Simplified Kafka

Richard Zhao

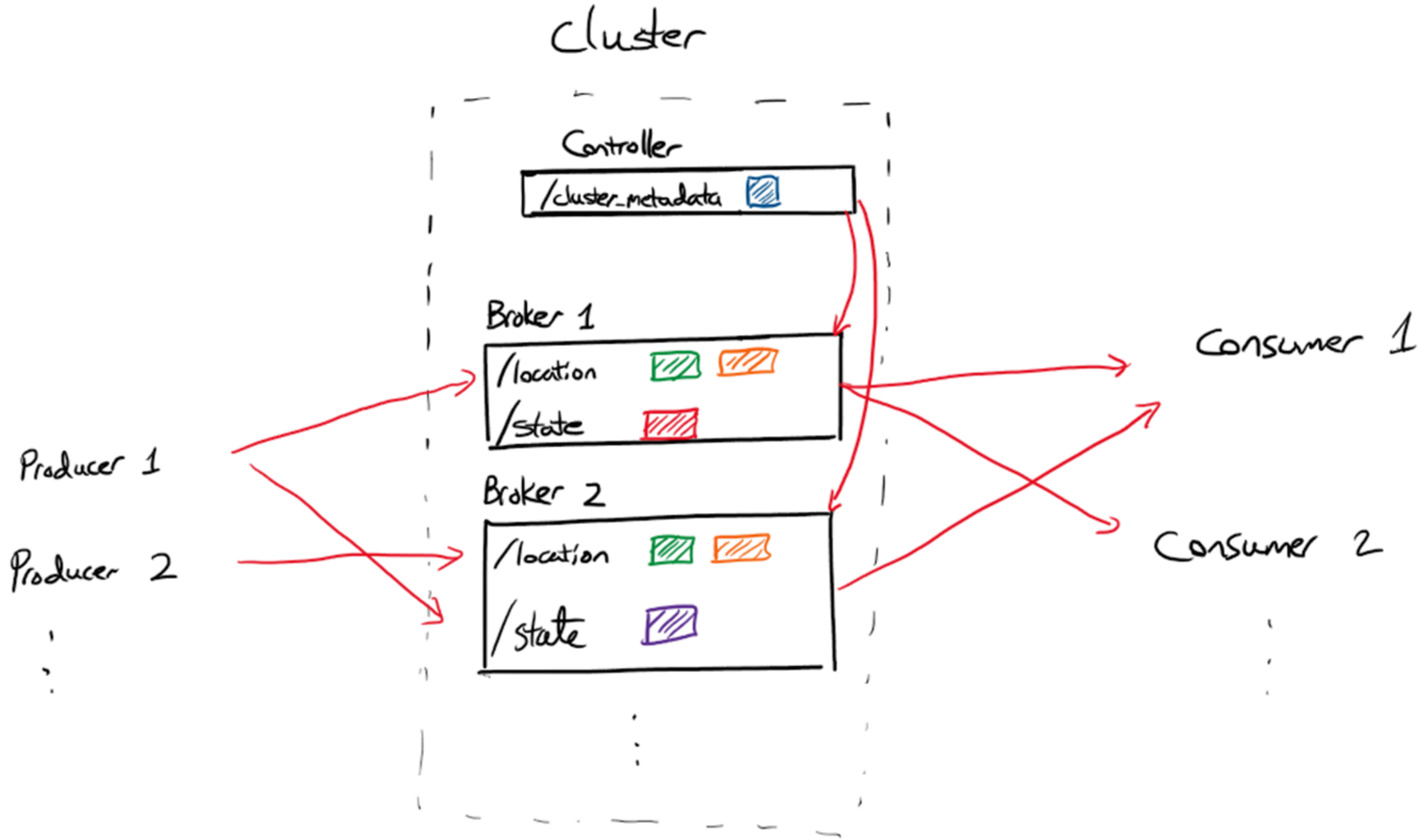
Goals

- Implementing a simplified version of Kafka as a distributed message streaming system:
  - **Producers** send records (messages) to a **topic**
  - **Consumers** subscribe to a **topic**
- Achieve availability and durability through replications and failure handling.
- Evaluate changes in system's throughput by varying number of partitions and number of brokers.

