Kafka Producers: Complete Developer Guide

A comprehensive refresher on Apache Kafka Producers, designed for both beginners and experienced developers. This README covers producer basics, partitioning strategies, reliability features, and real-world applications with detailed Java examples.

Table of Contents

Producer Basics

KafkaProducer API

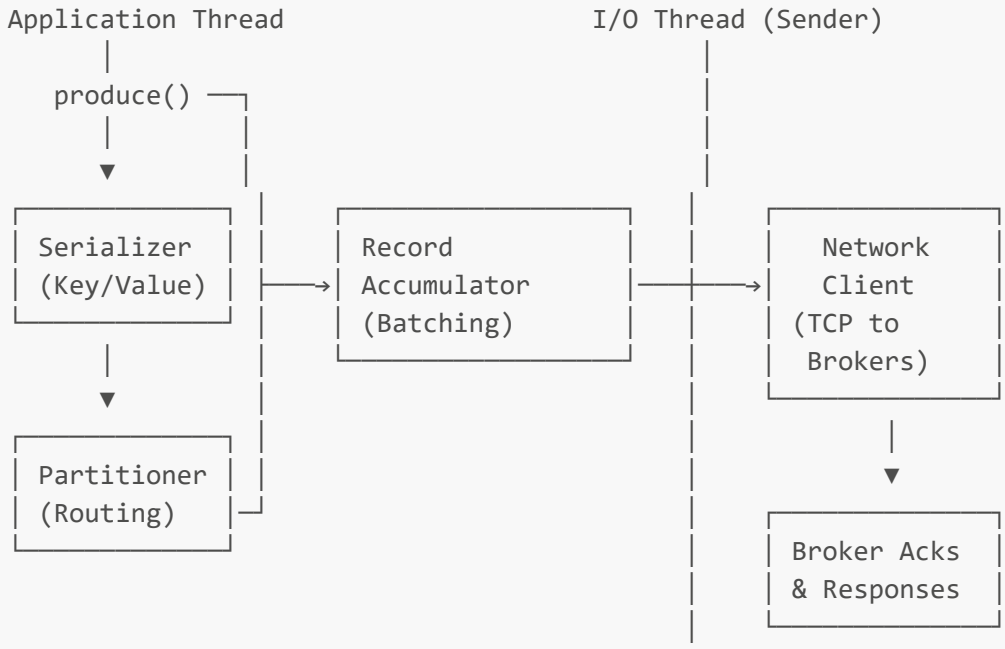
Simple Explanation

The KafkaProducer is the client API that applications use to publish records to Kafka topics. It's thread-safe and designed for high throughput with batching and asynchronous operations.

Problem It Solves

- **High Throughput**: Batches multiple messages for efficient network utilization
- **Fault Tolerance**: Handles retries and failures automatically
- **Scalability**: Distributes messages across topic partitions
- **Performance**: Asynchronous operations don't block application threads

Internal Architecture



Core Producer Configuration

```
Properties props = new Properties();

// Connection settings
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
props.put(ProducerConfig.CLIENT_ID_CONFIG, "my-producer");

// Serialization
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.StringSerializer");
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.StringSerializer");

// Performance & Batching
props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);        // 16KB batches
props.put(ProducerConfig.LINGER_MS_CONFIG, 5);             // Wait 5ms for batch
props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "gzip"); // Compress batches

// Reliability
props.put(ProducerConfig.ACKS_CONFIG, "all");              // Wait for all
replicas
props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE); // Retry forever
props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);  // Prevent duplicates
```

Asynchronous vs Synchronous Send

Asynchronous Send (Recommended)

```
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;
import java.util.Properties;
import java.util.concurrent.Future;

public class AsyncProducerExample {
    private static final String TOPIC = "user-events";

    public static void main(String[] args) {
        Properties props = createProducerConfig();

        try (KafkaProducer<String, String> producer = new KafkaProducer<>(props))
{

            for (int i = 0; i < 1000; i++) {
                String key = "user-" + i;
                String value = "login-event-" + System.currentTimeMillis();

                ProducerRecord<String, String> record =
                    new ProducerRecord<>(TOPIC, key, value);

                // Asynchronous send with callback
                producer.send(record, new Callback() {
                    @Override
                    public void onCompletion(RecordMetadata metadata, Exception
exception) {
                        if (exception != null) {
                            System.err.println("Failed to send record: " +
exception.getMessage());
                            // Implement retry logic or dead letter queue
                        } else {
                            System.out.printf("Sent record to partition %d, offset
%d, timestamp %d%n",
                                metadata.partition(), metadata.offset(),
metadata.timestamp());
                        }
                    }
                });

                // Don't block - continue sending more records
            }

            // Ensure all records are sent before closing
            producer.flush();

        } catch (Exception e) {
            System.err.println("Producer failed: " + e.getMessage());
        }
    }

    private static Properties createProducerConfig() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
```

```
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

        // High-throughput settings
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
        props.put(ProducerConfig.LINGER_MS_CONFIG, 5);
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "gzip");
        props.put(ProducerConfig.ACKS_CONFIG, "1");

        return props;
    }
}
```

Advantages:

