Kueue: An Implementation of Distributed Message Brokering Service

Pengyu Wang
Yale University

Yizheng Shi
Yale University

Evan Shi
Yale University

Abstract

Message queues (MQs) are foundational components in modern distributed systems, enabling decoupled and scalable communication between producers and consumers. In this document, we describe the technical design of a lightweight, distributed message queue system inspired by Apache Kafka. The system features robust persistence, distributed fault-tolerance, partition-based data sharding, and leader-based replication.

1 Introduction

In large-scale distributed systems, **message queues (MQs)** facilitate asynchronous communication among microservices. They enable decoupling of producers and consumers, allowing each side to scale or fail independently while preserving reliable data delivery [3]. Systems like Apache Kafka [3] and RabbitMQ [6] provide battle-tested solutions in production environments, employing partition-based sharding and replication mechanisms.

However, designing one's own lightweight MQ for local deployments or controlled containerized environments can be beneficial for educational purposes or specialized use cases, e.g., local testing or partial fault-tolerance scenarios. This project explores the design of such a simplified, *lightweight* message queue system, built with gRPC-based RPCs. Our implementation emphasizes partition-based architecture, leader-follower replication, configurable consistency semantics, and persistent on-disk storage to withstand node restarts or crashes.

Background and Related Work

Distributed message brokering has a long history in both industrial systems and academic research. Early distributed logging frameworks like Scribe at Facebook and Chukwa at Yahoo (later influencing Apache Flume) laid the groundwork for streaming ingestion pipelines. Apache Kafka [3] introduced a high-throughput, distributed commit log capable of partitioned, fault-tolerant message storage. Kafka's design combined the concepts of publish-subscribe messaging with partitioned, persistent logs, forming the backbone for many modern data pipelines.

From a broader distributed systems perspective, fault tolerance and replication can be achieved using consensus protocols such as Paxos [4] or Raft [5]. These protocols ensure strong consistency among replicas in the presence of node crashes or network partitions. Meanwhile, cluster coordination services like ZooKeeper [2] have been widely adopted to manage metadata and leader election in distributed systems. In our project, we take inspiration from Kafka's replication model (leader-based, asynchronous replication) but have not yet integrated a fully-fledged consensus-based controller. Instead, the Controller in our system coordinates partitions and replicas, while the actual node failover logic remains more ad-hoc. Future work could incorporate standard consensus implementations (e.g., Raft) to make the system more resilient and consistent.

2 Architecture and Design

Figure 1 depicts the *high-level architecture* of the system. It shows how Producers, Brokers, Consumers, and the Controller interact. The Controller (or broker acting as a control plane) tracks topic information, partition metadata, and broker health. Multiple Brokers manage message storage, replication, and client I/O (production and consumption). Producers and Consumers are gRPC clients that contact these Brokers based on metadata from the Controller.

2.1 Key Components

- **Producer Client API (gRPC):** Handles partition assignment, message batching, and metadata discovery.

