Kafka Internals: Complete Developer Guide

A comprehensive refresher on Apache Kafka internals, designed for both beginners and experienced developers. This README covers storage engines, replication mechanisms, metadata management, and consensus protocols with detailed Java examples and real-world implementations.

Table of Contents

Storage Engine

Simple Explanation

Kafka's storage engine is the foundation that manages how data is written to and read from disk. It uses an append-only log structure that is divided into segments for efficient management, indexing for fast lookups, and various retention policies to control data lifecycle.

Problem It Solves

- **Efficient disk I/O**: Sequential writes are much faster than random writes
- **Fast message retrieval**: Index files enable quick offset-based lookups
- **Storage management**: Retention policies prevent infinite disk growth
- **Operational simplicity**: Segment-based structure enables efficient cleanup and archival

Internal Architecture

```
Kafka Storage Architecture:

Topic: user-events
├── Partition 0/
│   ├── Segment 00000000000000000000.log     ← Active segment (being written)
│   ├── Segment 00000000000000000000.index   ← Offset → byte position mapping
│   ├── Segment 00000000000000000000.timeindex ← Timestamp → offset mapping
│   ├── Segment 00000000000000005000.log     ← Completed segment
│   ├── Segment 00000000000000005000.index
│   └── Segment 00000000000000005000.timeindex
├── Partition 1/
│   └── [Similar structure]
└── Partition 2/
    └── [Similar structure]

Data Flow:
Producer → Broker → Log Segment → Index Files → Consumer
```

How It Works Under the Hood

1. Log Segment Structure

Each partition is stored as a collection of segments on disk. Only one segment per partition is active (receiving new writes) at any time.

2. Segment Files

- `.log` **file**: Contains the actual message data in binary format
- `.index` **file**: Maps logical offsets to byte positions in the log file
- `.timeindex` **file**: Maps timestamps to logical offsets for time-based queries
- `.snapshot` **file**: Used for exactly-once semantics (producer idempotence)

3. Index Structure

Index files use a sparse indexing approach - not every message has an index entry. The frequency is controlled by `log.index.interval.bytes` (default: 4096 bytes).

```
Index Entry Structure:
┌──────────────────┬──────────────────┐
│ Relative Offset  │  Byte Position   │
│    (4 bytes)     │    (4 bytes)     │
└──────────────────┴──────────────────┘
```

Log Segments & Index Files

Segment Rolling Mechanism

```
import java.util.Properties;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

/**
 * Demonstrates Kafka's log segment configuration and monitoring
 */
public class KafkaSegmentManager {

    public static Properties getOptimalSegmentConfig() {
        Properties config = new Properties();

        // === Segment Size Configuration ===

        // Maximum segment size before rolling (default: 1GB)
        config.put("log.segment.bytes", "1073741824"); // 1GB

        // Time-based segment rolling (default: 7 days)
        config.put("log.roll.ms", "604800000"); // 7 days
        config.put("log.roll.hours", "168"); // Alternative time format

        // === Index Configuration ===

        // Maximum index file size (default: 10MB)
        config.put("log.index.size.max.bytes", "10485760"); // 10MB

        // Index density - bytes between index entries (default: 4KB)
        config.put("log.index.interval.bytes", "4096"); // 4KB

        // === Performance Tuning ===

        // Pre-allocate index files for better performance
        config.put("log.preallocate", "false"); // Default: false

        // Flush behavior
        config.put("log.flush.interval.messages", "9223372036854775807"); //
Long.MAX_VALUE
        config.put("log.flush.interval.ms", null); // No time-based flush

        return config;
    }

    // Example: Monitoring segment metrics
    public static void monitorSegmentMetrics() {
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

        scheduler.scheduleAtFixedRate(() -> {
            // In a real implementation, you would connect to JMX or use Kafka's
metrics
            System.out.println("=== Segment Metrics ===");

            // Key metrics to monitor:
```

```
            // - kafka.log:type=LogSize,name=Size,topic=*,partition=*
            // -
kafka.log:type=LogStartOffset,name=LogStartOffset,topic=*,partition=*
            // - kafka.log:type=LogEndOffset,name=LogEndOffset,topic=*,partition=*
            // - kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec

            logSegmentInfo();

        }, 0, 30, TimeUnit.SECONDS);
    }

    private static void logSegmentInfo() {
        // Simulate segment information
        System.out.printf("Topic: user-events, Partition: 0%n");
        System.out.printf("  Active Segment: 00000000000000125000.log (Size:
245MB)%n");
        System.out.printf("  Index Entries: 61,440 (Size: 491KB)%n");
        System.out.printf("  Segments Count: 15%n");
        System.out.printf("  Total Size: 12.8GB%n");
    }

    // Configuration for different use cases
    public static Properties getHighThroughputConfig() {
        Properties config = new Properties();

        // Larger segments for high-throughput scenarios
        config.put("log.segment.bytes", "2147483648"); // 2GB
        config.put("log.roll.ms", "86400000"); // 1 day

        // Less frequent indexing to reduce overhead
        config.put("log.index.interval.bytes", "8192"); // 8KB

        // Pre-allocate for better performance
        config.put("log.preallocate", "true");

        return config;
    }

    public static Properties getLowLatencyConfig() {
        Properties config = new Properties();

        // Smaller segments for lower latency during rolling
        config.put("log.segment.bytes", "268435456"); // 256MB
        config.put("log.roll.ms", "3600000"); // 1 hour

        // More frequent indexing for faster lookups
        config.put("log.index.interval.bytes", "2048"); // 2KB

        // Larger index files to accommodate more entries
        config.put("log.index.size.max.bytes", "20971520"); // 20MB

        return config;
    }
}
```

Index File Usage Example

```
/**
 * Demonstrates how Kafka uses index files for message retrieval
 */
public class KafkaIndexExample {

    /**
     * Simulates how Kafka finds a message by offset using index files
     */
    public static class OffsetLookup {

        // Simulated index entries (offset -> byte position)
        private static final IndexEntry[] INDEX_ENTRIES = {
            new IndexEntry(1000, 0),      // Base offset of segment
            new IndexEntry(1028, 4169),   // After ~4KB of data
            new IndexEntry(1056, 8364),   // After ~8KB of data
            new IndexEntry(1084, 12564)   // After ~12KB of data
        };

        public static class IndexEntry {
            final long offset;
            final int bytePosition;

            IndexEntry(long offset, int bytePosition) {
                this.offset = offset;
                this.bytePosition = bytePosition;
            }
        }

        /**
         * Binary search to find the appropriate index entry
         * Similar to Kafka's actual implementation
         */
        public static SearchResult findMessage(long targetOffset) {
            // Step 1: Find the correct segment based on filename
            String segmentName = findSegmentForOffset(targetOffset);

            // Step 2: Binary search in index file
            IndexEntry entry = binarySearchIndex(targetOffset);

            // Step 3: Sequential scan from index position
            return new SearchResult(segmentName, entry.bytePosition,
targetOffset);
        }

        private static String findSegmentForOffset(long offset) {
            // Segments are named with their base offset
            // e.g., 00000000000000001000.log contains offsets 1000+
            if (offset >= 1000) return "00000000000000001000.log";
```

```
            if (offset >= 500) return "00000000000000000500.log";
            return "00000000000000000000.log";
        }

        private static IndexEntry binarySearchIndex(long targetOffset) {
            int left = 0;
            int right = INDEX_ENTRIES.length - 1;
            IndexEntry result = INDEX_ENTRIES[0];

            while (left <= right) {
                int mid = left + (right - left) / 2;

                if (INDEX_ENTRIES[mid].offset <= targetOffset) {
                    result = INDEX_ENTRIES[mid];
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }

            return result;
        }

        public static class SearchResult {
            final String segmentFile;
            final int startBytePosition;
            final long targetOffset;

            SearchResult(String segmentFile, int startBytePosition, long
targetOffset) {
                this.segmentFile = segmentFile;
                this.startBytePosition = startBytePosition;
                this.targetOffset = targetOffset;
            }

            @Override
            public String toString() {
                return String.format("Segment: %s, Start at byte: %d, Target
offset: %d",
                    segmentFile, startBytePosition, targetOffset);
            }
        }

        // Example usage
        public static void main(String[] args) {
            System.out.println("=== Kafka Index Lookup Simulation ===");

            long[] testOffsets = {1001, 1030, 1055, 1090};

            for (long offset : testOffsets) {
                SearchResult result = findMessage(offset);
                System.out.println("Offset " + offset + " -> " + result);
            }
        }
```

```
    }
}
```

Retention Policies

Time-Based Retention

```
import java.time.Duration;
import java.util.concurrent.TimeUnit;

/**
 * Comprehensive guide to Kafka retention policies
 */
public class KafkaRetentionPolicies {

    /**
     * Time-based retention configuration
     */
    public static Properties getTimeBasedRetentionConfig() {
        Properties config = new Properties();

        // === Primary time-based retention ===

        // Retention in milliseconds (most precise)
        config.put("log.retention.ms", "604800000"); // 7 days

        // Retention in minutes (overridden by ms if both set)
        config.put("log.retention.minutes", "10080"); // 7 days

        // Retention in hours (overridden by ms or minutes)
        config.put("log.retention.hours", "168"); // 7 days (default)

        // === Segment deletion timing ===

        // How often to check for deletable segments (default: 5 minutes)
        config.put("log.retention.check.interval.ms", "300000");

        // Minimum age before deletion (prevents deletion of recently rolled
segments)
        config.put("log.segment.delete.delay.ms", "60000"); // 1 minute

        return config;
    }

    /**
     * Size-based retention configuration
     */
    public static Properties getSizeBasedRetentionConfig() {
        Properties config = new Properties();

        // Maximum partition size before deletion (default: -1 = unlimited)
```

```
        config.put("log.retention.bytes", "1073741824000"); // 1TB per partition

        // Cleanup policy
        config.put("log.cleanup.policy", "delete"); // vs "compact" or
"compact,delete"

        // Segment size (affects how granular deletion can be)
        config.put("log.segment.bytes", "1073741824"); // 1GB

        return config;
    }

    /**
     * Log compaction configuration - keeps only latest value per key
     */
    public static Properties getLogCompactionConfig() {
        Properties config = new Properties();

        // === Core compaction settings ===

        // Cleanup policy for compacted topics
        config.put("log.cleanup.policy", "compact");

        // Minimum ratio of dirty records to total records before compaction
        config.put("log.cleaner.min.cleanable.ratio", "0.5"); // Default: 0.5

        // === Compaction timing ===

        // How often to run the log cleaner (default: 15 seconds)
        config.put("log.cleaner.backoff.ms", "15000");

        // Minimum time before a record becomes eligible for compaction
        config.put("log.cleaner.min.compaction.lag.ms", "0"); // Default: 0

        // Maximum time before forcing compaction
        config.put("log.cleaner.max.compaction.lag.ms", "9223372036854775807"); //
Long.MAX_VALUE

        // === Compaction performance ===

        // Number of cleaner threads
        config.put("log.cleaner.threads", "1"); // Default: 1

        // Memory for cleaner deduplication buffer
        config.put("log.cleaner.dedupe.buffer.size", "134217728"); // 128MB

        // I/O buffer size for cleaner
        config.put("log.cleaner.io.buffer.size", "524288"); // 512KB

        // === Compaction constraints ===

        // Keep at least this many uncompacted segments
        config.put("log.cleaner.io.buffer.load.factor", "0.9");
```

```
        return config;
    }

    /**
     * Combined retention and compaction (useful for CDC use cases)
     */
    public static Properties getCombinedRetentionConfig() {
        Properties config = new Properties();

        // Use both compaction and deletion
        config.put("log.cleanup.policy", "compact,delete");

        // Time retention (compacted records still subject to time limits)
        config.put("log.retention.hours", "168"); // 7 days

        // Size retention
        config.put("log.retention.bytes", "1073741824000"); // 1TB

        // Compaction settings
        config.put("log.cleaner.min.cleanable.ratio", "0.1"); // More aggressive
compaction
        config.put("log.cleaner.min.compaction.lag.ms", "60000"); // 1 minute
delay

        return config;
    }

    // Example: Retention policy selection based on use case
    public static Properties getRetentionConfigForUseCase(UseCase useCase) {
        switch (useCase) {
            case EVENT_STREAMING:
                // Short retention, high throughput
                Properties eventConfig = new Properties();
                eventConfig.put("log.retention.hours", "24"); // 1 day
                eventConfig.put("log.cleanup.policy", "delete");
                eventConfig.put("log.segment.bytes", "1073741824"); // 1GB
                return eventConfig;

            case CHANGE_DATA_CAPTURE:
                // Long retention with compaction
                Properties cdcConfig = new Properties();
                cdcConfig.put("log.retention.hours", "8760"); // 1 year
                cdcConfig.put("log.cleanup.policy", "compact,delete");
                cdcConfig.put("log.cleaner.min.cleanable.ratio", "0.1");
                return cdcConfig;

            case AUDIT_LOGGING:
                // Very long retention, no compaction
                Properties auditConfig = new Properties();
                auditConfig.put("log.retention.hours", "26280"); // 3 years
                auditConfig.put("log.cleanup.policy", "delete");
                auditConfig.put("log.retention.bytes", "10737418240000L"); // 10TB
                return auditConfig;
```

```
            case SESSION_TRACKING:
                // Medium retention with compaction
                Properties sessionConfig = new Properties();
                sessionConfig.put("log.retention.hours", "720"); // 30 days
                sessionConfig.put("log.cleanup.policy", "compact");
                sessionConfig.put("log.cleaner.min.cleanable.ratio", "0.3");
                return sessionConfig;

            default:
                return getTimeBasedRetentionConfig();
        }
    }

    public enum UseCase {
        EVENT_STREAMING,
        CHANGE_DATA_CAPTURE,
        AUDIT_LOGGING,
        SESSION_TRACKING
    }
}
```

