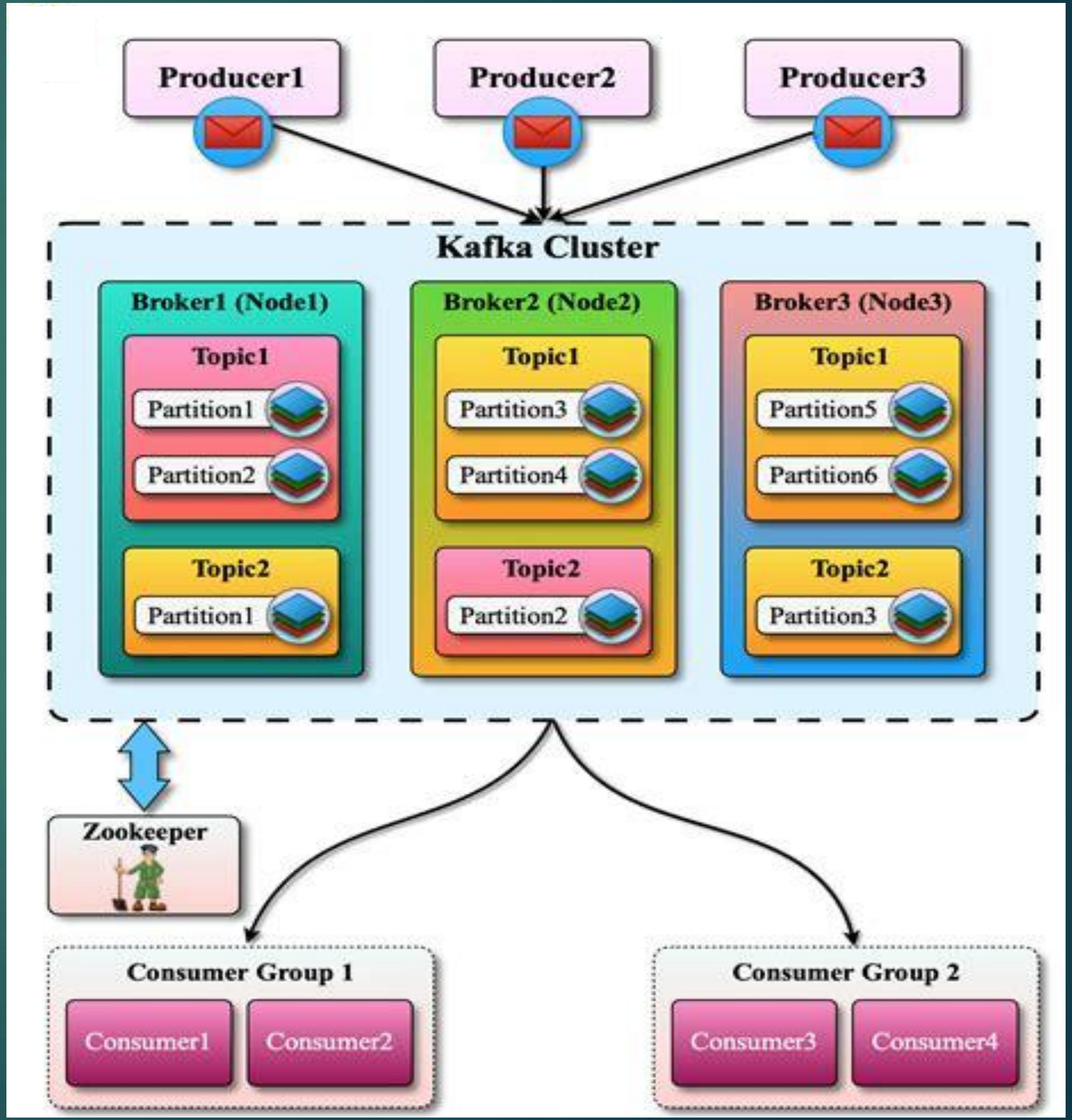
Core Components

- **Cluster**: Group of brokers working together
- **Broker**: Kafka server that stores data and serves client requests
- **Topic**: Named stream of records (like a category)
- **Partition**: Subpart of a topic for parallelism and horizontal scaling by distributing data across brokers
- **Producer**: Sends records to topics and selects specific partition for load balancing and message ordering
- **Consumer**: Reads data from topics via subscription, can be grouped into consumer groups
- **ZooKeeper**: Coordinates brokers and manages metadata