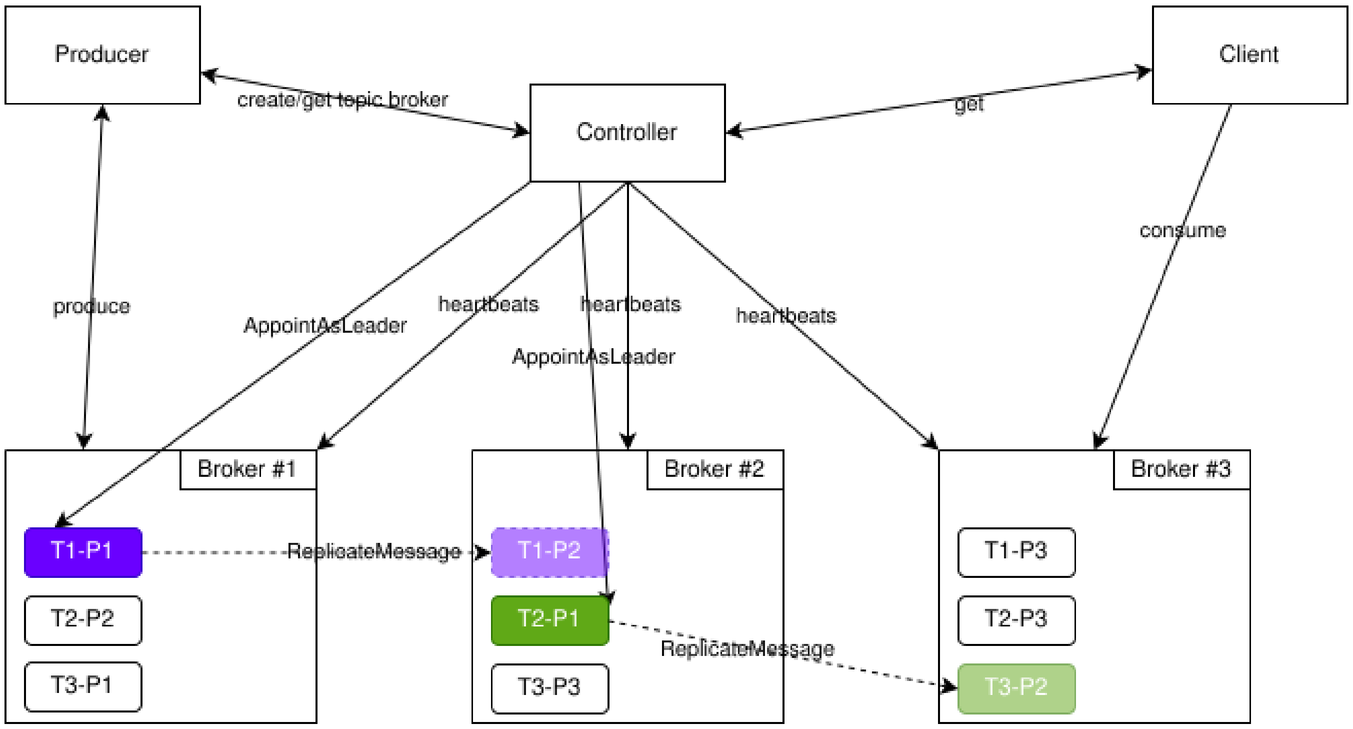

Figure 1: Overall architecture of our distributed message queue system, showing key components (Controller, Brokers, Producer, and Consumer).

- **Broker Server (gRPC):** Persists and replicates messages; ensures total order within each partition.
- **Control Plane (Metadata Broker):** Central coordinator for topic-partition-node mapping and node health monitoring.
- **Consumer Client API (gRPC):** Fetches messages, manages offsets, and supports dynamic scaling.

Producers connect to the Controller to discover the correct Broker leader for each partition. They can employ round-robin or key-based partitioning. Brokers store the messages and optionally replicate them to follower brokers, based on leader-follower protocols [3]. Consumers subscribe to topics and fetch messages from the Brokers, maintaining offsets to track their reading progress. The Controller orchestrates metadata updates, topic creation, and health checks (heartbeats).

2.2 Topic and Partition Semantics

Topics are logical groups of messages. Each topic is split into one or more *partitions*, each having a single leader broker. The leader broker receives and orders new messages and also serves consumer reads. Replicas (follower brokers) asynchronously replicate the partition's log to support failover. This architecture decouples horizontal scalability (by partitioning across multiple brokers) from concurrency (via partition-based ordering). The use of partitions for horizontal scaling is a well-known design choice in distributed log-based systems [1, 3].

2.3 Replication and Fault Tolerance

Replication is driven by a configurable replication factor. Followers periodically fetch new messages from the leader. If the leader fails, the Controller promotes a follower to the new leader. Our system does not implement full consensus protocols like Raft [5] or rely on external coordination services like ZooKeeper [2], but leader-follower replication provides partial fault tolerance. Once promoted, the new leader accepts reads and writes while re-replicating data to new or rejoining brokers.