Log Compaction Implementation Example

```
/**
 * Demonstrates how log compaction works in Kafka
 */
public class LogCompactionExample {

    /**
     * Simulates log compaction process
     */
    public static class CompactionSimulator {

        public static class LogRecord {
            final String key;
            final String value;
            final long offset;
            final long timestamp;
            final boolean isDelete; // Tombstone marker

            LogRecord(String key, String value, long offset, long timestamp) {
                this(key, value, offset, timestamp, false);
            }

            LogRecord(String key, String value, long offset, long timestamp,
boolean isDelete) {
                this.key = key;
                this.value = value;
                this.offset = offset;
                this.timestamp = timestamp;
                this.isDelete = isDelete;
```

```
            }

            @Override
            public String toString() {
                return String.format("Offset:%d Key:%s Value:%s Delete:%s",
                    offset, key, value, isDelete);
            }
        }

        public static List<LogRecord> compactSegment(List<LogRecord> originalLog)
{
            System.out.println("=== Log Compaction Simulation ===");
            System.out.println("Original log:");
            originalLog.forEach(System.out::println);

            // Step 1: Build map of latest records per key
            Map<String, LogRecord> latestRecords = new LinkedHashMap<>();

            for (LogRecord record : originalLog) {
                if (record.key != null) { // Skip records without keys
                    latestRecords.put(record.key, record);
                }
            }

            // Step 2: Create compacted log maintaining offset order
            List<LogRecord> compactedLog = new ArrayList<>();
            Set<String> processedKeys = new HashSet<>();

            for (LogRecord record : originalLog) {
                if (record.key == null) {
                    // Keep records without keys as-is
                    compactedLog.add(record);
                } else if (!processedKeys.contains(record.key)) {
                    // Add the latest record for this key
                    LogRecord latestRecord = latestRecords.get(record.key);
                    if (!latestRecord.isDelete) { // Don't include tombstones in
final log
                        compactedLog.add(latestRecord);
                    }
                    processedKeys.add(record.key);
                }
            }

            System.out.println("\nCompacted log:");
            compactedLog.forEach(System.out::println);

            System.out.printf("\nCompaction results: %d -> %d records (%.1f%%
reduction)%n",
                originalLog.size(), compactedLog.size(),
                (1.0 - (double) compactedLog.size() / originalLog.size()) * 100);

            return compactedLog;
        }
```

```
        // Example usage
        public static void main(String[] args) {
            List<LogRecord> originalLog = Arrays.asList(
                new LogRecord("user1", "{'name':'John','age':25}", 1000,
System.currentTimeMillis()),
                new LogRecord("user2", "{'name':'Jane','age':30}", 1001,
System.currentTimeMillis()),
                new LogRecord("user1", "{'name':'John','age':26}", 1002,
System.currentTimeMillis()), // Update
                new LogRecord("user3", "{'name':'Bob','age':35}", 1003,
System.currentTimeMillis()),
                new LogRecord("user2", null, 1004, System.currentTimeMillis(),
true), // Tombstone
                new LogRecord("user1", "{'name':'John','age':27}", 1005,
System.currentTimeMillis()), // Final update
                new LogRecord(null, "system-event-1", 1006,
System.currentTimeMillis()) // No key
            );

            compactSegment(originalLog);
        }
    }
}
```

Tiered Storage

Tiered Storage Architecture (Kafka 3.6+)

```
/**
 * Kafka Tiered Storage configuration and management
 * Available since Kafka 3.6 (Early Access)
 */
public class KafkaTieredStorage {

    /**
     * Basic tiered storage configuration
     */
    public static Properties getTieredStorageConfig() {
        Properties config = new Properties();

        // === Enable tiered storage ===

        // Cluster-level enablement
        config.put("remote.log.storage.system.enable", "true");

        // Topic-level enablement (can be set per topic)
        config.put("remote.storage.enable", "true");

        // === Storage retention configuration ===

        // Local storage retention (hot tier)
```

```
        config.put("local.retention.ms", "86400000"); // 1 day local
        config.put("local.retention.bytes", "1073741824"); // 1GB local

        // Total retention (local + remote)
        config.put("retention.ms", "604800000"); // 7 days total
        config.put("retention.bytes", "107374182400"); // 100GB total

        // === Remote storage configuration ===

        // Remote storage implementation
        config.put("remote.log.storage.manager.class.name",
            "org.apache.kafka.server.log.remote.storage.RemoteLogStorageManager");

        // Remote log metadata manager
        config.put("remote.log.metadata.manager.class.name",

"org.apache.kafka.server.log.remote.metadata.TopicBasedRemoteLogMetadataManager");

        // Remote metadata topic configuration
        config.put("remote.log.metadata.topic.partitions", "50");
        config.put("remote.log.metadata.topic.replication.factor", "3");

        // === Performance tuning ===

        // Index cache for remote segments
        config.put("remote.log.index.file.cache.total.size.bytes", "1073741824");
// 1GB cache

        // Remote log manager threads
        config.put("remote.log.manager.thread.pool.size", "10");

        // Copy/delete lag monitoring
        config.put("remote.log.manager.copy.max.bytes.per.second", "104857600");
// 100MB/s

        return config;
    }

    /**
     * S3-based tiered storage configuration
     */
    public static Properties getS3TieredStorageConfig() {
        Properties config = getTieredStorageConfig();

        // S3-specific remote storage manager
        config.put("remote.log.storage.manager.class.name",
            "io.confluent.connect.s3.storage.S3RemoteStorageManager");

        // S3 configuration
        config.put("remote.log.storage.manager.class.path",
"/usr/share/kafka/plugins/");
        config.put("remote.log.storage.manager.props",
            "s3.bucket.name=kafka-tiered-storage," +
            "s3.region=us-west-2," +
```

```
                "aws.access.key.id=${env:AWS_ACCESS_KEY_ID}," +
                "aws.secret.access.key=${env:AWS_SECRET_ACCESS_KEY}");

            return config;
        }

        /**
         * Monitoring tiered storage metrics
         */
        public static class TieredStorageMonitor {

            public static void monitorTieredStorageMetrics() {
                System.out.println("=== Tiered Storage Metrics ===");

                // Key metrics to monitor (would be retrieved from JMX in real
    implementation):

                // Remote copy lag (segments)
                System.out.println("RemoteCopyLagSegments: 2 segments");

                // Remote copy lag (bytes)
                System.out.println("RemoteCopyLagBytes: 512MB");

                // Remote delete lag
                System.out.println("RemoteDeleteLagSegments: 0 segments");

                // Upload/download rates
                System.out.println("RemoteUploadRate: 85MB/s");
                System.out.println("RemoteDownloadRate: 120MB/s");

                // Local vs remote storage distribution
                System.out.println("LocalStorageBytes: 2.1GB");
                System.out.println("RemoteStorageBytes: 45.8GB");
                System.out.println("LocalStorageRatio: 4.4%");

                // Cache statistics
                System.out.println("RemoteIndexCacheSize: 256MB");
                System.out.println("RemoteIndexCacheHitRate: 89.2%");
            }

            // Simulate tiered storage decision making
            public static boolean shouldTierSegment(SegmentInfo segment) {
                long localRetentionMs = 86400000; // 1 day
                long segmentAge = System.currentTimeMillis() - segment.lastModified;

                boolean isOldEnough = segmentAge > localRetentionMs;
                boolean isInactive = !segment.isActive;
                boolean isLeaderSegment = segment.isLeader;

                return isOldEnough && isInactive && isLeaderSegment;
            }

            public static class SegmentInfo {
                final String segmentName;
```

```
            final long lastModified;
            final boolean isActive;
            final boolean isLeader;
            final long sizeBytes;

            SegmentInfo(String segmentName, long lastModified, boolean isActive,
                        boolean isLeader, long sizeBytes) {
                this.segmentName = segmentName;
                this.lastModified = lastModified;
                this.isActive = isActive;
                this.isLeader = isLeader;
                this.sizeBytes = sizeBytes;
            }
        }
    }

    /**
     * Example: Configuring tiered storage for different use cases
     */
    public static Properties getTieredConfigForUseCase(TieredUseCase useCase) {
        Properties config = getTieredStorageConfig();

        switch (useCase) {
            case ANALYTICS_WORKLOAD:
                // Keep more data locally for recent analytics
                config.put("local.retention.ms", "604800000"); // 7 days local
                config.put("retention.ms", "31536000000"); // 1 year total
                break;

            case LOG_AGGREGATION:
                // Quick tiering for log data
                config.put("local.retention.ms", "3600000"); // 1 hour local
                config.put("retention.ms", "2592000000"); // 30 days total
                break;

            case COMPLIANCE_ARCHIVE:
                // Long-term storage for compliance
                config.put("local.retention.ms", "86400000"); // 1 day local
                config.put("retention.ms", "94608000000"); // 3 years total
                break;

            case IOT_TELEMETRY:
                // Balance between cost and access speed
                config.put("local.retention.ms", "259200000"); // 3 days local
                config.put("retention.ms", "7776000000"); // 90 days total
                break;
        }

        return config;
    }

    public enum TieredUseCase {
        ANALYTICS_WORKLOAD,
        LOG_AGGREGATION,
```

```
        COMPLIANCE_ARCHIVE,
        IOT_TELEMETRY
    }
}
```