- ☑ **High Throughput**: Non-blocking sends allow batching
- ☑ **Better Resource Utilization**: Application thread doesn't wait
- ☑ **Scalability**: Can send thousands of messages per second

Trade-offs:

- ⚠ **Complexity**: Need to handle callbacks for error handling
- ⚠ **Memory Usage**: Messages buffered in memory until sent

Synchronous Send (Use Sparingly)

```
public class SyncProducerExample {
    public static void main(String[] args) {
        Properties props = createProducerConfig();

        try (KafkaProducer<String, String> producer = new KafkaProducer<>(props))
{

            for (int i = 0; i < 10; i++) {
                ProducerRecord<String, String> record =
                    new ProducerRecord<>("critical-events", "key-" + i, "critical-
data-" + i);

                try {
                    // Synchronous send - blocks until complete
                    RecordMetadata metadata = producer.send(record).get();
                    System.out.printf("Sent record synchronously to partition %d,
offset %d%n",
                        metadata.partition(), metadata.offset());

                } catch (Exception e) {
                    System.err.println("Failed to send record synchronously: " +
e.getMessage());
                    // Handle error immediately
```

```
                    break;
                }
            }
        }
    }
}
```

When to Use:

- 🎯 **Critical Messages**: When you need immediate confirmation
- 🎯 **Error Handling**: When you must handle failures immediately
- 🎯 **Ordering**: When strict order is required (with max.in.flight.requests=1)

Trade-offs:

- ❌ **Low Throughput**: Blocks on every send
- ❌ **Poor Performance**: No batching benefits
- ❌ **Latency**: Application waits for network round trips

ProducerConfig Deep Dive

Core Configuration Parameters

```
public class ProducerConfigExample {
    public static Properties createProductionConfig() {
        Properties props = new Properties();

        // === CONNECTION SETTINGS ===
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "broker1:9092,broker2:9092,broker3:9092");
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "payment-service-producer");

        // === SERIALIZATION ===
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.connect.json.JsonSerializer"); // For JSON payloads

        // === RELIABILITY SETTINGS ===
        props.put(ProducerConfig.ACKS_CONFIG, "all");              // Wait for all
in-sync replicas
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE); // Retry
indefinitely
        props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 100);     // Wait 100ms
between retries
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000); // 30s request
timeout
        props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, 120000); // 2min
total timeout

        // === IDEMPOTENCE & ORDERING ===
```

```
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);  // Prevent
duplicates
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5); //
Default with idempotence

        // === BATCHING & PERFORMANCE ===
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);          // 32KB
batches
        props.put(ProducerConfig.LINGER_MS_CONFIG, 10);              // Wait 10ms
to fill batch
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "gzip");  // Compress
data
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 67108864);    // 64MB total
buffer

        // === NETWORK SETTINGS ===
        props.put(ProducerConfig.SEND_BUFFER_CONFIG, 131072);        // 128KB TCP
send buffer
        props.put(ProducerConfig.RECEIVE_BUFFER_CONFIG, 65536);      // 64KB TCP
receive buffer

        return props;
    }
}
```

Configuration Trade-offs

Parameter	Higher Value	Lower Value
batch.size	↑ Throughput, ↑ Latency, ↑ Memory	↓ Latency, ↓ Throughput, ↓ Memory
linger.ms	↑ Batching efficiency, ↑ Latency	↓ Latency, ↓ Batching
acks	↑ Durability, ↓ Throughput	↓ Durability, ↑ Throughput
retries	↑ Reliability, ↑ Latency	↓ Reliability, ↓ Latency
buffer.memory	↑ Buffering capacity, ↑ Memory usage	↓ Memory, may block sends

Environment-Specific Configurations

```
public class EnvironmentConfigs {

    // Development - Low latency, simple debugging
    public static Properties developmentConfig() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.ACKS_CONFIG, "1");                    // Leader
only
        props.put(ProducerConfig.RETRIES_CONFIG, 3);                   // Limited
retries
        props.put(ProducerConfig.LINGER_MS_CONFIG, 0);                 // No
```

```
batching delay
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, false);    //
Simplicity
        return props;
    }

    // Production - High reliability and performance
    public static Properties productionConfig() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "prod-broker1:9092,prod-broker2:9092,prod-broker3:9092");
        props.put(ProducerConfig.ACKS_CONFIG, "all");                   // Maximum
durability
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);    // Retry
forever
        props.put(ProducerConfig.LINGER_MS_CONFIG, 10);                 // Optimize
batching
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);             // Larger
batches
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "gzip");      // Reduce
bandwidth
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);      // Exactly-
once
        return props;
    }

    // High-throughput - Maximum performance
    public static Properties highThroughputConfig() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.ACKS_CONFIG, "0");                     // Fire and
forget
        props.put(ProducerConfig.LINGER_MS_CONFIG, 100);                //
Aggressive batching
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 65536);             // 64KB
batches
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4");       // Fast
compression
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 134217728);      // 128MB
buffer
        return props;
    }
}
```