3. Kafka Architecture Overview (2)

Data Flow

1. Producers send records to a specific topic within the Kafka cluster
2. Each topic is split into partitions; producer decides to which partition record should be sent
3. Brokers store records in partitions and handle all read and write requests for them
4. Consumers subscribe to topics and read records from partitions
5. ZooKeeper manages cluster metadata and leader election for partitions



4. Kafka Topics & Partitions

Topics

About

- Named stream of records
- Functioning as a category or feed name
- Records are published by producers and from which consumers read

Characteristics

- Can have zero, one, or many consumers that subscribe to the data written to it
- Topics are split into partitions to allow for parallel processing

Partitions

About

- Subpart of a topic
- Ordered, immutable sequence of records continually appended to a commit log

Purpose

- Scalability: Distributing data across brokers
- Parallelism: Consumers can read from multiple partitions in parallel
- Fault Tolerance: Replication across brokers

Ordering

- Order of records guaranteed within partition, but not across partitions in the same topic
- Each record in a partition has a unique offset, which is a sequential unique identifier

5. Kafka Producers & Consumers

Definition

- Applications that send records to Kafka topics using the Producer API

Partitioning Logic

- Producers can specify a partition directly
- With key: Kafka hashes key to choose a partition | Without key: Round-robin distribution

Features

- Supports batching for performance
- Enables compression (gzip, snappy, lz4, zstd)
- Requires serialization of data into byte format

Delivery Guarantees

- At most once: no retries
- At least once: retries enabled
- Exactly once: via idempotence and transactions.

5. Kafka Producers & Consumers

Definition

- Applications that subscribe to Kafka topics and read records using the Consumer API

Offset Management

- Kafka assigns each message an offset in a partition
- Consumers use offsets to track progress

Poll & Rebalance

- Consumers poll for data instead of receiving it push-style
- Kafka automatically rebalances partition assignments when consumers join/leave a group

Parallelism & Grouping

- Consumers can join a consumer group to share load
- Each partition is read by only one consumer per group
- Enables scalable and fault-tolerant consumption

6. Client Libraries

Purpose

- Enable applications to interact with Kafka clusters

Main Kafka APIs

- Producer API: Send data to topics
- Consumer API: Read data from topics
- Admin API: Manage Kafka objects
- Streams API: Build stream processing apps (Java only)

Official Clients (Confluent-supported)

- Java: Most complete, includes Streams & Connect
- Python: confluent-kafka-python, kafka-python
- Go, C/C++, .NET: Bindings via librdkafka
- REST Proxy: HTTP access to Kafka

Community Clients

- Node.js: KafkaJS, kafka-node
- Rust: rust-rdkafka
- Scala: Java client + Akka Streams

7. Use Cases

Real-Time Analytics & Stream Processing

- Enable applications to process and analyze streaming data in real-time
- Examples: Financial market data processing, real-time fraud detection

Log Aggregation

- Centralize logs from various systems for monitoring and analysis
- Examples: Collecting logs from microservices for centralized monitoring