Replication

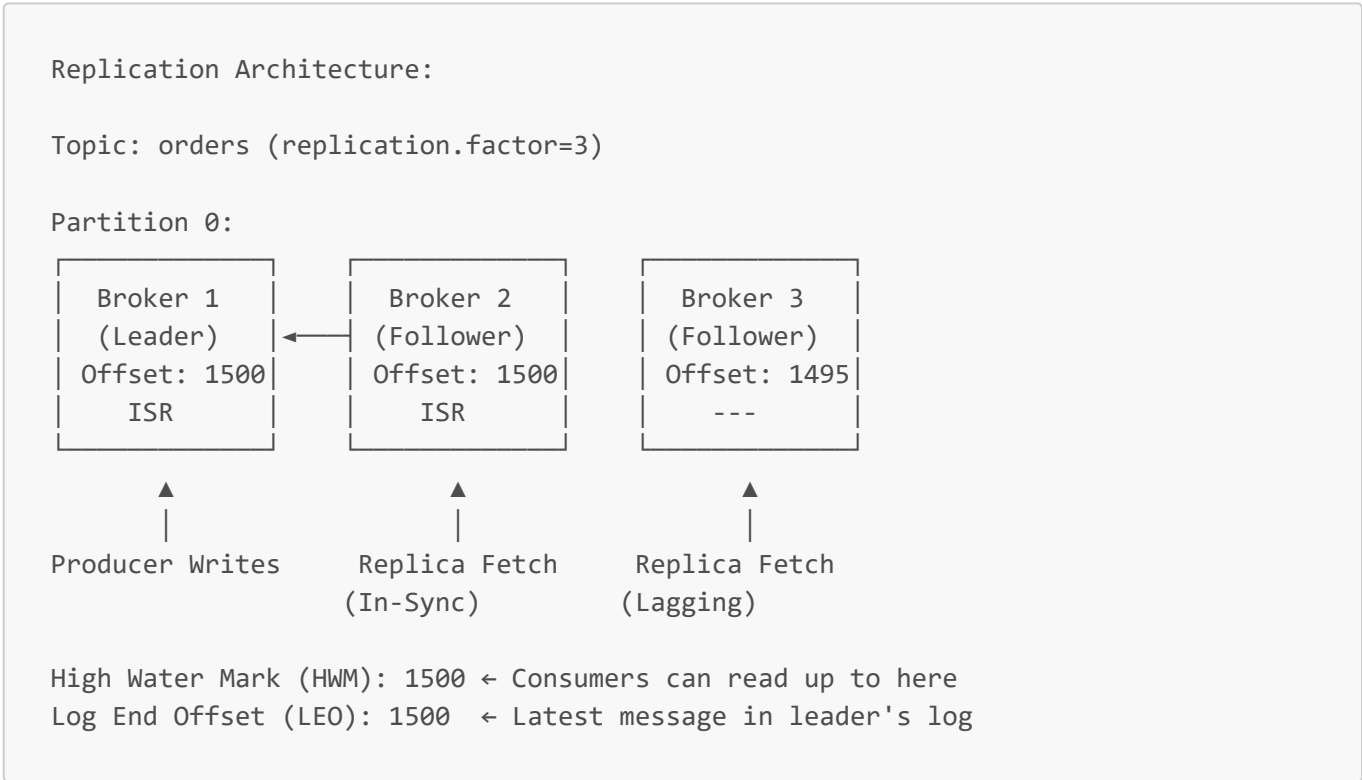
Simple Explanation

Kafka replication ensures data durability and availability by maintaining multiple copies of each partition across different brokers. The replication mechanism uses a leader-follower model where one replica handles all reads and writes while followers synchronize with the leader.

Problem It Solves

- **Fault tolerance**: Data survives broker failures
- **High availability**: Service continues even with node failures
- **Data consistency**: All replicas eventually have the same data
- **Load distribution**: Followers can serve read requests in some configurations

Internal Architecture



In-Sync Replicas (ISR)

ISR Management Implementation

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ScheduledExecutorService;
```

```
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

/**
 * Demonstrates ISR (In-Sync Replicas) management in Kafka
 */
public class ISRManager {

    /**
     * Core ISR configuration parameters
     */
    public static Properties getISRConfig() {
        Properties config = new Properties();

        // === ISR Management ===

        // Maximum time a follower can lag before being removed from ISR
        config.put("replica.lag.time.max.ms", "30000"); // 30 seconds (default:
30s)

        // Socket timeout for replica fetcher
        config.put("replica.socket.timeout.ms", "30000"); // 30 seconds

        // Receive buffer for replica fetcher
        config.put("replica.socket.receive.buffer.bytes", "65536"); // 64KB

        // === Fetch Configuration ===

        // Maximum wait time for fetch requests
        config.put("replica.fetch.wait.max.ms", "500"); // 500ms

        // Minimum bytes to fetch (helps with batching)
        config.put("replica.fetch.min.bytes", "1"); // 1 byte (default)

        // Maximum bytes per fetch request
        config.put("replica.fetch.max.bytes", "1048576"); // 1MB

        // Fetch response size for replica fetcher
        config.put("replica.fetch.response.max.bytes", "10485760"); // 10MB

        // === Thread Configuration ===

        // Number of replica fetcher threads per source broker
        config.put("num.replica.fetchers", "1"); // Default: 1

        // === High Water Mark ===

        // Frequency of high water mark checkpoint
        config.put("replica.high.watermark.checkpoint.interval.ms", "5000"); // 5
seconds

        return config;
    }
```

```
    /**
     * Simulates ISR management logic
     */
    public static class ISRSimulator {

        private final Map<String, ReplicaInfo> replicas = new ConcurrentHashMap<>
();
        private final ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(2);
        private final long maxLagTimeMs;
        private volatile Set<String> currentISR = new HashSet<>();
        private volatile long leaderHWM = 0;

        public ISRSimulator(long maxLagTimeMs) {
            this.maxLagTimeMs = maxLagTimeMs;
            this.currentISR.add("broker-1-leader"); // Leader is always in ISR
        }

        public void start() {
            // Monitor replica lag
            scheduler.scheduleAtFixedRate(this::checkISRStatus, 0, 5,
TimeUnit.SECONDS);

            // Simulate replica fetch requests
            scheduler.scheduleAtFixedRate(this::simulateReplicaFetch, 0, 1,
TimeUnit.SECONDS);
        }

        public void addReplica(String brokerId, boolean isLeader) {
            replicas.put(brokerId, new ReplicaInfo(brokerId, isLeader));
            if (isLeader) {
                currentISR.add(brokerId);
                System.out.println("Added leader replica: " + brokerId);
            } else {
                System.out.println("Added follower replica: " + brokerId);
            }
        }

        private void checkISRStatus() {
            System.out.println("\n=== ISR Status Check ===");
            long currentTime = System.currentTimeMillis();
            Set<String> newISR = new HashSet<>();

            for (ReplicaInfo replica : replicas.values()) {
                if (replica.isLeader) {
                    newISR.add(replica.brokerId);
                    continue;
                }

                long lagTime = currentTime - replica.lastFetchTime;
                long offsetLag = leaderHWM - replica.lastFetchedOffset;

                boolean isInSync = lagTime <= maxLagTimeMs && offsetLag <= 1000;
// Allow small offset lag
```

```
                if (isInSync) {
                    newISR.add(replica.brokerId);
                    System.out.printf("Replica %s: IN-SYNC (lag: %dms, offset_lag:
%d)%n",
                        replica.brokerId, lagTime, offsetLag);
                } else {
                    System.out.printf("Replica %s: OUT-OF-SYNC (lag: %dms,
offset_lag: %d)%n",
                        replica.brokerId, lagTime, offsetLag);
                }
            }

            if (!newISR.equals(currentISR)) {
                System.out.println("ISR changed: " + currentISR + " -> " +
newISR);
                currentISR = newISR;
                updateHighWaterMark();
            }

            System.out.println("Current ISR: " + currentISR);
            System.out.println("High Water Mark: " + leaderHWM);
        }

        private void simulateReplicaFetch() {
            // Simulate leader receiving new messages
            leaderHWM += 100; // 100 new messages per second

            // Update leader replica
            ReplicaInfo leader = replicas.values().stream()
                .filter(r -> r.isLeader)
                .findFirst().orElse(null);

            if (leader != null) {
                leader.lastFetchedOffset = leaderHWM;
                leader.lastFetchTime = System.currentTimeMillis();
            }

            // Simulate follower fetch requests with varying lag
            for (ReplicaInfo replica : replicas.values()) {
                if (!replica.isLeader) {
                    // Simulate network delay and processing time
                    long randomDelay = (long) (Math.random() * 2000); // 0-2s
delay

                    if (Math.random() > 0.1) { // 90% success rate
                        replica.lastFetchedOffset = Math.min(leaderHWM - 50,
replica.lastFetchedOffset + 80);
                        replica.lastFetchTime = System.currentTimeMillis() -
randomDelay;
                    }
                    // 10% of the time, replica fails to fetch (simulating network
issues)
                }
```

```
            }
        }

        private void updateHighWaterMark() {
            // High water mark is the minimum offset among all ISR replicas
            long minOffset = currentISR.stream()
                .mapToLong(brokerId -> replicas.get(brokerId).lastFetchedOffset)
                .min()
                .orElse(0);

            if (minOffset < leaderHWM) {
                System.out.println("High Water Mark updated: " + leaderHWM + " ->
" + minOffset);
                leaderHWM = minOffset;
            }
        }

        public void stop() {
            scheduler.shutdown();
        }

        private static class ReplicaInfo {
            final String brokerId;
            final boolean isLeader;
            volatile long lastFetchedOffset = 0;
            volatile long lastFetchTime = System.currentTimeMillis();

            ReplicaInfo(String brokerId, boolean isLeader) {
                this.brokerId = brokerId;
                this.isLeader = isLeader;
            }
        }

        // Example usage
        public static void main(String[] args) throws InterruptedException {
            ISRSimulator simulator = new ISRSimulator(30000); // 30 second max lag

            // Add replicas
            simulator.addReplica("broker-1-leader", true);
            simulator.addReplica("broker-2-follower", false);
            simulator.addReplica("broker-3-follower", false);

            simulator.start();

            // Run simulation for 30 seconds
            Thread.sleep(30000);

            simulator.stop();
        }
    }
}
```

Replica Fetcher Threads

Replica Fetcher Implementation

```
/**
 * Demonstrates replica fetcher thread mechanism
 */
public class ReplicaFetcherExample {

    /**
     * Replica fetcher configuration
     */
    public static Properties getReplicaFetcherConfig() {
        Properties config = new Properties();

        // === Fetcher Thread Configuration ===

        // Number of fetcher threads per source broker
        config.put("num.replica.fetchers", "1");

        // Fetch size limits
        config.put("replica.fetch.min.bytes", "1");
        config.put("replica.fetch.max.bytes", "1048576"); // 1MB
        config.put("replica.fetch.wait.max.ms", "500");

        // Socket configuration
        config.put("replica.socket.timeout.ms", "30000");
        config.put("replica.socket.receive.buffer.bytes", "65536");

        // === Backoff and Retry ===

        // Backoff when no data available
        config.put("replica.fetch.backoff.ms", "1000");

        // Response size limit
        config.put("replica.fetch.response.max.bytes", "10485760"); // 10MB

        return config;
    }

    /**
     * Simulates replica fetcher thread behavior
     */
    public static class ReplicaFetcher {

        private final String followerId;
        private final String leaderId;
        private final String topicPartition;
        private final AtomicLong fetchOffset = new AtomicLong(0);
        private final AtomicBoolean running = new AtomicBoolean(true);
        private final ScheduledExecutorService executor =
Executors.newSingleThreadScheduledExecutor();
```

```
        public ReplicaFetcher(String followerId, String leaderId, String
topicPartition) {
            this.followerId = followerId;
            this.leaderId = leaderId;
            this.topicPartition = topicPartition;
        }

        public void start() {
            executor.scheduleWithFixedDelay(this::fetchFromLeader, 0, 500,
TimeUnit.MILLISECONDS);
            System.out.printf("Started replica fetcher: %s -> %s for %s%n",
                followerId, leaderId, topicPartition);
        }

        private void fetchFromLeader() {
            if (!running.get()) return;

            try {
                FetchRequest request = createFetchRequest();
                FetchResponse response = sendFetchRequest(request);
                processFetchResponse(response);

            } catch (Exception e) {
                System.err.printf("Fetch error for %s: %s%n", followerId,
e.getMessage());
                // Implement exponential backoff in real implementation
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException ie) {
                    Thread.currentThread().interrupt();
                    running.set(false);
                }
            }
        }

        private FetchRequest createFetchRequest() {
            return new FetchRequest(
                topicPartition,
                fetchOffset.get(),
                1048576, // max bytes
                500      // max wait ms
            );
        }

        private FetchResponse sendFetchRequest(FetchRequest request) {
            // Simulate network call to leader
            try {
                Thread.sleep((long) (Math.random() * 100)); // Simulate network
latency
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                throw new RuntimeException(e);
            }
```

```
            // Simulate response from leader
            int messageCount = (int) (Math.random() * 10) + 1; // 1-10 messages
            List<LogRecord> records = new ArrayList<>();

            long startOffset = request.fetchOffset;
            for (int i = 0; i < messageCount; i++) {
                records.add(new LogRecord(
                    startOffset + i,
                    "key-" + (startOffset + i),
                    "value-" + (startOffset + i),
                    System.currentTimeMillis()
                ));
            }

            return new FetchResponse(records, startOffset + messageCount);
        }

        private void processFetchResponse(FetchResponse response) {
            if (response.records.isEmpty()) {
                return; // No new data
            }