Design: Components

There are 4 components:

- **Producer** and **Consumers**(Clients):
  - Producer send records to specified topics.
  - Consumer subscribe to topics and fetch records from them.
- **Brokers** and **Controller** (Cluster):
  - Controller: Master node that stores and manages cluster metadata.
  - Broker: Stores and replicates topic records in files, handle producer and consumer requests, fetch metadata from controller.

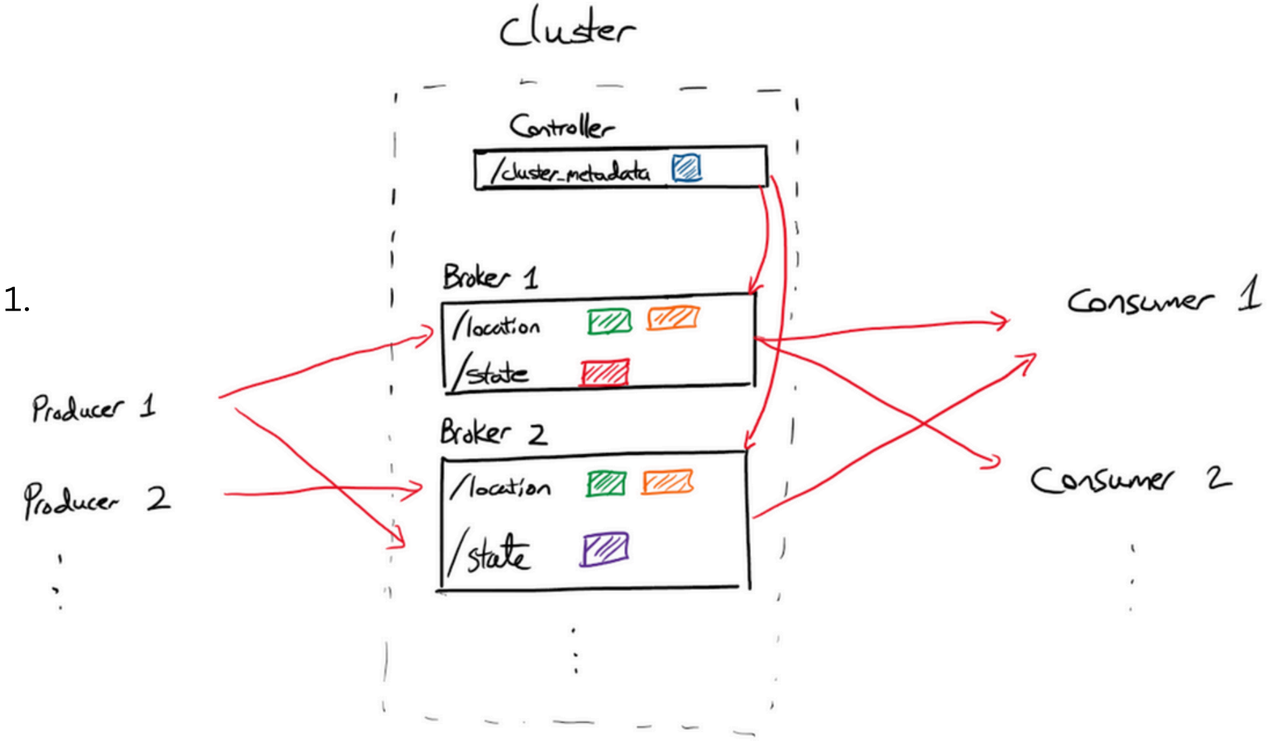


Design: Records

- A **record** (message) is a key-value pair.
- A **topic** is split/sharded into fixed number of **partitions**. Each **topic-partition** represents a single stream of records.
  - To achieve durability, each topic-partition is stored as an **append-only log**.
  - To achieve replication, each topic-partition log is replicated with a **primary-backup (leader-follower) protocol** across different brokers: One act as leader, the rest act as followers.

E.g., There are three topics:

- Topic “location” with 2 partitions and replication factor of 2.
- Topic “State” with 2 partitions and replication factor of 1
- Topic “cluster_metadata” with 1 partition and replication factor of 1.