3 Implementation Details

The system is primarily written in Go, with each component defined via dedicated gRPC services and client interfaces. The code references below illustrate the internal structures, concurrency abstractions, and persistent storage mechanisms.

3.1 Broker: On-Disk Persistence and Concurrency

Each broker runs a gRPC service (`BrokerServiceServer`) that exposes methods for producing, consuming, and replicating messages. Internally, the broker uses a sharded concurrent map structure (`ConcurrentMap`) to store messages in memory. Each key is a *topic-partition identifier* of the form `"topicName-partitionID"`. The value is a slice of `ConsumerMessage` protobuf objects representing the message log for that partition.

For crash resilience, the broker also employs a `Persister` object, which handles file-based persistence. Each partition directory on disk is subdivided into one or more `.bin` files, each file storing a batch of messages. The number of messages per file batch is configurable via `NumMessagePerBatch`. The `Persister` writes each serialized `ConsumerMessage` to disk in a size-prefixed, binary-encoded format using protobuf. When the broker starts up, it invokes `loadPersistedData` to scan partition directories and read these `.bin` files, restoring in-memory state. Similarly, consumer offsets are also persisted to `.bin` files in a dedicated `-offset` directory for each partition, and then reloaded on startup via `loadConsumerOffsets`. This design ensures that even if a broker node restarts, it can recover the partition log and individual consumer offsets from disk.

Within the broker, concurrency is managed through mutex locks at two levels:

1. A per-shard lock within each `ConcurrentMapShard` provides fine-grained concurrency for read and write operations on the in-memory message slices.
2. A broker-wide mutex (`replicaLock`) serializes updates to replication metadata (i.e., the set of follower brokers for each partition).

This combination ensures that updates to shared data structures are thread-safe while allowing parallel operations on different partitions.

3.2 Controller: Metadata Management and Heartbeats

A single `Controller` node (implemented by `ControllerServiceServer`) manages cluster-level metadata. The `Metadata` structure, defined in `metadata.go`, stores two key mappings:

- `BrokerInfos: map[string]*BrokerInfo` maps each broker ID to a `BrokerInfo` object containing the broker's address and unique name.
- `TopicInfos: map[string]*TopicInfo` holds topic-related metadata, including replication factor and partition assignments (represented by `PartitionInfo` objects that reference the leader broker and replica brokers).

When a new broker starts up, it registers itself with the controller via `RegisterBroker`. The controller locks its `Metadata` using a `sync.RWMutex` to ensure atomic updates. A heartbeat mechanism (`Heartbeat`) updates a `BrokerStatus` map to track recent liveness timestamps for each broker. If a broker fails to heartbeat, the controller eventually marks it inactive.

The controller also orchestrates topic creation with the method `GetTopic`. If the topic does not exist, the controller allocates the partitions across available brokers in a round-robin manner, sets the replication factor, and calls `AppointAsLeader` on each broker to establish the leader and follower replicas. This step uses a Go `errgroup` to send asynchronous gRPC calls to each designated partition leader. Upon success, the controller updates its in-memory `Metadata` so that subsequent producer and consumer requests can locate the correct broker leaders.

3.3 Producer Client

The producer first queries the controller for a `ProducerTopicResponse`, which lists partitions and their leader brokers. It can use round-robin or key-based hashing to choose a partition. Messages are batched in memory and sent to the leader broker via `ProduceMessage` RPC calls. For consistency levels requiring acknowledgments, the leader replicates to followers and only acknowledges once replicas confirm receipt.

```
[caption={Pseudocode for Producer Partition Selection}]
def send_message(topic, message, key=None):
    # 1. Fetch metadata from Control
    Plane partitions = control_plane.get_partitions(topic)

    # 2. Partitioning logic
    if key is not None:
        partition_id = hash(key) % len(partitions)
    else:
        partition_id = round_robin_counter %
            len(partitions)

    # 3. Batching logic
    batch_buffer.append(message)
    if len(batch_buffer) >= BATCH_SIZE:
        broker = partitions[partition_id].leader_broker
        broker.send_batch(topic, partition_id, batch_buffer)
        batch_buffer.clear()
```