            // Simulate writing to local log
            System.out.printf("Follower %s fetched %d records (offsets %d-%d)%n",
                followerId, response.records.size(),
                response.records.get(0).offset,
                response.records.get(response.records.size() - 1).offset);

            // Update fetch offset
            fetchOffset.set(response.highWaterMark);

            // Simulate log append time
            try {
                Thread.sleep(10); // Simulate disk write
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                running.set(false);
            }
        }

        public void stop() {
            running.set(false);
            executor.shutdown();
            System.out.printf("Stopped replica fetcher: %s%n", followerId);
        }

        // Data classes
        private static class FetchRequest {
            final String topicPartition;
            final long fetchOffset;
            final int maxBytes;
            final int maxWaitMs;
```

```
            FetchRequest(String topicPartition, long fetchOffset, int maxBytes,
int maxWaitMs) {
                this.topicPartition = topicPartition;
                this.fetchOffset = fetchOffset;
                this.maxBytes = maxBytes;
                this.maxWaitMs = maxWaitMs;
            }
        }

        private static class FetchResponse {
            final List<LogRecord> records;
            final long highWaterMark;

            FetchResponse(List<LogRecord> records, long highWaterMark) {
                this.records = records;
                this.highWaterMark = highWaterMark;
            }
        }

        private static class LogRecord {
            final long offset;
            final String key;
            final String value;
            final long timestamp;

            LogRecord(long offset, String key, String value, long timestamp) {
                this.offset = offset;
                this.key = key;
                this.value = value;
                this.timestamp = timestamp;
            }
        }

        // Example usage
        public static void main(String[] args) throws InterruptedException {
            // Start multiple replica fetchers for different partitions
            List<ReplicaFetcher> fetchers = Arrays.asList(
                new ReplicaFetcher("broker-2", "broker-1", "topic-orders-0"),
                new ReplicaFetcher("broker-3", "broker-1", "topic-orders-0"),
                new ReplicaFetcher("broker-1", "broker-2", "topic-orders-1")
            );

            fetchers.forEach(ReplicaFetcher::start);

            // Run for 30 seconds
            Thread.sleep(30000);

            fetchers.forEach(ReplicaFetcher::stop);
        }
    }
}
```

Leader & Follower Roles

Leader Election and Management

```
/**
 * Demonstrates leader election and management in Kafka
 */
public class LeaderElectionExample {

    /**
     * Leader election configuration
     */
    public static Properties getLeaderElectionConfig() {
        Properties config = new Properties();

        // === Election Configuration ===

        // Enable automatic leader rebalancing
        config.put("auto.leader.rebalance.enable", "true");

        // Percentage of out-of-sync leaders before triggering rebalance
        config.put("leader.imbalance.per.broker.percentage", "10");

        // How often to check for leader imbalance
        config.put("leader.imbalance.check.interval.seconds", "300"); // 5 minutes

        // === Preferred Leader Election ===

        // Whether to enable preferred replica election
        config.put("unclean.leader.election.enable", "false"); // Prefer data
safety

        // Controller socket timeout
        config.put("controller.socket.timeout.ms", "30000");

        // Inter-broker protocol version
        config.put("inter.broker.protocol.version", "3.0");

        return config;
    }

    /**
     * Simulates leader election process
     */
    public static class LeaderElectionSimulator {

        private final Map<String, PartitionReplica> replicas = new
ConcurrentHashMap<>();
        private volatile String currentLeader;
        private volatile Set<String> isr = new HashSet<>();
        private final String partitionName;
```

```
    public LeaderElectionSimulator(String partitionName) {
        this.partitionName = partitionName;
    }

    public void addReplica(String brokerId, boolean isPreferred) {
        replicas.put(brokerId, new PartitionReplica(brokerId, isPreferred));
        isr.add(brokerId);

        if (currentLeader == null && isPreferred) {
            electLeader(brokerId, "initial-election");
        }
    }

    public void simulateBrokerFailure(String brokerId) {
        System.out.printf("\n🔥 Broker %s failed!%n", brokerId);

        // Remove from ISR
        isr.remove(brokerId);

        // If failed broker was leader, trigger election
        if (brokerId.equals(currentLeader)) {
            System.out.println("Leader failed - triggering election");
            triggerLeaderElection("leader-failure");
        }

        PartitionReplica replica = replicas.get(brokerId);
        if (replica != null) {
            replica.isOnline = false;
        }
    }

    public void simulateBrokerRecovery(String brokerId) {
        System.out.printf("\n✅ Broker %s recovered!%n", brokerId);

        PartitionReplica replica = replicas.get(brokerId);
        if (replica != null) {
            replica.isOnline = true;

            // Start catch-up process
            catchUpReplica(brokerId);
        }
    }

    private void catchUpReplica(String brokerId) {
        System.out.printf("Replica %s starting catch-up process%n", brokerId);

        // Simulate catch-up time based on lag
        new Thread(() -> {
            try {
                Thread.sleep(5000); // Simulate catch-up time

                // Add back to ISR once caught up
                isr.add(brokerId);
                System.out.printf("Replica %s caught up and added to ISR%n",
```

```
brokerId);

                    // Check if we should elect preferred leader
                    PartitionReplica replica = replicas.get(brokerId);
                    if (replica.isPreferred && !brokerId.equals(currentLeader)) {
                        triggerPreferredLeaderElection();
                    }

                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }).start();
        }

        private void triggerLeaderElection(String reason) {
            System.out.printf("Triggering leader election for %s (reason: %s)%n",
partitionName, reason);

            // Find best candidate from ISR
            String newLeader = electBestCandidate();

            if (newLeader != null) {
                electLeader(newLeader, reason);
            } else {
                System.err.println("No eligible leader found in ISR!");
            }
        }

        private void triggerPreferredLeaderElection() {
            // Find preferred replica that's in ISR
            String preferredLeader = replicas.entrySet().stream()
                .filter(entry -> entry.getValue().isPreferred)
                .filter(entry -> isr.contains(entry.getKey()))
                .filter(entry -> entry.getValue().isOnline)
                .map(Map.Entry::getKey)
                .findFirst()
                .orElse(null);

            if (preferredLeader != null && !preferredLeader.equals(currentLeader))
{
                System.out.println("Preferred leader available - triggering
preferred leader election");
                electLeader(preferredLeader, "preferred-leader-election");
            }
        }

        private String electBestCandidate() {
            // Election priority:
            // 1. Preferred replica if available and in ISR
            // 2. Any replica in ISR
            // 3. If unclean election enabled, any online replica

            // Try preferred replica first
            String preferredCandidate = replicas.entrySet().stream()
```

```
                .filter(entry -> entry.getValue().isPreferred)
                .filter(entry -> isr.contains(entry.getKey()))
                .filter(entry -> entry.getValue().isOnline)
                .map(Map.Entry::getKey)
                .findFirst()
                .orElse(null);

            if (preferredCandidate != null) {
                return preferredCandidate;
            }

            // Try any ISR replica
            String isrCandidate = isr.stream()
                .filter(brokerId -> replicas.get(brokerId).isOnline)
                .findFirst()
                .orElse(null);

            if (isrCandidate != null) {
                return isrCandidate;
            }

            // Unclean election (if enabled) - any online replica
            boolean uncleanElectionEnabled = false; // Configurable
            if (uncleanElectionEnabled) {
                return replicas.entrySet().stream()
                    .filter(entry -> entry.getValue().isOnline)
                    .map(Map.Entry::getKey)
                    .findFirst()
                    .orElse(null);
            }

            return null;
        }

        private void electLeader(String newLeader, String reason) {
            String oldLeader = currentLeader;
            currentLeader = newLeader;

            System.out.printf("🎯 Leader elected: %s -> %s (reason: %s)%n",
                oldLeader, newLeader, reason);

            // Update replica roles
            for (PartitionReplica replica : replicas.values()) {
                replica.isLeader = replica.brokerId.equals(newLeader);
            }

            printPartitionState();
        }

        public void printPartitionState() {
            System.out.printf("\n=== Partition State: %s ===\n", partitionName);
            System.out.printf("Leader: %s\n", currentLeader);
            System.out.printf("ISR: %s\n", isr);
```

```
            System.out.println("Replicas:");
            for (PartitionReplica replica : replicas.values()) {
                String status = replica.isOnline ? "ONLINE" : "OFFLINE";
                String role = replica.isLeader ? "LEADER" : "FOLLOWER";
                String preferred = replica.isPreferred ? "(PREFERRED)" : "";
                String inIsr = isr.contains(replica.brokerId) ? "[ISR]" : "";

                System.out.printf("  %s: %s %s %s %s\n",
                    replica.brokerId, status, role, preferred, inIsr);
            }
            System.out.println();
        }

        private static class PartitionReplica {
            final String brokerId;
            final boolean isPreferred;
            volatile boolean isLeader = false;
            volatile boolean isOnline = true;

            PartitionReplica(String brokerId, boolean isPreferred) {
                this.brokerId = brokerId;
                this.isPreferred = isPreferred;
            }
        }

        // Example usage
        public static void main(String[] args) throws InterruptedException {
            LeaderElectionSimulator sim = new LeaderElectionSimulator("orders-
partition-0");

            // Set up replicas (broker-1 is preferred leader)
            sim.addReplica("broker-1", true);  // Preferred leader
            sim.addReplica("broker-2", false);
            sim.addReplica("broker-3", false);

            sim.printPartitionState();

            // Simulate leader failure
            Thread.sleep(2000);
            sim.simulateBrokerFailure("broker-1");

            // Simulate recovery
            Thread.sleep(3000);
            sim.simulateBrokerRecovery("broker-1");

            // Wait for preferred leader election
            Thread.sleep(6000);
            sim.printPartitionState();
        }
    }
}
```

🏛️ Metadata & Consensus

Simple Explanation

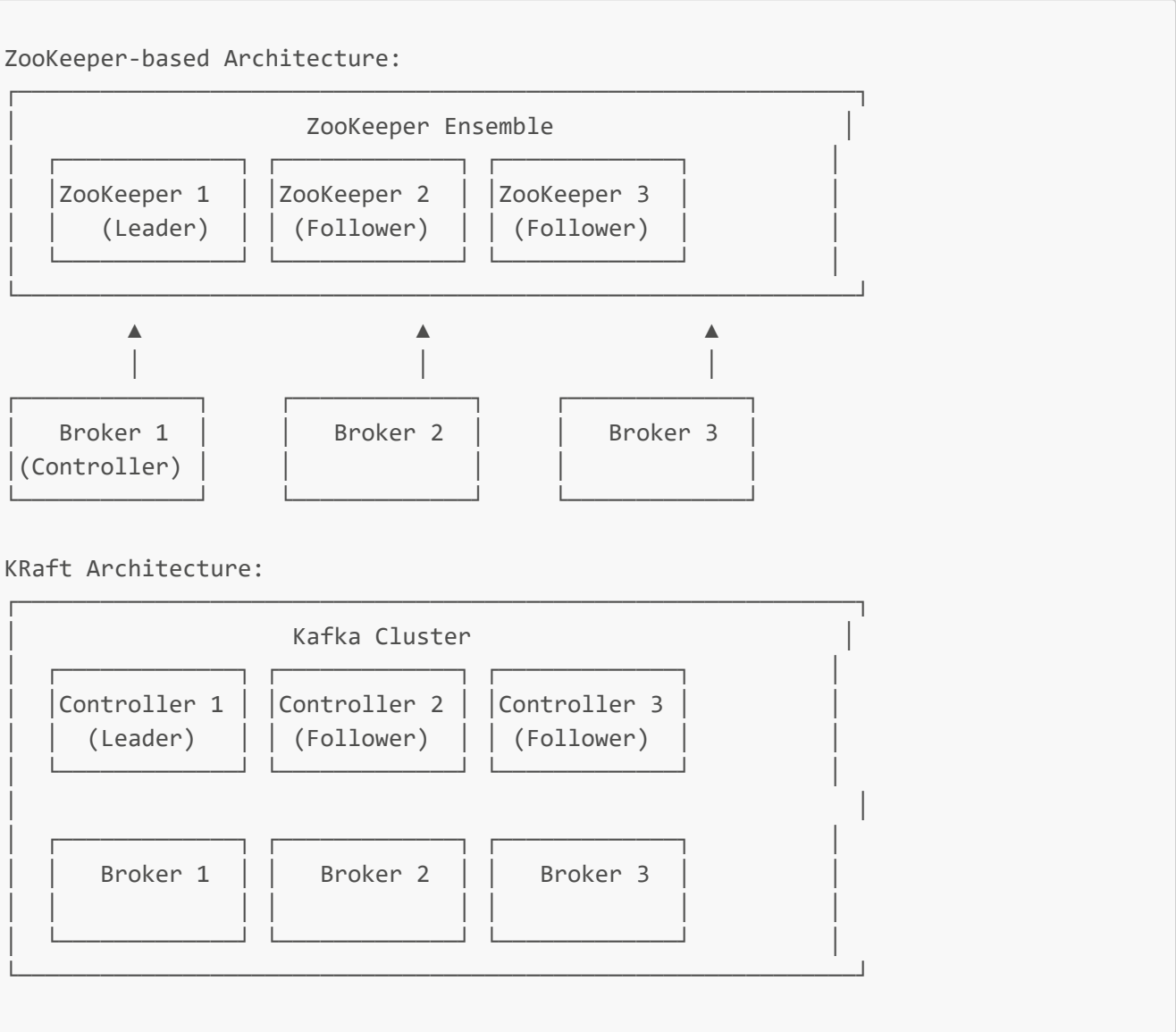
Kafka's metadata and consensus system manages cluster-wide information like topic configurations, partition assignments, and broker membership. Historically managed by ZooKeeper, Kafka now uses its own KRaft (Kafka Raft) protocol for improved performance and operational simplicity.

Problem It Solves

- **Cluster coordination**: Ensures all brokers have consistent metadata
- **Leader election**: Determines which broker leads each partition
- **Configuration management**: Stores and distributes topic and broker configurations
- **Failure detection**: Monitors broker health and triggers failover

ZooKeeper vs KRaft

Architecture Comparison



KRaft Implementation