Partitioning & Ordering

Default Partitioner

How Default Partitioner Works

```
Partitioning Logic:

1. If record has a key:
   partition = hash(key) % num_partitions

2. If record has no key:
   partition = round-robin (sticky partitioner in Kafka 2.4+)

Examples:
Topic: "orders" (3 partitions)

Key-based:
"user123" → hash("user123") % 3 = 1 → Partition 1
"user456" → hash("user456") % 3 = 0 → Partition 0
"user789" → hash("user789") % 3 = 2 → Partition 2

No key:
null → Round-robin → Partition 0, 1, 2, 0, 1, 2...
```

Default Partitioner Java Example

```
import org.apache.kafka.clients.producer.*;
import java.util.Properties;

public class DefaultPartitionerExample {

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        try (KafkaProducer<String, String> producer = new KafkaProducer<>(props))
{

            // Key-based partitioning - all messages for same user go to same
partition
            sendWithKey(producer, "user123", "User 123 placed order #1");
            sendWithKey(producer, "user123", "User 123 placed order #2");  // Same
partition
            sendWithKey(producer, "user456", "User 456 placed order #1");  //
Different partition

            // Round-robin partitioning - no key
            sendWithoutKey(producer, "Anonymous order #1");
            sendWithoutKey(producer, "Anonymous order #2");
            sendWithoutKey(producer, "Anonymous order #3");
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void sendWithKey(KafkaProducer<String, String> producer,
                                    String key, String value) {
        ProducerRecord<String, String> record =
            new ProducerRecord<>("orders", key, value);

        producer.send(record, (metadata, exception) -> {
            if (exception == null) {
                System.out.printf("Key: %s → Partition: %d, Offset: %d%n",
                    key, metadata.partition(), metadata.offset());
            }
        });
    }

    private static void sendWithoutKey(KafkaProducer<String, String> producer,
String value) {
        ProducerRecord<String, String> record =
            new ProducerRecord<>("orders", null, value);

        producer.send(record, (metadata, exception) -> {
            if (exception == null) {
                System.out.printf("No key → Partition: %d, Offset: %d%n",
                    metadata.partition(), metadata.offset());
            }
        });
    }
}
```

Sticky Partitioner (Kafka 2.4+)

Problem with Old Round-Robin:

- Poor batching efficiency when no key provided
- Each message could go to different partition, reducing batch effectiveness

Sticky Partitioner Solution:

```
// Old behavior (pre-2.4):
// msg1 → partition 0, msg2 → partition 1, msg3 → partition 2, msg4 → partition
0...

// Sticky partitioner (2.4+):
// msg1-10 → partition 0, msg11-20 → partition 1, msg21-30 → partition 2...
// Sticks to partition until batch is sent, then switches
```

Custom Partitioner

When to Use Custom Partitioner

- 🎯 **Business Logic**: Route messages based on business rules
- 🎯 **Load Balancing**: Custom distribution strategies
- 🎯 **Data Locality**: Keep related data together
- 🎯 **Performance**: Optimize for specific access patterns

Custom Partitioner Implementation

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import java.util.List;
import java.util.Map;

// Route VIP customers to specific partition for priority processing
public class VipCustomerPartitioner implements Partitioner {

    private static final int VIP_PARTITION = 0;  // Dedicate partition 0 for VIP
customers

    @Override
    public int partition(String topic, Object key, byte[] keyBytes,
                         Object value, byte[] valueBytes, Cluster cluster) {

        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();

        if (key == null) {
            // No key - use round-robin for remaining partitions
            return (int) (Math.random() * (numPartitions - 1)) + 1;
        }

        String keyStr = key.toString();

        // VIP customers go to partition 0
        if (keyStr.startsWith("vip-")) {
            return VIP_PARTITION;
        }

        // Premium customers get better partitions (1-2)
        if (keyStr.startsWith("premium-")) {
            return (keyStr.hashCode() & Integer.MAX_VALUE) % 2 + 1;
        }

        // Regular customers use remaining partitions (3+)
        if (numPartitions > 3) {
            return (keyStr.hashCode() & Integer.MAX_VALUE) % (numPartitions - 3) +
3;
        }

        // Fallback to default behavior
```

```
        return (keyStr.hashCode() & Integer.MAX_VALUE) % numPartitions;
    }

    @Override
    public void close() {
        // Cleanup resources if needed
    }

    @Override
    public void configure(Map<String, ?> configs) {
        // Configuration if needed
    }
}

// Geographic partitioner - route by region
public class GeographicPartitioner implements Partitioner {

    @Override
    public int partition(String topic, Object key, byte[] keyBytes,
                         Object value, byte[] valueBytes, Cluster cluster) {

        if (key == null) {
            return 0; // Default partition
        }

        String keyStr = key.toString();
        int numPartitions = cluster.partitionsForTopic(topic).size();

        // Extract region from key (format: "region-user-id")
        String[] parts = keyStr.split("-");
        if (parts.length >= 2) {
            String region = parts[0].toLowerCase();

            switch (region) {
                case "us": return 0;
                case "eu": return 1;
                case "asia": return 2;
                case "latam": return 3;
                default:
                    return (keyStr.hashCode() & Integer.MAX_VALUE) %
numPartitions;
            }
        }

        return (keyStr.hashCode() & Integer.MAX_VALUE) % numPartitions;
    }

    @Override
    public void close() {}

    @Override
    public void configure(Map<String, ?> configs) {}
}
```

Using Custom Partitioner