Event Sourcing

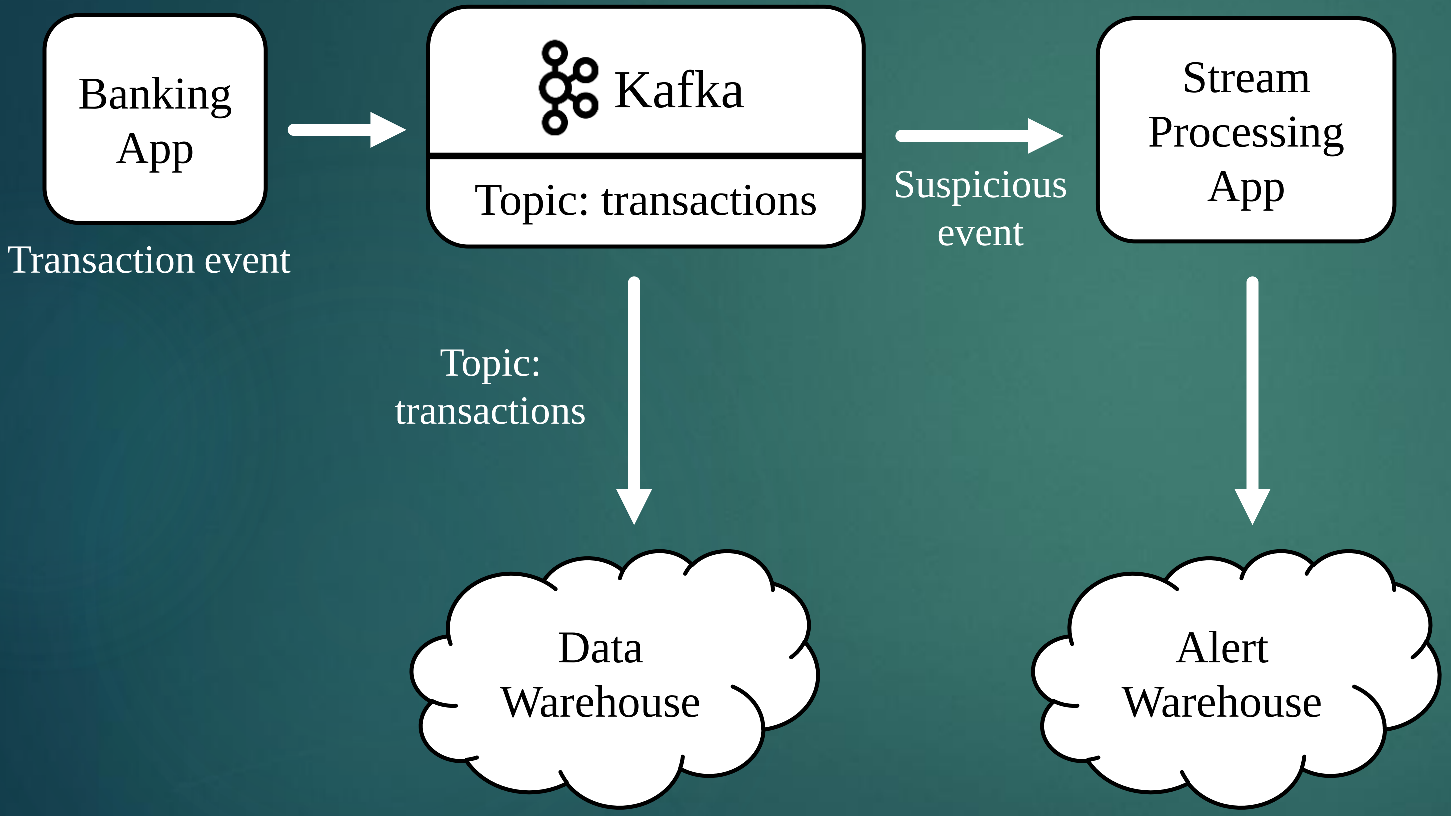
- Capture and store all changes as a sequence of events
- Examples: Maintaining a history of changes in e-commerce orders

Metrics Collection and Monitoring

- Aggregate metrics from distributed applications
- Examples: Monitoring application performance across different services

7. Use Cases

Example: Real-Time Fraud Detection in Banking



1. Banking App sends transaction events (producer).
2. Kafka receives events on “transactions” topic.
3. Stream Processing App consumes from topic, applies fraud detection logic, and writes suspicious events to another topic.
4. Alert Warehouse consumes from “suspicious-transactions” topic: sends alert/flags account.
5. Data Warehouse consumes and stores all transaction events.

8. Technical Setup (1)

Installation

- Requires Java 8 or higher
- Downloadable from the official Kafka website
- Can run in KRaft mode (no ZooKeeper) or ZooKeeper mode (for older versions)
- Start the broker after formatting or configuring the environment

Configuration

- Key broker settings: broker.id, log.dirs, listeners, num.partitions
- Control topic creation with auto.create.topics.enable
- Producers: configure acknowledgments, compression, batch size
- Consumers: configure group ID, offset reset behavior, auto-commit
- Full settings reference available from Confluent documentation

8. Technical Setup (2)

Programming

- Kafka supports Java, Python, Go, .NET, and others
- Common steps: create topics, run producers, and start consumers
- Use Kafka Streams for data processing within Kafka
- Use Kafka Connect to integrate with external systems

Thanks for your attention!