```java
/**
 * Kafka KRaft (Kafka Raft) configuration and implementation guide
 */
public class KRaftExample {

    /**
     * KRaft controller configuration
     */
    public static Properties getKRaftControllerConfig() {
        Properties config = new Properties();

        // === Process Role Configuration ===

        // Process roles: broker, controller, or both
        config.put("process.roles", "controller"); // controller-only mode
        // config.put("process.roles", "broker"); // broker-only mode
        // config.put("process.roles", "broker,controller"); // combined mode (dev
only)

        // Node ID (must be unique within cluster)
        config.put("node.id", "1");

        // === Controller Quorum Configuration ===

        // Controller quorum voters (must match across all nodes)
        config.put("controller.quorum.voters",
"1@localhost:9093,2@localhost:9094,3@localhost:9095");

        // === Metadata Log Configuration ===

        // Metadata log directory
        config.put("metadata.log.dir", "/var/kafka-logs/__cluster_metadata");

        // Metadata topic configuration
        config.put("metadata.log.segment.bytes", "1073741824"); // 1GB
        config.put("metadata.log.retention.ms", "604800000"); // 7 days
        config.put("metadata.log.retention.bytes", "1073741824000"); // 1TB

        // === Snapshot Configuration ===

        // Snapshot generation threshold
        config.put("metadata.log.max.record.bytes.between.snapshots", "20971520");
// 20MB

        // === Performance Tuning ===

        // Election timeout
        config.put("controller.quorum.election.timeout.ms", "1000");

        // Fetch timeout
        config.put("controller.quorum.fetch.timeout.ms", "2000");

        // Heartbeat interval
```

```
        config.put("controller.quorum.heartbeat.interval.ms", "100");

        // Request timeout
        config.put("controller.quorum.request.timeout.ms", "2000");

        // Retry backoff
        config.put("controller.quorum.retry.backoff.ms", "20");

        return config;
    }

    /**
     * KRaft broker configuration
     */
    public static Properties getKRaftBrokerConfig() {
        Properties config = new Properties();

        // === Process Role Configuration ===

        config.put("process.roles", "broker");
        config.put("node.id", "101"); // Different from controller node IDs

        // === Controller Connection ===

        // Controller quorum voters (same as controller config)
        config.put("controller.quorum.voters",
"1@localhost:9093,2@localhost:9094,3@localhost:9095");

        // Controller listener for brokers
        config.put("controller.listener.names", "CONTROLLER");

        // === Standard Broker Configuration ===

        config.put("listeners",
"PLAINTEXT://localhost:9092,CONTROLLER://localhost:9093");
        config.put("inter.broker.listener.name", "PLAINTEXT");
        config.put("advertised.listeners", "PLAINTEXT://localhost:9092");

        // Log directories
        config.put("log.dirs", "/var/kafka-logs");

        return config;
    }

    /**
     * Demonstrates KRaft cluster management
     */
    public static class KRaftClusterManager {

        private final Set<ControllerNode> controllers = new HashSet<>();
        private final Set<BrokerNode> brokers = new HashSet<>();
        private volatile ControllerNode activeController;

        public void addController(int nodeId, String host, int port) {
```

```
            ControllerNode controller = new ControllerNode(nodeId, host, port);
            controllers.add(controller);

            if (activeController == null) {
                activeController = controller;
                controller.isActive = true;
                System.out.printf("Controller %d elected as active controller%n",
nodeId);
            }

            System.out.printf("Added controller: %d@%s:%d%n", nodeId, host, port);
        }

        public void addBroker(int nodeId, String host, int port) {
            BrokerNode broker = new BrokerNode(nodeId, host, port);
            brokers.add(broker);

            System.out.printf("Added broker: %d@%s:%d%n", nodeId, host, port);

            // Simulate broker registration with controller
            registerBrokerWithController(broker);
        }

        private void registerBrokerWithController(BrokerNode broker) {
            if (activeController != null) {
                System.out.printf("Broker %d registered with controller %d%n",
                    broker.nodeId, activeController.nodeId);

                // In real implementation, this would involve:
                // 1. Broker sending BrokerRegistrationRequest
                // 2. Controller updating metadata log
                // 3. Controller responding with BrokerRegistrationResponse
            }
        }

        public void simulateControllerFailover() {
            if (activeController == null) return;

            System.out.printf("\n🔥 Active controller %d failed!%n",
activeController.nodeId);
            activeController.isActive = false;

            // Simulate leader election
            ControllerNode newLeader = controllers.stream()
                .filter(c -> c != activeController)
                .filter(c -> c.isOnline)
                .findFirst()
                .orElse(null);

            if (newLeader != null) {
                activeController = newLeader;
                newLeader.isActive = true;
                System.out.printf("🎯 Controller %d elected as new active
controller%n",
```

```
                    newLeader.nodeId);

                // Simulate metadata catchup and broker re-registration
                System.out.println("New controller catching up on metadata...");

                for (BrokerNode broker : brokers) {
                    if (broker.isOnline) {
                        registerBrokerWithController(broker);
                    }
                }
            }
        }

        public void printClusterState() {
            System.out.println("\n=== KRaft Cluster State ===");

            System.out.println("Controllers:");
            for (ControllerNode controller : controllers) {
                String status = controller.isOnline ? "ONLINE" : "OFFLINE";
                String role = controller.isActive ? "ACTIVE" : "STANDBY";
                System.out.printf("  Controller %d: %s (%s)%n",
                    controller.nodeId, status, role);
            }

            System.out.println("Brokers:");
            for (BrokerNode broker : brokers) {
                String status = broker.isOnline ? "ONLINE" : "OFFLINE";
                System.out.printf("  Broker %d: %s%n", broker.nodeId, status);
            }

            if (activeController != null) {
                System.out.printf("Active Controller: %d%n",
activeController.nodeId);
            }
            System.out.println();
        }

        private static class ControllerNode {
            final int nodeId;
            final String host;
            final int port;
            volatile boolean isOnline = true;
            volatile boolean isActive = false;

            ControllerNode(int nodeId, String host, int port) {
                this.nodeId = nodeId;
                this.host = host;
                this.port = port;
            }

            @Override
            public boolean equals(Object o) {
                if (this == o) return true;
                if (!(o instanceof ControllerNode)) return false;
```

```
            ControllerNode that = (ControllerNode) o;
            return nodeId == that.nodeId;
        }

        @Override
        public int hashCode() {
            return Integer.hashCode(nodeId);
        }
    }

    private static class BrokerNode {
        final int nodeId;
        final String host;
        final int port;
        volatile boolean isOnline = true;

        BrokerNode(int nodeId, String host, int port) {
            this.nodeId = nodeId;
            this.host = host;
            this.port = port;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (!(o instanceof BrokerNode)) return false;
            BrokerNode that = (BrokerNode) o;
            return nodeId == that.nodeId;
        }

        @Override
        public int hashCode() {
            return Integer.hashCode(nodeId);
        }
    }

    // Example usage
    public static void main(String[] args) throws InterruptedException {
        KRaftClusterManager cluster = new KRaftClusterManager();

        // Add controllers
        cluster.addController(1, "controller-1", 9093);
        cluster.addController(2, "controller-2", 9094);
        cluster.addController(3, "controller-3", 9095);

        // Add brokers
        cluster.addBroker(101, "broker-1", 9092);
        cluster.addBroker(102, "broker-2", 9092);
        cluster.addBroker(103, "broker-3", 9092);

        cluster.printClusterState();

        // Simulate controller failover
        Thread.sleep(2000);
```

```
            cluster.simulateControllerFailover();

            cluster.printClusterState();
        }
    }
}
```

Controller Quorum

Controller Quorum Management

```
/**
 * Demonstrates KRaft controller quorum operations
 */
public class ControllerQuorumExample {

    /**
     * Quorum configuration and monitoring
     */
    public static class QuorumManager {

        private final Map<Integer, QuorumMember> members = new ConcurrentHashMap<>
();
        private volatile Integer leaderId;
        private final AtomicLong currentEpoch = new AtomicLong(0);
        private final ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(2);

        public void addQuorumMember(int nodeId, String endpoint) {
            members.put(nodeId, new QuorumMember(nodeId, endpoint));
            System.out.printf("Added quorum member: %d@%s%n", nodeId, endpoint);

            if (leaderId == null) {
                electLeader();
            }
        }

        public void start() {
            // Monitor quorum health
            scheduler.scheduleAtFixedRate(this::checkQuorumHealth, 0, 5,
TimeUnit.SECONDS);

            // Simulate heartbeats
            scheduler.scheduleAtFixedRate(this::sendHeartbeats, 0, 1,
TimeUnit.SECONDS);

            System.out.println("Quorum manager started");
        }

        private void electLeader() {
            // Simple leader election - highest node ID wins in this simulation
```

```
        // Real KRaft uses more sophisticated election algorithm
        Integer newLeader = members.keySet().stream()
            .filter(nodeId -> members.get(nodeId).isOnline)
            .max(Integer::compare)
            .orElse(null);

        if (newLeader != null && !newLeader.equals(leaderId)) {
            Integer oldLeader = leaderId;
            leaderId = newLeader;
            currentEpoch.incrementAndGet();

            System.out.printf("🎯 Leader elected: %s -> %d (epoch: %d)%n",
                oldLeader, newLeader, currentEpoch.get());

            // Update member roles
            for (QuorumMember member : members.values()) {
                member.isLeader = member.nodeId == newLeader;
            }

            // Simulate metadata log replication
            if (oldLeader != null) {
                System.out.println("New leader replicating metadata log...");
            }
        }
    }

    private void checkQuorumHealth() {
        System.out.println("\n=== Quorum Health Check ===");

        int onlineMembers = (int) members.values().stream()
            .mapToInt(m -> m.isOnline ? 1 : 0)
            .sum();

        int requiredQuorum = (members.size() / 2) + 1;
        boolean hasQuorum = onlineMembers >= requiredQuorum;

        System.out.printf("Online members: %d/%d (quorum: %s)%n",
            onlineMembers, members.size(), hasQuorum ? "YES" : "NO");

        if (!hasQuorum) {
            System.err.println("⚠️  QUORUM LOST - Cluster unavailable!");
        }

        // Check if leader is still online
        if (leaderId != null && !members.get(leaderId).isOnline) {
            System.err.println("Leader is offline - triggering election");
            leaderId = null;
            electLeader();
        }

        printQuorumState();
    }

    private void sendHeartbeats() {
```

```
            if (leaderId == null) return;