```
public class CustomPartitionerExample {

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        // Configure custom partitioner
        props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
            "com.example.VipCustomerPartitioner");

        try (KafkaProducer<String, String> producer = new KafkaProducer<>(props))
{

            // VIP customer - goes to partition 0
            sendMessage(producer, "vip-customer-001", "VIP order: $10000");

            // Premium customer - goes to partition 1-2
            sendMessage(producer, "premium-customer-123", "Premium order: $1000");

            // Regular customer - goes to partition 3+
            sendMessage(producer, "regular-customer-456", "Regular order: $100");

            // Geographic routing
            sendMessage(producer, "us-customer-789", "US order");
            sendMessage(producer, "eu-customer-321", "European order");
            sendMessage(producer, "asia-customer-654", "Asian order");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void sendMessage(KafkaProducer<String, String> producer,
                                    String key, String value) {
        ProducerRecord<String, String> record =
            new ProducerRecord<>("customer-orders", key, value);

        producer.send(record, (metadata, exception) -> {
            if (exception == null) {
                System.out.printf("Key: %s → Partition: %d (Custom Logic)%n",
                    key, metadata.partition());
            } else {
                System.err.println("Failed to send: " + exception.getMessage());
            }
        });
```

```
    }
}
```

Ordering Guarantees

Single Partition Ordering

```
Single Partition = Total Order

Partition 0:
┌─────┬─────┬─────┬─────┬─────┐
│ M1  │ M2  │ M3  │ M4  │ M5  │
└─────┴─────┴─────┴─────┴─────┘
  ↑                       ↑
offset 0               offset 4

Consumer reads: M1, M2, M3, M4, M5 (always same order)
```

```
// Ensuring single partition for ordering
public class OrderedMessagesExample {

    public static void main(String[] args) {
        Properties props = createProducerConfig();

        try (KafkaProducer<String, String> producer = new KafkaProducer<>(props))
{

            String userId = "user123";  // Same key = same partition

            // These messages will be ordered within the partition
            sendOrderedMessage(producer, userId, "User created account");
            sendOrderedMessage(producer, userId, "User verified email");
            sendOrderedMessage(producer, userId, "User updated profile");
            sendOrderedMessage(producer, userId, "User made first purchase");

            producer.flush(); // Ensure all sent

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void sendOrderedMessage(KafkaProducer<String, String> producer,
                                           String userId, String event) {
        ProducerRecord<String, String> record =
            new ProducerRecord<>("user-events", userId, event);

        // For strict ordering, consider using synchronous send
        try {
```

```
            RecordMetadata metadata = producer.send(record).get();
            System.out.printf("Ordered event for %s: %s → Partition %d, Offset
%d%n",
                userId, event, metadata.partition(), metadata.offset());
        } catch (Exception e) {
            System.err.println("Failed to send ordered message: " +
e.getMessage());
        }
    }

    private static Properties createProducerConfig() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        // Settings for strict ordering
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 1);
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
        props.put(ProducerConfig.ACKS_CONFIG, "all");

        return props;
    }
}
```

Multiple Partitions - No Global Order

```
Multiple Partitions = No Global Order

Partition 0: [M1, M3, M5] ← User A events
Partition 1: [M2, M4, M6] ← User B events

Consumer Group:
Consumer 1 reads P0: M1, M3, M5
Consumer 2 reads P1: M2, M4, M6

Global consumption order could be: M1, M2, M3, M4, M5, M6
                                OR: M2, M1, M4, M3, M6, M5
                                OR: Any interleaving!
```

Ordering Configuration Trade-offs

Configuration	Ordering Guarantee	Performance Impact
`max.in.flight.requests=1`	☑ Strict per partition	❌ Lower throughput

Configuration	Ordering Guarantee	Performance Impact
`max.in.flight.requests=5` + `enable.idempotence=true`	☑ Order preserved with retries	★ Good throughput
`max.in.flight.requests=5` + `enable.idempotence=false`	✗ May reorder on retry	★★ Higher throughput

🛡 Reliability

Idempotent Producer

Problem It Solves

Producer Retry Duplication:

```
Without Idempotence:
1. Producer sends message M1
2. Broker receives M1, but ACK is lost
3. Producer retries, sends M1 again
4. Result: M1 appears twice in log

With Idempotence:
1. Producer sends M1 with sequence number
2. Broker receives M1, but ACK is lost
3. Producer retries M1 with same sequence number
4. Broker recognizes duplicate, ignores retry
5. Result: M1 appears once in log
```

Internal Mechanism

```
Idempotent Producer Internals:

Producer ID (PID): Unique producer identifier
Sequence Number: Per-partition counter starting from 0

Message Format:
┌──────────┬──────────────┬──────────────┬─────────────────┐
│ PID      │ Epoch        │ Sequence     │ Message         │
│ 12345    │ 0            │ 0            │ "Hello"         │
└──────────┴──────────────┴──────────────┴─────────────────┘

Broker State:
Partition 0: Last seen (PID=12345, Epoch=0, Seq=2)
Partition 1: Last seen (PID=12345, Epoch=0, Seq=5)
```

Idempotent Producer Java Example