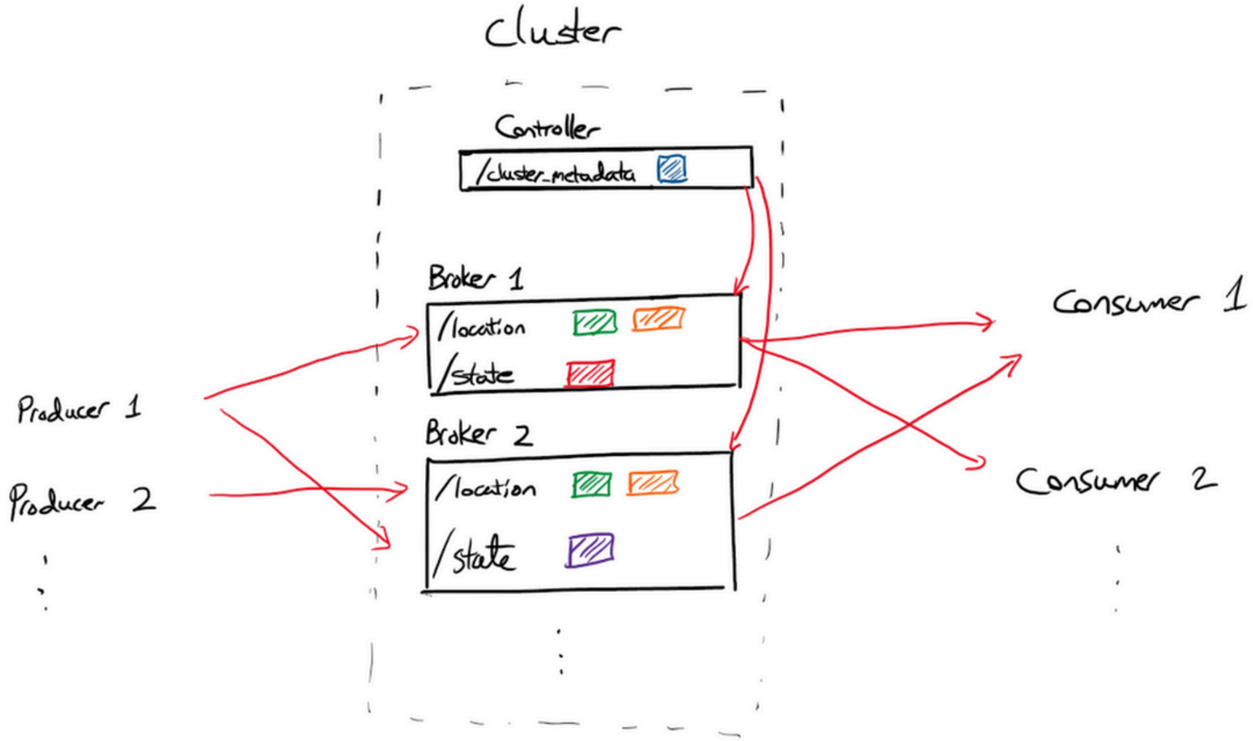
Design: Cluster Metadata

The cluster metadata includes:

- Connection information (IP, Port) of all brokers and controller in the cluster.
- The set of live brokers:
- Topic-partition assignment of brokers:
    - For example, an assignment can be (topic, partition, broker leader id, [broker follower id])

The cluster metadata is treated as any topic /cluster_metadata. Any changes to the metadata is appended as a record to its log by the controller.