            QuorumMember leader = members.get(leaderId);
            if (!leader.isOnline) return;

            // Leader sends heartbeats to followers
            for (QuorumMember member : members.values()) {
                if (member.nodeId != leaderId && member.isOnline) {
                    member.lastHeartbeat = System.currentTimeMillis();

                    // Simulate network issues occasionally
                    if (Math.random() < 0.05) { // 5% chance of network issue
                        System.out.printf("Network issue: heartbeat lost to %d%n",
member.nodeId);
                        member.consecutiveFailures++;

                        if (member.consecutiveFailures >= 3) {
                            System.out.printf("Member %d considered offline%n",
member.nodeId);
                            member.isOnline = false;
                        }
                    } else {
                        member.consecutiveFailures = 0;
                    }
                }
            }
        }

        public void simulateMemberFailure(int nodeId) {
            QuorumMember member = members.get(nodeId);
            if (member != null) {
                member.isOnline = false;
                System.out.printf("🔥 Member %d failed!%n", nodeId);

                if (nodeId == leaderId) {
                    leaderId = null;
                    electLeader();
                }
            }
        }

        public void simulateMemberRecovery(int nodeId) {
            QuorumMember member = members.get(nodeId);
            if (member != null) {
                member.isOnline = true;
                member.consecutiveFailures = 0;
                System.out.printf("✅ Member %d recovered!%n", nodeId);

                // Trigger election if no current leader
                if (leaderId == null) {
                    electLeader();
                }
            }
        }
```

```
        private void printQuorumState() {
            System.out.println("Quorum Members:");
            for (QuorumMember member : members.values()) {
                String status = member.isOnline ? "ONLINE" : "OFFLINE";
                String role = member.isLeader ? "LEADER" : "FOLLOWER";
                System.out.printf("  %d: %s (%s) - failures: %d%n",
                    member.nodeId, status, role, member.consecutiveFailures);
            }

            if (leaderId != null) {
                System.out.printf("Current Leader: %d (epoch: %d)%n", leaderId,
currentEpoch.get());
            }
            System.out.println();
        }

        public void stop() {
            scheduler.shutdown();
        }

        private static class QuorumMember {
            final int nodeId;
            final String endpoint;
            volatile boolean isOnline = true;
            volatile boolean isLeader = false;
            volatile long lastHeartbeat = System.currentTimeMillis();
            volatile int consecutiveFailures = 0;

            QuorumMember(int nodeId, String endpoint) {
                this.nodeId = nodeId;
                this.endpoint = endpoint;
            }
        }

        // Example usage
        public static void main(String[] args) throws InterruptedException {
            QuorumManager quorum = new QuorumManager();

            // Add 3-member quorum
            quorum.addQuorumMember(1, "controller-1:9093");
            quorum.addQuorumMember(2, "controller-2:9094");
            quorum.addQuorumMember(3, "controller-3:9095");

            quorum.start();

            // Run for a while
            Thread.sleep(10000);

            // Simulate failures and recoveries
            quorum.simulateMemberFailure(3);
            Thread.sleep(5000);

            quorum.simulateMemberFailure(2);
```

```
            Thread.sleep(5000);

            quorum.simulateMemberRecovery(3);
            Thread.sleep(5000);

            quorum.stop();
        }
    }
}
```

Partition Reassignment

Partition Reassignment Implementation

```
/**
 * Demonstrates partition reassignment process in Kafka
 */
public class PartitionReassignmentExample {

    /**
     * Partition reassignment configuration
     */
    public static Properties getReassignmentConfig() {
        Properties config = new Properties();

        // === Reassignment Performance ===

        // Throttle reassignment bandwidth (bytes/sec per broker)
        config.put("leader.replication.throttled.rate", "104857600"); // 100MB/s
        config.put("follower.replication.throttled.rate", "104857600"); // 100MB/s

        // Inter-broker throttle
        config.put("replica.alter.log.dirs.io.max.bytes.per.second", "104857600");

        // === Reassignment Monitoring ===

        // How often to check reassignment progress
        config.put("replica.lag.time.max.ms", "30000");

        // Socket timeout for reassignment
        config.put("replica.socket.timeout.ms", "30000");

        return config;
    }

    /**
     * Simulates partition reassignment process
     */
    public static class PartitionReassigner {

        private final Map<String, PartitionInfo> partitions = new
```

```
ConcurrentHashMap<>();
        private final Map<String, ReassignmentTask> activeReassignments = new
ConcurrentHashMap<>();
        private final ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(2);

        public void addPartition(String topicPartition, List<Integer>
currentReplicas) {
            partitions.put(topicPartition, new PartitionInfo(topicPartition,
currentReplicas));
            System.out.printf("Added partition %s with replicas %s%n",
topicPartition, currentReplicas);
        }

        public void reassignPartition(String topicPartition, List<Integer>
newReplicas) {
            PartitionInfo partition = partitions.get(topicPartition);
            if (partition == null) {
                System.err.println("Partition not found: " + topicPartition);
                return;
            }

            if (partition.currentReplicas.equals(newReplicas)) {
                System.out.println("No reassignment needed - replicas unchanged");
                return;
            }

            System.out.printf("Starting reassignment: %s from %s to %s%n",
                topicPartition, partition.currentReplicas, newReplicas);

            ReassignmentTask task = new ReassignmentTask(topicPartition,
                partition.currentReplicas, newReplicas);

            activeReassignments.put(topicPartition, task);

            // Start reassignment process
            executeReassignment(task);
        }

        private void executeReassignment(ReassignmentTask task) {
            // Phase 1: Add new replicas to ISR
            System.out.printf("Phase 1: Adding new replicas %s to %s%n",
                getAddedReplicas(task), task.topicPartition);

            // Simulate adding new replicas
            scheduler.schedule(() -> {
                task.currentStep = ReassignmentStep.ADDING_REPLICAS;
                simulateReplicaAddition(task);
            }, 1, TimeUnit.SECONDS);
        }

        private void simulateReplicaAddition(ReassignmentTask task) {
            List<Integer> addedReplicas = getAddedReplicas(task);
```

```
            for (Integer replica : addedReplicas) {
                System.out.printf("Adding replica %d for %s - copying data...%n",
                    replica, task.topicPartition);

                // Simulate data copy progress
                scheduler.schedule(() -> {
                    System.out.printf("Replica %d for %s: 50%% copied%n", replica,
task.topicPartition);
                }, 2, TimeUnit.SECONDS);

                scheduler.schedule(() -> {
                    System.out.printf("Replica %d for %s: 100%% copied, added to
ISR%n",
                        replica, task.topicPartition);

                    // Check if all new replicas are added
                    checkAdditionComplete(task);
                }, 4, TimeUnit.SECONDS);
            }
        }

        private void checkAdditionComplete(ReassignmentTask task) {
            // In real implementation, check if all new replicas are in ISR
            task.additionComplete = true;

            if (task.additionComplete) {
                System.out.printf("Phase 2: Removing old replicas %s from %s%n",
                    getRemovedReplicas(task), task.topicPartition);

                scheduler.schedule(() -> {
                    task.currentStep = ReassignmentStep.REMOVING_REPLICAS;
                    simulateReplicaRemoval(task);
                }, 1, TimeUnit.SECONDS);
            }
        }

        private void simulateReplicaRemoval(ReassignmentTask task) {
            List<Integer> removedReplicas = getRemovedReplicas(task);

            for (Integer replica : removedReplicas) {
                System.out.printf("Removing replica %d from %s%n", replica,
task.topicPartition);

                scheduler.schedule(() -> {
                    System.out.printf("Replica %d removed from %s%n", replica,
task.topicPartition);

                    // Check if all old replicas are removed
                    checkRemovalComplete(task);
                }, 2, TimeUnit.SECONDS);
            }
        }

        private void checkRemovalComplete(ReassignmentTask task) {
```

```
            task.removalComplete = true;

            if (task.removalComplete) {
                // Update partition info
                PartitionInfo partition = partitions.get(task.topicPartition);
                partition.currentReplicas = new ArrayList<>(task.newReplicas);

                task.currentStep = ReassignmentStep.COMPLETED;
                activeReassignments.remove(task.topicPartition);

                System.out.printf("✅ Reassignment completed for %s: final
replicas %s%n",
                    task.topicPartition, task.newReplicas);
            }
        }

        private List<Integer> getAddedReplicas(ReassignmentTask task) {
            return task.newReplicas.stream()
                .filter(replica -> !task.oldReplicas.contains(replica))
                .collect(Collectors.toList());
        }

        private List<Integer> getRemovedReplicas(ReassignmentTask task) {
            return task.oldReplicas.stream()
                .filter(replica -> !task.newReplicas.contains(replica))
                .collect(Collectors.toList());
        }

        public void checkReassignmentStatus() {
            System.out.println("\n=== Reassignment Status ===");

            if (activeReassignments.isEmpty()) {
                System.out.println("No active reassignments");
                return;
            }

            for (ReassignmentTask task : activeReassignments.values()) {
                System.out.printf("Partition %s: %s%n", task.topicPartition,
task.currentStep);
                System.out.printf("  From: %s -> To: %s%n", task.oldReplicas,
task.newReplicas);

                if (task.currentStep == ReassignmentStep.ADDING_REPLICAS) {
                    System.out.printf("  Adding: %s%n", getAddedReplicas(task));
                } else if (task.currentStep == ReassignmentStep.REMOVING_REPLICAS)
{
                    System.out.printf("  Removing: %s%n",
getRemovedReplicas(task));
                }
            }
            System.out.println();
        }

        public void generateReassignmentPlan(List<String> topics, List<Integer>
```

```
brokers) {
            System.out.println("\n=== Generated Reassignment Plan ===");

            // Simple round-robin assignment
            int replicationFactor = 3;

            for (String topic : topics) {
                // Simulate multiple partitions per topic
                for (int partition = 0; partition < 3; partition++) {
                    String topicPartition = topic + "-" + partition;
                    List<Integer> assignedBrokers = new ArrayList<>();

                    for (int i = 0; i < replicationFactor; i++) {
                        int brokerIndex = (partition + i) % brokers.size();
                        assignedBrokers.add(brokers.get(brokerIndex));
                    }

                    System.out.printf("  %s: %s%n", topicPartition,
assignedBrokers);
                }
            }

            System.out.println("\nJSON format:");
            System.out.println("{");
            System.out.println("  \"version\": 1,");
            System.out.println("  \"partitions\": [");

            boolean first = true;
            for (String topic : topics) {
                for (int partition = 0; partition < 3; partition++) {
                    if (!first) System.out.println(",");

                    List<Integer> assignedBrokers = new ArrayList<>();
                    for (int i = 0; i < replicationFactor; i++) {
                        int brokerIndex = (partition + i) % brokers.size();
                        assignedBrokers.add(brokers.get(brokerIndex));
                    }

                    System.out.printf("    {\"topic\": \"%s\", \"partition\": %d,
\"replicas\": %s}",
                        topic, partition, assignedBrokers);

                    first = false;
                }
            }

            System.out.println();
            System.out.println("  ]");
            System.out.println("}");
        }

        private static class PartitionInfo {
            final String topicPartition;
            volatile List<Integer> currentReplicas;
```

```
        PartitionInfo(String topicPartition, List<Integer> currentReplicas) {
            this.topicPartition = topicPartition;
            this.currentReplicas = new ArrayList<>(currentReplicas);
        }
    }

    private static class ReassignmentTask {
        final String topicPartition;
        final List<Integer> oldReplicas;
        final List<Integer> newReplicas;
        volatile ReassignmentStep currentStep = ReassignmentStep.STARTED;
        volatile boolean additionComplete = false;
        volatile boolean removalComplete = false;

        ReassignmentTask(String topicPartition, List<Integer> oldReplicas,
                         List<Integer> newReplicas) {
            this.topicPartition = topicPartition;
            this.oldReplicas = new ArrayList<>(oldReplicas);
            this.newReplicas = new ArrayList<>(newReplicas);
        }
    }

    private enum ReassignmentStep {
        STARTED,
        ADDING_REPLICAS,
        REMOVING_REPLICAS,
        COMPLETED
    }

    public void stop() {
        scheduler.shutdown();
    }

    // Example usage
    public static void main(String[] args) throws InterruptedException {
        PartitionReassigner reassigner = new PartitionReassigner();