```
import org.apache.kafka.clients.producer.*;
import java.util.Properties;
import java.util.concurrent.ExecutionException;

public class IdempotentProducerExample {

    public static void main(String[] args) {
        Properties props = createIdempotentProducerConfig();

        try (KafkaProducer<String, String> producer = new KafkaProducer<>(props))
{

            // Simulate sending critical financial transactions
            for (int i = 0; i < 10; i++) {
                String transactionId = "txn-" + i;
                String transactionData = String.format(
                    "Transfer $%.2f from account A to account B",
                    1000.0 + (i * 100));

                sendIdempotentMessage(producer, transactionId, transactionData);
            }

            // Force any buffered messages to be sent
            producer.flush();

        } catch (Exception e) {
            System.err.println("Idempotent producer failed: " + e.getMessage());
            e.printStackTrace();
        }
    }

    private static void sendIdempotentMessage(KafkaProducer<String, String>
producer,
                                              String key, String value) {
        ProducerRecord<String, String> record =
            new ProducerRecord<>("financial-transactions", key, value);

        producer.send(record, new Callback() {
            @Override
            public void onCompletion(RecordMetadata metadata, Exception exception)
{
                if (exception != null) {
                    if (exception instanceof
org.apache.kafka.common.errors.OutOfOrderSequenceException) {
                        System.err.println("Out of order sequence - this shouldn't
happen with idempotence!");
                    } else if (exception instanceof
org.apache.kafka.common.errors.DuplicateSequenceNumberException) {
                        System.out.println("Duplicate detected and handled by
broker");
```

```
                    } else {
                        System.err.println("Send failed: " +
exception.getMessage());
                    }
                } else {
                    System.out.printf("Idempotent send: %s → P%d:O%d%n",
                        key, metadata.partition(), metadata.offset());
                }
            }
        });
    }

    private static Properties createIdempotentProducerConfig() {
        Properties props = new Properties();

        // Basic configuration
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        // Idempotence configuration (Kafka 3.0+ defaults)
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);

        // Required settings for idempotence
        props.put(ProducerConfig.ACKS_CONFIG, "all");               // Must wait
for all replicas
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE); // Must
enable retries
        props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 5); // Max
5 for idempotence

        // Additional reliability
        props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 100);
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000);

        return props;
    }
}
```

Idempotence Limitations & Considerations

```
public class IdempotenceLimitations {

    public void demonstrateLimitations() {
        // 1. Producer restart loses PID - can't prevent duplicates across
restarts
        System.out.println("Limitation 1: Producer restart breaks idempotence");

        // 2. Only works within producer session
```

```
        System.out.println("Limitation 2: Only prevents retries within same
producer instance");

        // 3. Only prevents duplicates from producer retries, not application
retries
        System.out.println("Limitation 3: Application-level retries can still
create duplicates");
    }

    // For cross-session idempotence, use transactional producer
    public Properties transactionalConfig() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        // Transactional settings
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "payment-processor-1");
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true); //
Automatically true
        props.put(ProducerConfig.ACKS_CONFIG, "all");

        return props;
    }
}
```

Transactional Producer

Problem It Solves

- **Exactly-Once Processing**: Guaranteed exactly-once semantics across producer restarts
- **Atomic Multi-Topic Writes**: All messages in transaction committed together
- **Read Committed**: Consumers only see committed messages

Transactional Producer Architecture

```
Transactional Producer Flow:

1. Producer gets Transaction Coordinator assignment
2. Producer begins transaction
3. Producer sends messages to multiple topics/partitions
4. Producer commits transaction
5. Transaction Coordinator marks all messages as committed
6. Consumers with isolation.level=read_committed see messages

Transaction Log:
┌──────────────┬──────────────┬──────────────┬──────────────┐
│ TXN_ID       │ Status       │ Partitions   │ Timestamp    │
```

```
├──────────────┼──────────────┼──────────────┼──────────────┤
│ payment-1    │ ONGOING      │ orders:0     │ 1693875600   │
│              │              │ inventory:1  │              │
│ payment-1    │ COMMITTED    │ orders:0     │ 1693875605   │
│              │              │ inventory:1  │              │
└──────────────┴──────────────┴──────────────┴──────────────┘
```

Comprehensive Transactional Producer Example

```
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.errors.ProducerFencedException;
import org.apache.kafka.common.errors.OutOfOrderSequenceException;
import org.apache.kafka.common.errors.AuthorizationException;
import java.util.Properties;

public class TransactionalProducerExample {

    private static final String ORDERS_TOPIC = "orders";
    private static final String INVENTORY_TOPIC = "inventory";
    private static final String PAYMENTS_TOPIC = "payments";

    public static void main(String[] args) {
        Properties props = createTransactionalProducerConfig();

        KafkaProducer<String, String> producer = new KafkaProducer<>(props);

        // Initialize transactions
        producer.initTransactions();

        try {
            // Process multiple orders atomically
            for (int i = 0; i < 5; i++) {
                processOrderTransactionally(producer, "order-" + i, i * 100.0);
            }