Figure 2 complements the pseudocode by illustrating how a producer sends messages to a designated broker leader. First, the producer queries the controller, determines the target partition, and dispatches messages to the leader broker. The broker then persists the messages to disk and coordinates replication to followers before returning an acknowledgment to the producer.

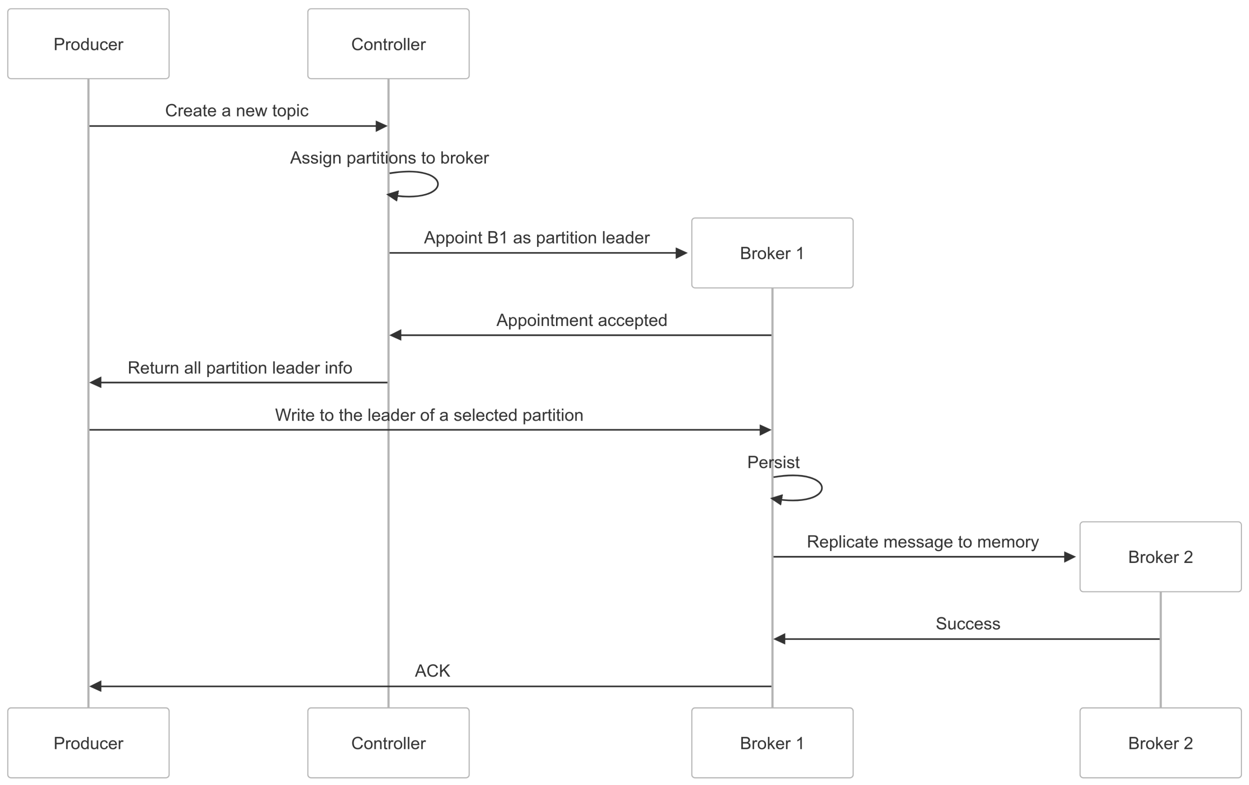
Producer
Controller
Create a new topic
Assign partitions to broker
Appoint B1 as partition leader
Broker 1
Appointment accepted
Return all partition leader info
Write to the leader of a selected partition
Persist
Replicate message to memory
Broker 2
Success
ACK
Producer
Controller
Broker 1
Broker 2