        // Add some partitions
        reassigner.addPartition("orders-0", Arrays.asList(1, 2, 3));
        reassigner.addPartition("orders-1", Arrays.asList(2, 3, 4));
        reassigner.addPartition("users-0", Arrays.asList(1, 3, 5));

        // Generate reassignment plan for adding new brokers
        reassigner.generateReassignmentPlan(
            Arrays.asList("orders", "users"),
            Arrays.asList(1, 2, 3, 4, 5, 6)
        );

        // Simulate reassignment
        reassigner.reassignPartition("orders-0", Arrays.asList(2, 4, 6));

        // Monitor progress
        for (int i = 0; i < 6; i++) {
```

```
                Thread.sleep(2000);
                reassigner.checkReassignmentStatus();
            }

            reassigner.stop();
        }
    }
}
```

Comprehensive Java Examples

Production Kafka Internals Monitoring System

```
import javax.management.MBeanServer;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;

/**
 * Comprehensive Kafka internals monitoring and management system
 */
public class KafkaInternalsMonitor {

    private final MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
    private final ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(3);
    private final Map<String, MetricSnapshot> metricHistory = new
ConcurrentHashMap<>();

    public void startMonitoring() {
        // Monitor storage metrics
        scheduler.scheduleAtFixedRate(this::monitorStorageMetrics, 0, 30,
TimeUnit.SECONDS);

        // Monitor replication metrics
        scheduler.scheduleAtFixedRate(this::monitorReplicationMetrics, 0, 30,
TimeUnit.SECONDS);

        // Monitor controller metrics
        scheduler.scheduleAtFixedRate(this::monitorControllerMetrics, 0, 60,
TimeUnit.SECONDS);

        System.out.println("Kafka internals monitoring started");
    }

    private void monitorStorageMetrics() {
        System.out.println("\n=== Storage Metrics ===");

        try {
            // Log size metrics
            ObjectName logSizePattern = new
```

```
ObjectName("kafka.log:type=LogSize,name=Size,topic=*,partition=*");
            Set<ObjectName> logSizeObjects = mbs.queryNames(logSizePattern, null);

            long totalLogSize = 0;
            for (ObjectName objName : logSizeObjects) {
                Long size = (Long) mbs.getAttribute(objName, "Value");
                totalLogSize += size;

                String topic = objName.getKeyProperty("topic");
                String partition = objName.getKeyProperty("partition");
                System.out.printf("  %s-%s: %s%n", topic, partition,
formatBytes(size));
            }

            System.out.printf("Total log size: %s%n", formatBytes(totalLogSize));

            // Segment metrics
            monitorSegmentMetrics();

            // Retention metrics
            monitorRetentionMetrics();

        } catch (Exception e) {
            System.err.println("Error monitoring storage metrics: " +
e.getMessage());
        }
    }

    private void monitorSegmentMetrics() {
        System.out.println("\nSegment Information:");

        // In a real implementation, you would access Kafka's internal metrics
        // Here we simulate the data

        Map<String, SegmentMetrics> segmentData = getSegmentMetrics();

        for (Map.Entry<String, SegmentMetrics> entry : segmentData.entrySet()) {
            SegmentMetrics metrics = entry.getValue();
            System.out.printf("  %s:%n", entry.getKey());
            System.out.printf("    Active segment: %s (%s)%n",
                metrics.activeSegmentName,
formatBytes(metrics.activeSegmentSize));
            System.out.printf("    Total segments: %d%n", metrics.totalSegments);
            System.out.printf("    Index size: %s%n",
formatBytes(metrics.indexSize));
            System.out.printf("    Time index size: %s%n",
formatBytes(metrics.timeIndexSize));
        }
    }

    private void monitorRetentionMetrics() {
        System.out.println("\nRetention Activity:");

        // Monitor retention/cleanup activity
```

```
        RetentionMetrics retention = getRetentionMetrics();

        System.out.printf("  Segments deleted (last hour): %d%n",
retention.segmentsDeleted);
        System.out.printf("  Bytes reclaimed: %s%n",
formatBytes(retention.bytesReclaimed));
        System.out.printf("  Compaction runs: %d%n", retention.compactionRuns);
        System.out.printf("  Compaction ratio: %.2f%%%n",
retention.compactionRatio * 100);
    }

    private void monitorReplicationMetrics() {
        System.out.println("\n=== Replication Metrics ===");

        try {
            // ISR metrics
            monitorISRMetrics();

            // Replica lag metrics
            monitorReplicaLag();

            // Leader election metrics
            monitorLeaderElection();

        } catch (Exception e) {
            System.err.println("Error monitoring replication metrics: " +
e.getMessage());
        }
    }

    private void monitorISRMetrics() {
        System.out.println("\nISR Status:");

        // In a real implementation, access ISR metrics from JMX
        Map<String, ISRMetrics> isrData = getISRMetrics();

        for (Map.Entry<String, ISRMetrics> entry : isrData.entrySet()) {
            ISRMetrics metrics = entry.getValue();
            System.out.printf("  %s:%n", entry.getKey());
            System.out.printf("    ISR size: %d/%d%n", metrics.isrSize,
metrics.replicationFactor);
            System.out.printf("    Under-replicated: %s%n",
metrics.isUnderReplicated ? "YES" : "NO");

            if (metrics.isUnderReplicated) {
                System.out.printf("    Missing replicas: %s%n",
metrics.missingReplicas);
            }
        }
    }

    private void monitorReplicaLag() {
        System.out.println("\nReplica Lag:");
```

```
        Map<String, ReplicaLagMetrics> lagData = getReplicaLagMetrics();

        for (Map.Entry<String, ReplicaLagMetrics> entry : lagData.entrySet()) {
            ReplicaLagMetrics metrics = entry.getValue();
            System.out.printf("  %s:%n", entry.getKey());
            System.out.printf("    Max lag: %d messages%n", metrics.maxLag);
            System.out.printf("    Avg lag: %.1f messages%n", metrics.avgLag);
            System.out.printf("    Lagging replicas: %d%n",
metrics.laggingReplicas);
        }
    }

    private void monitorLeaderElection() {
        System.out.println("\nLeader Elections:");

        LeaderElectionMetrics metrics = getLeaderElectionMetrics();

        System.out.printf("  Elections (last hour): %d%n",
metrics.electionsLastHour);
        System.out.printf("  Unclean elections: %d%n", metrics.uncleanElections);
        System.out.printf("  Preferred leader ratio: %.2f%%%n",
metrics.preferredLeaderRatio * 100);
    }

    private void monitorControllerMetrics() {
        System.out.println("\n=== Controller Metrics ===");

        try {
            ControllerMetrics metrics = getControllerMetrics();

            System.out.printf("Active controller: %d%n",
metrics.activeControllerId);
            System.out.printf("Controller epoch: %d%n", metrics.controllerEpoch);
            System.out.printf("Metadata lag: %d records%n", metrics.metadataLag);
            System.out.printf("Pending requests: %d%n", metrics.pendingRequests);

            if (metrics.isKRaft) {
                System.out.println("\nKRaft Metrics:");
                System.out.printf("  Quorum size: %d%n", metrics.quorumSize);
                System.out.printf("  Leader epoch: %d%n", metrics.leaderEpoch);
                System.out.printf("  Log end offset: %d%n", metrics.logEndOffset);
                System.out.printf("  High water mark: %d%n",
metrics.highWaterMark);
            }

        } catch (Exception e) {
            System.err.println("Error monitoring controller metrics: " +
e.getMessage());
        }
    }

    // Simulated metric getters (in real implementation, these would access JMX)
    private Map<String, SegmentMetrics> getSegmentMetrics() {
        Map<String, SegmentMetrics> metrics = new HashMap<>();
```

```
        metrics.put("orders-0", new SegmentMetrics(
            "00000000000001234567.log", 523456789, 15, 245760, 81920));
        metrics.put("users-0", new SegmentMetrics(
            "00000000000000987654.log", 234567890, 8, 163840, 65536));
        return metrics;
    }

    private RetentionMetrics getRetentionMetrics() {
        return new RetentionMetrics(5, 2147483648L, 3, 0.35);
    }

    private Map<String, ISRMetrics> getISRMetrics() {
        Map<String, ISRMetrics> metrics = new HashMap<>();
        metrics.put("orders-0", new ISRMetrics(3, 3, false,
Collections.emptyList()));
        metrics.put("users-0", new ISRMetrics(2, 3, true, Arrays.asList(3)));
        return metrics;
    }

    private Map<String, ReplicaLagMetrics> getReplicaLagMetrics() {
        Map<String, ReplicaLagMetrics> metrics = new HashMap<>();
        metrics.put("orders-0", new ReplicaLagMetrics(50, 12.5, 0));
        metrics.put("users-0", new ReplicaLagMetrics(1250, 845.2, 1));
        return metrics;
    }

    private LeaderElectionMetrics getLeaderElectionMetrics() {
        return new LeaderElectionMetrics(2, 0, 0.95);
    }

    private ControllerMetrics getControllerMetrics() {
        return new ControllerMetrics(1, 45, 0, 5, true, 3, 125, 12456, 12450);
    }

    private String formatBytes(long bytes) {
        if (bytes < 1024) return bytes + " B";
        if (bytes < 1024 * 1024) return String.format("%.1f KB", bytes / 1024.0);
        if (bytes < 1024 * 1024 * 1024) return String.format("%.1f MB", bytes /
(1024.0 * 1024));
        return String.format("%.1f GB", bytes / (1024.0 * 1024 * 1024));
    }

    public void stop() {
        scheduler.shutdown();
    }

    // Metric data classes
    private static class SegmentMetrics {
        final String activeSegmentName;
        final long activeSegmentSize;
        final int totalSegments;
        final long indexSize;
        final long timeIndexSize;
```

```
        SegmentMetrics(String activeSegmentName, long activeSegmentSize, int
totalSegments,
                       long indexSize, long timeIndexSize) {
            this.activeSegmentName = activeSegmentName;
            this.activeSegmentSize = activeSegmentSize;
            this.totalSegments = totalSegments;
            this.indexSize = indexSize;
            this.timeIndexSize = timeIndexSize;
        }
    }

    private static class RetentionMetrics {
        final int segmentsDeleted;
        final long bytesReclaimed;
        final int compactionRuns;
        final double compactionRatio;

        RetentionMetrics(int segmentsDeleted, long bytesReclaimed, int
compactionRuns, double compactionRatio) {
            this.segmentsDeleted = segmentsDeleted;
            this.bytesReclaimed = bytesReclaimed;
            this.compactionRuns = compactionRuns;
            this.compactionRatio = compactionRatio;
        }
    }

    private static class ISRMetrics {
        final int isrSize;
        final int replicationFactor;
        final boolean isUnderReplicated;
        final List<Integer> missingReplicas;

        ISRMetrics(int isrSize, int replicationFactor, boolean isUnderReplicated,
List<Integer> missingReplicas) {
            this.isrSize = isrSize;
            this.replicationFactor = replicationFactor;
            this.isUnderReplicated = isUnderReplicated;
            this.missingReplicas = missingReplicas;
        }
    }

    private static class ReplicaLagMetrics {
        final long maxLag;
        final double avgLag;
        final int laggingReplicas;

        ReplicaLagMetrics(long maxLag, double avgLag, int laggingReplicas) {
            this.maxLag = maxLag;
            this.avgLag = avgLag;
            this.laggingReplicas = laggingReplicas;
        }
    }

    private static class LeaderElectionMetrics {
```

```
        final int electionsLastHour;
        final int uncleanElections;
        final double preferredLeaderRatio;

        LeaderElectionMetrics(int electionsLastHour, int uncleanElections, double
preferredLeaderRatio) {
            this.electionsLastHour = electionsLastHour;
            this.uncleanElections = uncleanElections;
            this.preferredLeaderRatio = preferredLeaderRatio;
        }
    }

    private static class ControllerMetrics {
        final int activeControllerId;
        final long controllerEpoch;
        final long metadataLag;
        final int pendingRequests;
        final boolean isKRaft;
        final int quorumSize;
        final long leaderEpoch;
        final long logEndOffset;
        final long highWaterMark;