        } catch (Exception e) {
            System.err.println("Transactional processing failed: " +
e.getMessage());
        } finally {
            producer.close();
        }
    }

    private static void processOrderTransactionally(KafkaProducer<String, String>
producer,
                                                    String orderId, double amount)
{
        try {
            // Begin transaction
            producer.beginTransaction();
```

```
            // Send order creation event
            ProducerRecord<String, String> orderRecord = new ProducerRecord<>(
                ORDERS_TOPIC, orderId,
                String.format("Order created: %s, amount: $%.2f", orderId,
amount));
            producer.send(orderRecord);

            // Reserve inventory
            ProducerRecord<String, String> inventoryRecord = new ProducerRecord<>(
                INVENTORY_TOPIC, orderId,
                String.format("Inventory reserved for order: %s", orderId));
            producer.send(inventoryRecord);

            // Process payment
            ProducerRecord<String, String> paymentRecord = new ProducerRecord<>(
                PAYMENTS_TOPIC, orderId,
                String.format("Payment processed: %s, amount: $%.2f", orderId,
amount));
            producer.send(paymentRecord);

            // Simulate business logic that might fail
            if (orderId.equals("order-3")) {
                throw new RuntimeException("Payment processor temporarily
unavailable");
            }

            // Commit transaction - all messages become visible atomically
            producer.commitTransaction();
            System.out.println("Transaction committed successfully for " +
orderId);

        } catch (ProducerFencedException e) {
            // Another producer with same transactional.id is active
            System.err.println("Producer fenced: " + e.getMessage());
            producer.close();
        } catch (OutOfOrderSequenceException | AuthorizationException e) {
            // Unrecoverable errors
            System.err.println("Unrecoverable error: " + e.getMessage());
            producer.close();
        } catch (Exception e) {
            // Abort transaction on any error
            System.err.println("Transaction failed for " + orderId + ": " +
e.getMessage());
            try {
                producer.abortTransaction();
                System.out.println("Transaction aborted for " + orderId);
            } catch (Exception abortEx) {
                System.err.println("Failed to abort transaction: " +
abortEx.getMessage());
            }
        }
    }

    private static Properties createTransactionalProducerConfig() {
```

```
        Properties props = new Properties();

        // Basic configuration
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        // Transactional configuration
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "order-processor-1");

        // These are automatically set with transactional.id
        // props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
        // props.put(ProducerConfig.ACKS_CONFIG, "all");
        // props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);

        // Performance settings
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384);
        props.put(ProducerConfig.LINGER_MS_CONFIG, 5);

        return props;
    }
}
```

Transactional Consumer (Read Committed)

```
import org.apache.kafka.clients.consumer.*;
import java.time.Duration;
import java.util.Arrays;
import java.util.Properties;

public class TransactionalConsumerExample {

    public static void main(String[] args) {
        Properties props = createTransactionalConsumerConfig();

        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props))
{
            consumer.subscribe(Arrays.asList("orders", "inventory", "payments"));

            while (true) {
                ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(1000));

                for (ConsumerRecord<String, String> record : records) {
                    // Only see committed messages due to
isolation.level=read_committed
                    System.out.printf("Consumed committed record: topic=%s,
key=%s, value=%s, " +
                        "partition=%d, offset=%d%n",
```

```
                        record.topic(), record.key(), record.value(),
                        record.partition(), record.offset());
                }

                // Commit offsets
                consumer.commitSync();
            }
        }
    }

    private static Properties createTransactionalConsumerConfig() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "transactional-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringDeserializer");

        // Only read committed messages (default is read_uncommitted)
        props.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG, "read_committed");

        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

        return props;
    }
}
```

Delivery Semantics

At-Most-Once (Fire and Forget)

```
public class AtMostOnceProducer {

    public static Properties createConfig() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        // At-most-once settings - fire and forget
        props.put(ProducerConfig.ACKS_CONFIG, "0");           // Don't wait for
ack
        props.put(ProducerConfig.RETRIES_CONFIG, 0);          // No retries
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, false);

        return props;
    }
```

```
    public static void main(String[] args) {
        Properties props = createConfig();

        try (KafkaProducer<String, String> producer = new KafkaProducer<>(props))
{
            for (int i = 0; i < 100; i++) {
                ProducerRecord<String, String> record =
                    new ProducerRecord<>("metrics", "sensor-" + i, "temperature:"
+ (20 + i));

                // Fire and forget - don't care about result
                producer.send(record);
            }
        }

        System.out.println("Messages sent (at-most-once) - some may be lost, none
duplicated");
    }
}
```

At-Least-Once (Standard Retry)

```
public class AtLeastOnceProducer {

    public static Properties createConfig() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        // At-least-once settings
        props.put(ProducerConfig.ACKS_CONFIG, "all");         // Wait for all
replicas
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE); // Retry on
failure
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, false); // Allow
duplicates

        return props;
    }

    public static void main(String[] args) {
        Properties props = createConfig();

        try (KafkaProducer<String, String> producer = new KafkaProducer<>(props))
{
            for (int i = 0; i < 100; i++) {
                ProducerRecord<String, String> record =
```