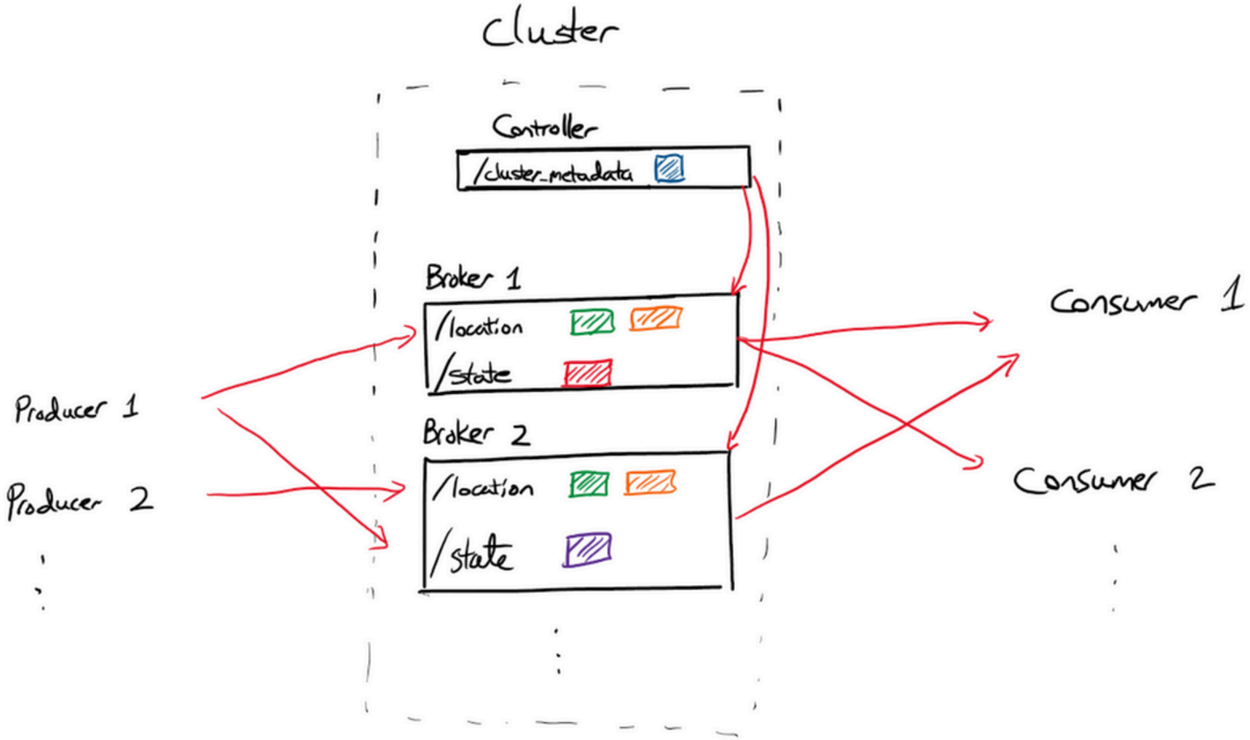
Brokers periodically send fetch requests to the controller to update their cached metadata.



Design: Controller

The Controller acts purely as a server.

- Once all brokers has connected, it performs the initial topic-partition assignment (there is a topic JSON file).
- Handle broker fetch requests for metadata.
- Monitors liveness of brokers by tracking their last fetch request time
- Performs topic-partition reassignment when a broker fails: promotes first follower if leader fails, drops from follower list otherwise.



Assumptions:

- Controller is implemented as a single node assuming it never fails.

Design: Producer

Initialization

- Bootstrap with any broker's IP/port to fetch initial cluster metadata via ClusterMetadataRequest

API

- Single-threaded with synchronous send operations:
  - send(topic, key, value)
  - send_batch(topic, key, vector<values>)

Send Process

1. Hash key → determine target partition
2. Lookup metadata → find broker leader for topic-partition
3. Connect & send ProduceRequest to leader (contains topic, partition, record batch)

Design: Consumer

Initialization

1. Get metadata: fetchClusterMetadata() or setClusterMetadata()
2. Subscribe: subscribe(topic, partition, leader_only) → randomly selects a healthy replica for load balancing
3. Poll: poll(n) returns record batches round-robin from subscriptions