        ControllerMetrics(int activeControllerId, long controllerEpoch, long
metadataLag,
                          int pendingRequests, boolean isKRaft, int quorumSize,
                          long leaderEpoch, long logEndOffset, long highWaterMark)
{
            this.activeControllerId = activeControllerId;
            this.controllerEpoch = controllerEpoch;
            this.metadataLag = metadataLag;
            this.pendingRequests = pendingRequests;
            this.isKRaft = isKRaft;
            this.quorumSize = quorumSize;
            this.leaderEpoch = leaderEpoch;
            this.logEndOffset = logEndOffset;
            this.highWaterMark = highWaterMark;
        }
    }

    private static class MetricSnapshot {
        final long timestamp;
        final double value;

        MetricSnapshot(long timestamp, double value) {
            this.timestamp = timestamp;
            this.value = value;
        }
    }

    // Example usage
    public static void main(String[] args) throws InterruptedException {
        KafkaInternalsMonitor monitor = new KafkaInternalsMonitor();
        monitor.startMonitoring();
```

```
        // Run monitoring for 5 minutes
        Thread.sleep(300000);

        monitor.stop();
    }
}
```

⚖️ Comparisons & Trade-offs

Storage Configuration Trade-offs

Configuration	High Throughput	Low Latency	Storage Efficiency	Operational Complexity
Large Segments	☑ Good	❌ Poor	☑ Good	☑ Simple
Small Segments	❌ Poor	☑ Good	❌ Poor	❌ Complex
Dense Indexing	❌ Poor	☑ Good	❌ Poor	❌ Complex
Sparse Indexing	☑ Good	❌ Poor	☑ Good	☑ Simple
Log Compaction	❌ Poor	❌ Poor	☑ Excellent	❌ Complex
Time Retention	☑ Good	☑ Good	❌ Poor	☑ Simple
Tiered Storage	❌ Fair	❌ Poor	☑ Excellent	❌ Very Complex

Replication Trade-offs

Aspect	More Replicas	Fewer Replicas
Durability	Higher	Lower
Availability	Higher	Lower
Throughput	Lower (more replication overhead)	Higher
Storage Cost	Higher	Lower
Network Usage	Higher	Lower
Recovery Time	Faster (more sources)	Slower

Consensus Protocol Comparison

Feature	ZooKeeper Mode	KRaft Mode
Operational Complexity	High (2 systems)	Lower (1 system)
Metadata Performance	Slower	Faster
Scalability	Limited	Better

Feature	ZooKeeper Mode	KRaft Mode
Network Overhead	Higher	Lower
Recovery Time	Slower	Faster
Maturity	Very High	Growing
Migration Effort	None	High

Common Pitfalls & Best Practices

Storage Configuration Pitfalls

❌ Incorrect Segment Sizing

```
// DON'T - Segment too small for high throughput
Properties badConfig = new Properties();
badConfig.put("log.segment.bytes", "1048576"); // 1MB - too small!
badConfig.put("log.roll.ms", "60000"); // 1 minute - too frequent!
```

```
// DO - Appropriate segment sizing
Properties goodConfig = new Properties();
badConfig.put("log.segment.bytes", "1073741824"); // 1GB for high throughput
badConfig.put("log.roll.ms", "604800000"); // 7 days
```

❌ Index Configuration Mistakes

```
// DON'T - Index too dense (performance impact)
config.put("log.index.interval.bytes", "512"); // Too frequent indexing
config.put("log.index.size.max.bytes", "1048576"); // Too small index file
```

```
// DO - Balanced index configuration
config.put("log.index.interval.bytes", "4096"); // Default 4KB
config.put("log.index.size.max.bytes", "10485760"); // 10MB index size
```

Replication Best Practices

☑ Optimal ISR Configuration

```
Properties isrConfig = new Properties();
```

```
// Allow sufficient time for transient network issues
isrConfig.put("replica.lag.time.max.ms", "30000"); // 30 seconds

// Balance between throughput and safety
isrConfig.put("min.insync.replicas", "2"); // For RF=3

// Adequate fetch sizes
isrConfig.put("replica.fetch.max.bytes", "1048576"); // 1MB
isrConfig.put("num.replica.fetchers", "2"); // Multiple fetcher threads
```

❌ Dangerous Replication Settings

```
// DON'T - Allow unclean leader election in production
config.put("unclean.leader.election.enable", "true"); // Data loss risk!

// DON'T - Set min.insync.replicas too low
config.put("min.insync.replicas", "1"); // No safety margin
```

KRaft Migration Considerations

☑ Safe Migration Approach

```
public class KRaftMigrationGuide {

    public static void safeMigrationSteps() {
        System.out.println("KRaft Migration Checklist:");
        System.out.println("1. Upgrade to Kafka 3.4+ (migration support)");
        System.out.println("2. Test thoroughly in development");
        System.out.println("3. Backup ZooKeeper data");
        System.out.println("4. Plan rollback procedures");
        System.out.println("5. Migrate during maintenance window");
        System.out.println("6. Monitor cluster health closely");
    }

    // Pre-migration validation
    public static boolean validateMigrationReadiness() {
        // Check Kafka version compatibility
        // Verify no deprecated features in use
        // Ensure adequate monitoring
        // Validate backup procedures
        return true;
    }
}
```

Best Practices Summary

☑ Storage Engine Best Practices

1. **Choose segment size based on use case** - Larger for throughput, smaller for latency
2. **Monitor disk usage** - Set up alerts for storage consumption
3. **Test retention policies** - Validate cleanup is working as expected
4. **Plan for tiered storage** - Consider long-term storage strategy
5. **Optimize index density** - Balance between lookup speed and overhead

☑ Replication Best Practices

1. **Set appropriate replication factor** - Usually 3 for production
2. **Monitor ISR closely** - Alert on under-replicated partitions
3. **Configure producer acks properly** - Use `acks=all` for durability
4. **Plan for broker failures** - Ensure adequate ISR for availability
5. **Test failover scenarios** - Regular disaster recovery exercises

☑ Metadata & Consensus Best Practices

1. **Migrate to KRaft when ready** - Plan migration carefully
2. **Monitor controller metrics** - Watch for election frequency
3. **Use dedicated controller nodes** - In large clusters
4. **Backup metadata regularly** - Essential for disaster recovery
5. **Plan controller capacity** - Size appropriately for metadata volume

---

🌍 Real-World Use Cases

High-Throughput Analytics Pipeline

```
public class AnalyticsPipelineInternals {

    public static Properties getAnalyticsStorageConfig() {
        Properties config = new Properties();

        // Large segments for batch processing
        config.put("log.segment.bytes", "2147483648"); // 2GB
        config.put("log.roll.ms", "86400000"); // 24 hours

        // Time-based retention for analytics data
        config.put("log.retention.hours", "720"); // 30 days
        config.put("log.cleanup.policy", "delete");

        // Optimize for sequential reads
        config.put("log.index.interval.bytes", "8192"); // Less frequent indexing

        // Tiered storage for cost optimization
        config.put("remote.storage.enable", "true");
        config.put("local.retention.hours", "24"); // 1 day local

        return config;
    }
}
```

Real-Time CDC Implementation

```
public class CDCInternals {

    public static Properties getCDCStorageConfig() {
        Properties config = new Properties();

        // Log compaction for state management
        config.put("log.cleanup.policy", "compact");
        config.put("log.cleaner.min.cleanable.ratio", "0.1");
        config.put("log.cleaner.max.compaction.lag.ms", "300000"); // 5 minutes

        // Small segments for low latency
        config.put("log.segment.bytes", "268435456"); // 256MB
        config.put("log.roll.ms", "3600000"); // 1 hour

        // Dense indexing for quick lookups
        config.put("log.index.interval.bytes", "2048"); // 2KB

        return config;
    }

    public static Properties getCDCReplicationConfig() {
        Properties config = new Properties();

        // High availability requirements
        config.put("default.replication.factor", "3");
        config.put("min.insync.replicas", "2");

        // Fast failure detection
        config.put("replica.lag.time.max.ms", "10000"); // 10 seconds

        // Optimize for low latency
        config.put("replica.fetch.wait.max.ms", "100");
        config.put("replica.fetch.min.bytes", "1");

        return config;
    }
}
```

Financial Services Compliance

```
public class ComplianceInternals {

    public static Properties getComplianceStorageConfig() {
        Properties config = new Properties();

        // Long retention for regulatory compliance
```

```
        config.put("log.retention.hours", "26280"); // 3 years
        config.put("log.cleanup.policy", "delete"); // No compaction for audit
trail

        // Immutable segments for compliance
        config.put("log.segment.bytes", "536870912"); // 512MB
        config.put("log.roll.ms", "43200000"); // 12 hours

        // Preserve all data locally (no tiered storage)
        config.put("remote.storage.enable", "false");

        return config;
    }

    public static Properties getComplianceReplicationConfig() {
        Properties config = new Properties();

        // Maximum safety
        config.put("default.replication.factor", "5"); // Higher than usual
        config.put("min.insync.replicas", "3");
        config.put("unclean.leader.election.enable", "false"); // Never allow data
loss

        // Cross-datacenter replication
        config.put("replica.lag.time.max.ms", "60000"); // Allow for network
latency

        return config;
    }
}
```

Version Highlights

Kafka Storage Evolution

Version	Release Date	Storage Features
4.0	September 2025	Enhanced tiered storage, KRaft-only
3.6	October 2023	**Tiered storage (Early Access)**
3.3	October 2022	**KRaft production ready**
3.0	September 2021	KRaft preview, better log management
2.8	April 2021	Enhanced exactly-once semantics
2.4	December 2019	Improved log compaction
2.0	July 2018	Better retention handling
1.0	October 2017	Exactly-once semantics

Version	Release Date	Storage Features
0.11	June 2017	**Exactly-once delivery**
0.10	May 2016	**Log compaction improvements**

Replication Milestones

Version	Features
4.0	Enhanced ISR management, better leader election
3.3	KRaft replication protocol
2.8	**Incremental cooperative rebalancing**
2.1	Replica placement improvements
1.1	**Improved exactly-once replication**
0.9	**New consumer with rebalancing**
0.8	**Replication protocol introduced**

Metadata & Consensus Timeline

Version	Milestone
4.0	**ZooKeeper removal (KRaft only)**
3.3	**KRaft production ready**
2.8	KRaft early access
2.4	**KRaft development begins (KIP-500)**
0.7	**ZooKeeper integration**

Current Recommendations (2025)

```
// Modern Kafka internals configuration (4.0+)
public static Properties modernKafkaConfig() {
    Properties config = new Properties();

    // KRaft-only configuration (ZooKeeper removed)
    config.put("process.roles", "broker");
    config.put("controller.quorum.voters",
"1@controller1:9093,2@controller2:9093,3@controller3:9093");

    // Enhanced storage with tiered support
    config.put("remote.storage.enable", "true");
    config.put("local.retention.hours", "24");
    config.put("retention.hours", "8760"); // 1 year total
```

```
        // Optimized replication
        config.put("default.replication.factor", "3");
        config.put("min.insync.replicas", "2");
        config.put("replica.lag.time.max.ms", "30000");

        // Modern segment sizing
        config.put("log.segment.bytes", "1073741824"); // 1GB
        config.put("log.retention.check.interval.ms", "300000");

        return config;
    }
```

🔗 Additional Resources

📚 Official Documentation

- Kafka Storage Internals
- Replication Design
- KRaft Overview

🎓 Deep Dive Resources

- Kafka Log Compaction Guide
- Tiered Storage Architecture
- KRaft Protocol Details

🔧 Performance Tuning

- Kafka Performance Tuning
- Storage Optimization Guide
- Replication Performance

📊 Monitoring & Operations

- JMX Metrics Reference
- Storage Monitoring
- Replication Monitoring

🐛 Troubleshooting

- Common Storage Issues
- Replication Troubleshooting
- KRaft Migration Guide

Last Updated: September 2025
Kafka Version: 4.0.0
Architecture: KRaft-native (ZooKeeper removed)

> **Pro Tip**: Modern Kafka internals are optimized for cloud-native deployments. Use KRaft for new clusters, implement tiered storage for cost efficiency, and monitor ISR health closely. Understanding these internals is crucial for optimizing performance and troubleshooting issues in production environments.