```
                    new ProducerRecord<>("events", "event-" + i, "data-" + i);

                producer.send(record, (metadata, exception) -> {
                    if (exception != null) {
                        System.err.println("Failed after retries: " +
exception.getMessage());
                    } else {
                        System.out.printf("Sent: partition=%d, offset=%d%n",
                            metadata.partition(), metadata.offset());
                    }
                });
            }
        }

        System.out.println("Messages sent (at-least-once) - none lost, duplicates
possible");
    }
}
```

Exactly-Once (Idempotent/Transactional)

```
public class ExactlyOnceProducer {

    // Exactly-once with idempotence (single producer session)
    public static Properties createIdempotentConfig() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        // Exactly-once within producer session
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
        // Other settings automatically configured: acks=all, retries=MAX,
max.in.flight=5

        return props;
    }

    // Exactly-once across restarts (transactional)
    public static Properties createTransactionalConfig() {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            "org.apache.kafka.common.serialization.StringSerializer");

        // Exactly-once across producer restarts
        props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "exactly-once-
```

```
producer");
        // Idempotence automatically enabled

        return props;
    }

    public static void main(String[] args) {
        // Example with transactional exactly-once
        Properties props = createTransactionalConfig();

        KafkaProducer<String, String> producer = new KafkaProducer<>(props);
        producer.initTransactions();

        try {
            producer.beginTransaction();

            for (int i = 0; i < 10; i++) {
                ProducerRecord<String, String> record =
                    new ProducerRecord<>("financial-events", "txn-" + i,
                        "amount:" + (1000.0 * i));
                producer.send(record);
            }

            producer.commitTransaction();
            System.out.println("Transaction committed - exactly-once guarantee");

        } catch (Exception e) {
            producer.abortTransaction();
            System.err.println("Transaction aborted: " + e.getMessage());
        } finally {
            producer.close();
        }
    }
}
```

Comprehensive Java Examples

Complete Producer Application

```
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.StringSerializer;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.core.JsonProcessingException;

import java.util.Properties;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;
```

```
/**
 * Production-ready Kafka Producer with error handling, metrics, and monitoring
 */
public class ProductionKafkaProducerExample {

    private final KafkaProducer<String, String> producer;
    private final ObjectMapper objectMapper;
    private final AtomicInteger successCount = new AtomicInteger(0);
    private final AtomicInteger errorCount = new AtomicInteger(0);

    public ProductionKafkaProducerExample(Properties config) {
        this.producer = new KafkaProducer<>(config);
        this.objectMapper = new ObjectMapper();
    }

    public static void main(String[] args) {
        Properties config = createProductionConfig();
        ProductionKafkaProducerExample producerApp =
            new ProductionKafkaProducerExample(config);

        // Simulate high-throughput application
        producerApp.runHighThroughputTest();

        // Cleanup
        producerApp.close();
    }

    private static Properties createProductionConfig() {
        Properties props = new Properties();

        // Cluster connection
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            "broker1:9092,broker2:9092,broker3:9092");
        props.put(ProducerConfig.CLIENT_ID_CONFIG, "order-service-producer-v1.0");

        // Serialization
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

        // Reliability - exactly-once
        props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
        props.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG, 100);

        // Performance optimization
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);      // 32KB batches
        props.put(ProducerConfig.LINGER_MS_CONFIG, 20);          // 20ms batching
window
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "gzip"); // Compression
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 67108864); // 64MB buffer
```

```
        // Timeouts
        props.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG, 30000);
        props.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG, 120000);

        return props;
    }

    public void runHighThroughputTest() {
        int numThreads = 4;
        int messagesPerThread = 1000;
        CountDownLatch latch = new CountDownLatch(numThreads);
        ExecutorService executor = Executors.newFixedThreadPool(numThreads);

        long startTime = System.currentTimeMillis();

        for (int t = 0; t < numThreads; t++) {
            final int threadId = t;
            executor.submit(() -> {
                try {
                    for (int i = 0; i < messagesPerThread; i++) {
                        sendOrderEvent(threadId, i);
                    }
                } finally {
                    latch.countDown();
                }
            });
        }

        try {
            latch.await(); // Wait for all threads to complete
            producer.flush(); // Ensure all messages are sent

            long duration = System.currentTimeMillis() - startTime;

            System.out.printf("Performance Results:%n");
            System.out.printf("Total messages: %d%n", numThreads *
messagesPerThread);
            System.out.printf("Successful: %d%n", successCount.get());
            System.out.printf("Errors: %d%n", errorCount.get());
            System.out.printf("Duration: %d ms%n", duration);
            System.out.printf("Throughput: %.2f messages/sec%n",
                (numThreads * messagesPerThread * 1000.0) / duration);

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            executor.shutdown();
        }
    }

    private void sendOrderEvent(int threadId, int messageId) {
        try {
            // Create order event object
            OrderEvent orderEvent = new OrderEvent(
```

```
                "order-" + threadId + "-" + messageId,
                "user-" + (messageId % 100),
                100.0 + (messageId % 500),
                System.currentTimeMillis()
            );

            String key = orderEvent.getUserId(); // Partition by user
            String value = objectMapper.writeValueAsString(orderEvent);

            ProducerRecord<String, String> record =
                new ProducerRecord<>("order-events", key, value);