Fetch Details

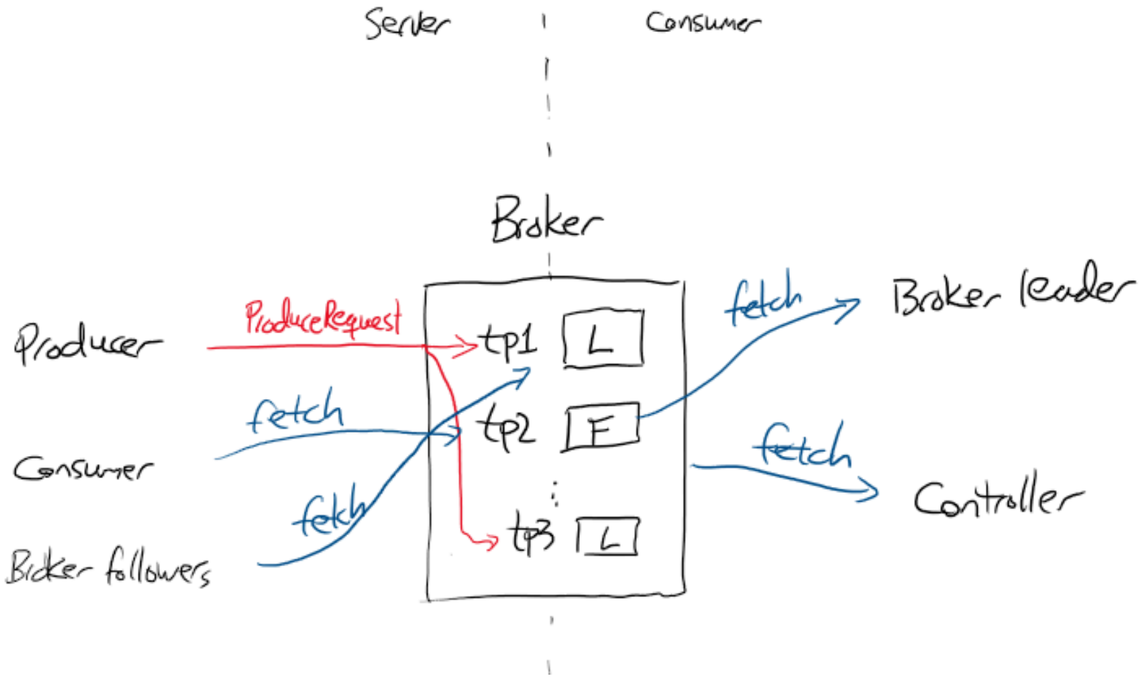
- Records identified by logical offset (monotonically increasing ID)
- FetchRequest specifies topic, partition, **consumer offset** (0 = start, -1 = latest). Consumer internally tracks its offset for the next fetch.

Design: Broker

Broker is both a server and a client/consumer:

- **Server**. Handles 3 request types:
  - ProduceRequest (from producers): Append record batch to topic-partition log
  - FetchRequest (from consumers/follower brokers):
  - ClusterMetadataRequest (from producers/consumers): Return cached cluster metadata object

- **Client**: Uses Consumer instances for 2 fetching tasks:
  - Metadata fetching (1 Consumer instance):
  - Replication (multiple Consumer instances):
    - Follower partitions fetch from their broker leaders



Design: Replication Protocol and Consistency

Primary-backup protocol (pull based):

- Each topic-partition log holds a **commited offset** and **last written offset.**
- A broker acting as leader keep track of its followers last written offset.
  - When ProduceRequest arrives, it appends and update last written offset.
  - When FetchRequest from a follower arrives, it knows what is its last written offset by the fetch offset in the request.
  - When all followers's last written offset > commit offset, update confidently the commit offset.
- A broker acting as follower:
  - FetchResponse contains the commit offset.
  - It simply updates its commit offset by copying this leader's commit offset.

Consistency:

Consumers can fetch from any replica (leader or follower), up to the commited offset.

Implementation Assumptions

- No message corruption on disk or network.
- No network partition.
- Fail-stop failure for broker, doesn’t come back up
- No dynamic addition of brokers, only failure.
- No dynamic addition of topics.
- Single controller node, assume to never fail.

Project Outcomes

✅ Achievements

- Core functionality works at small scale with multiple producers, consumers, and brokers, although only under graceful shutdown procedures.
- Follower failure tolerance verified: consumers successfully fetch from alternate replicas when followers fail

❌ Limitations & Failures

Critical Bugs (unknown root cause):

- Segmentation faults in Broker/Consumer code for non-graceful shutdowns
- Leader failure crashes entire cluster → cannot evaluate leader failure scenarios

Missing Implementations:

- Log reconciliation not implemented (required for leader failure recovery)
- No throughput evaluation (blocked by crash bugs)

Evaluation (Demo Usage)

- Basic functionalities.
- Follower failure tolerance: If a topic-partition is replicated R times, it can tolerate R-1 follower failures.