Figure 2: Illustration of the write path. The Producer determines the partition, then sends batched messages to the partition leader (Broker 1). Broker 1 persists messages on disk, replicates them to a follower (Broker 2), then acknowledges the Producer.

3.4 Consumer Client

Consumers subscribe to a topic by calling the controller's `Subscribe` method. The controller balances partitions among consumers. A consumer then fetches messages from the partition leader in bounded batches. Offsets are updated and persisted on the broker's side in `-offset` directories. On consumer or broker restart, the correct offset is restored.

```
[caption={Pseudocode for Consumer Fetch}]
def fetch_messages(topic, partition_id, offset,
     batch_size=100):
    broker = control_plane.get_leader_broker(topic,
         partition_id)
    messages = broker.read(topic, partition_id, offset,
         batch_size)
    update_local_offset(offset + len(messages))
    return messages
```

Figure 3 shows the flow of consumer reads. Once assigned a partition, the consumer periodically requests new messages from the leader broker. The broker sends messages back and persists the consumer's new offset in case of failover or restarts.

3.5 Fault Tolerance and Recovery

When a follower broker recovers after a crash, it reloads partition data from disk and also catches up from the leader by replaying the leader's partition log. If a broker's local storage is lost or corrupted, the system can be configured to reassign the partition to another broker. That broker can then load the most up-to-date log from an existing leader or from replicas. If a leader fails entirely, the controller promotes a follower to become the new leader and updates the cluster metadata. Producers and consumers subsequently direct traffic to the new leader.

4 Testing and Validation

Testing strategies consist of both unit tests and integration tests:

Unit Tests focus on verifying the concurrency map, the `Persister` logic for chunked `.bin` files, and the correctness of gRPC endpoints for Broker and Controller servers. Special attention is paid to offset handling in corner cases such as new consumers or large backlog replays.

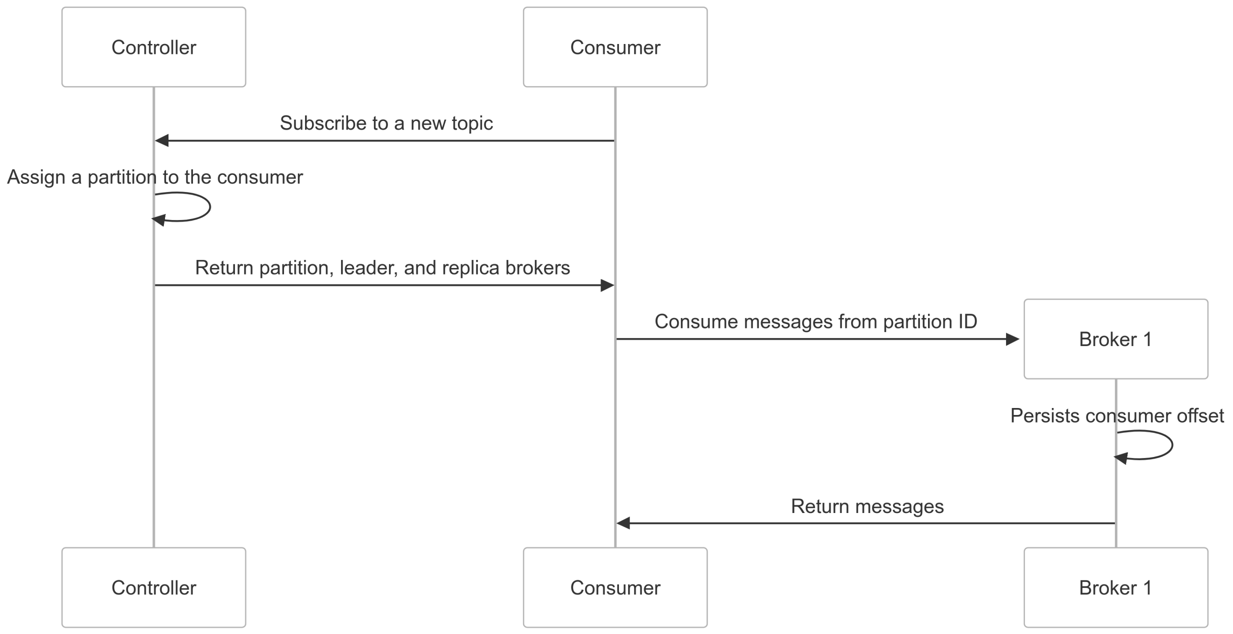

Figure 3: Illustration of the read path. The Consumer queries the controller for leader metadata, then fetches messages from the assigned partition on the leader broker. The broker returns messages and updates offset data.

Integration Tests simulate end-to-end pipelines from Producer to Broker to Consumer, testing multi-partition workloads and checking correct metadata updates when a broker restarts. Fault injection (e.g., stopping a broker during replication) validates that the controller's failover procedures and the broker's catch-up protocols are correct.

5 Deployment Plans

Local Deployment (MVP) The simplest deployment involves starting the controller, one or more brokers, and any number of producers and consumers as separate processes. Each broker internally creates its own directory for persisted message data.

Containerization and Orchestration For more advanced scenarios, Docker Compose can spin up multiple containers of brokers, producers, and consumers in a single development cluster. A potential Kubernetes-based deployment might use StatefulSets for brokers, Deployments for producers and consumers, and a Service for load balancing requests to the controller.

6 Milestones and Deliverables

Our system is built in stages, prioritizing a working MVP with basic correctness before exploring advanced features:

Stage 1 (MVP): Producer/Consumer gRPC APIs, broker partition/replication logic, file-system persistence, and unit/integration tests.

Stage 2 (Enhancements): Additional consistency mechanisms (potentially leveraging two-phase commits or Raft), dead-letter queues, delay queues, and improved observability through OpenTelemetry metrics.

7 Labor Division

Pengyu Wang contributes to the core broker server implementation, including partition management and replication logic. Potentially extends the system with more formal consistency mechanisms and designs for deployment.

Evan Shi focuses on building the Producer Client API, batch logic, partial Broker support, and integration tests. Also develops performance benchmarks to evaluate throughput and latency.

Yizheng Shi implements the Consumer Client API, offset handling, Broker support, and Docker Compose scripts for demonstration and monitoring.

8 Conclusion

This technical report outlines a lightweight distributed message queue system emphasizing partition-based data distribution and leader-follower replication, backed by file-based persistence. The design is inspired by Apache Kafka but employs simpler concurrency and metadata management strategies. Future work includes advanced replication protocols (Raft), additional data safety guarantees, and production-grade orchestration through Kubernetes.

Overall, the system demonstrates a basic yet effective approach to partitioned message queues with multi-broker replication, internal state persistence, and an extensible gRPC-based design.

References

[1] AKIDAU, T., BALIKOV, A., BEKIROĞLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., AND WHITTLE, S. Millwheel: fault-tolerant stream processing at internet scale. *Proc. VLDB Endow. 6*, 11 (Aug. 2013), 1033–1044.

[2] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)* (June 2010), USENIX Association.

[3] KREPS, J., NARKHEDE, N., AND RAO, J. Kafka: A Distributed Messaging System for Log Processing.

[4] LAMPORT, L. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58.

[5] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 305–319.

[6] VIDELA, A., AND WILLIAMS, J. J. W. *RabbitMQ in Action.* Manning Publications, 2012.