            // Add headers for tracing/monitoring
            record.headers().add("source-service", "order-service".getBytes());
            record.headers().add("thread-id",
String.valueOf(threadId).getBytes());

            // Send asynchronously with callback
            producer.send(record, new Callback() {
                @Override
                public void onCompletion(RecordMetadata metadata, Exception
exception) {
                    if (exception == null) {
                        successCount.incrementAndGet();
                        if (messageId % 100 == 0) {
                            System.out.printf("Thread %d sent message %d to
partition %d%n",
                                threadId, messageId, metadata.partition());
                        }
                    } else {
                        errorCount.incrementAndGet();
                        System.err.printf("Thread %d failed to send message %d:
%s%n",
                            threadId, messageId, exception.getMessage());

                        // Handle different exception types
                        handleProducerException(exception, orderEvent);
                    }
                }
            });

        } catch (JsonProcessingException e) {
            System.err.println("JSON serialization error: " + e.getMessage());
            errorCount.incrementAndGet();
        }
    }

    private void handleProducerException(Exception exception, OrderEvent
orderEvent) {
        if (exception instanceof
org.apache.kafka.common.errors.RetriableException) {
            // Will be retried automatically by producer
            System.out.println("Retriable error - producer will retry: " +
exception.getMessage());
```

```
        } else if (exception instanceof
org.apache.kafka.common.errors.RecordTooLargeException) {
            // Message too large - need to handle specially
            System.err.println("Message too large, sending to DLQ: " +
orderEvent.getOrderId());
            // Send to dead letter queue or break into smaller messages
        } else {
            // Other non-retriable errors
            System.err.println("Non-retriable error: " + exception.getMessage());
            // Could send to DLQ or alert monitoring system
        }
    }

    public void close() {
        if (producer != null) {
            producer.close();
        }
    }

    // Data class for order events
    static class OrderEvent {
        private String orderId;
        private String userId;
        private double amount;
        private long timestamp;

        public OrderEvent(String orderId, String userId, double amount, long
timestamp) {
            this.orderId = orderId;
            this.userId = userId;
            this.amount = amount;
            this.timestamp = timestamp;
        }

        // Getters for JSON serialization
        public String getOrderId() { return orderId; }
        public String getUserId() { return userId; }
        public double getAmount() { return amount; }
        public long getTimestamp() { return timestamp; }
    }
}
```

Producer with Custom Serializer

```
import org.apache.kafka.common.serialization.Serializer;
import com.fasterxml.jackson.databind.ObjectMapper;
import java.util.Map;

// Custom JSON serializer for complex objects
public class JsonSerializer<T> implements Serializer<T> {
    private final ObjectMapper objectMapper = new ObjectMapper();
```

```
    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
        // Configuration if needed
    }

    @Override
    public byte[] serialize(String topic, T data) {
        try {
            return objectMapper.writeValueAsBytes(data);
        } catch (Exception e) {
            throw new RuntimeException("Error serializing JSON", e);
        }
    }

    @Override
    public void close() {
        // Cleanup if needed
    }
}

// Usage with custom serializer
public class CustomSerializerExample {

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);

        try (KafkaProducer<String, OrderEvent> producer = new KafkaProducer<>
(props)) {
            OrderEvent order = new OrderEvent("order-123", "user-456", 999.99,
System.currentTimeMillis());

            ProducerRecord<String, OrderEvent> record =
                new ProducerRecord<>("orders", order.getUserId(), order);

            producer.send(record, (metadata, exception) -> {
                if (exception == null) {
                    System.out.println("Custom serialized object sent
successfully");
                } else {
                    System.err.println("Failed to send: " +
exception.getMessage());
                }
            });
        }
    }
}
```

⚖️ Comparisons & Trade-offs

Delivery Semantics Comparison

Delivery Semantic	Configuration	Guarantees	Performance	Use Cases
At-Most-Once	`acks=0`, `retries=0`	Messages may be lost, never duplicated	★★★ Highest throughput	Metrics, logs, non-critical data
At-Least-Once	`acks=all`, `retries>0`, no idempotence	No data loss, duplicates possible	★★ Good performance	Event streaming, analytics
Exactly-Once	`enable.idempotence=true` or transactional	No loss, no duplicates	★ Higher latency	Financial systems, critical data

Producer Configuration Trade-offs

Parameter	Higher Value Impact	Lower Value Impact	Recommendation
`batch.size`	↑ Throughput, ↑ Latency, ↑ Memory	↓ Latency, ↓ Throughput	16KB-32KB for most use cases
`linger.ms`	↑ Batching, ↑ Latency	↓ Latency, ↓ Batching	5-20ms for balanced performance
`acks`	↑ Durability, ↓ Throughput	↓ Durability, ↑ Throughput	`all` for production systems
`compression.type`	↓ Bandwidth, ↑ CPU	↑ Bandwidth, ↓ CPU	`gzip` for bandwidth, `lz4` for speed

Partitioning Strategy Comparison

Strategy	Advantages	Disadvantages	When to Use
Key-based	Data locality, ordering per key	Uneven distribution if keys skewed	User events, entity updates
Round-robin	Even distribution	No data locality	Logs, metrics without keys
Custom	Business-specific optimization	Implementation complexity	Geographic routing, priority queues

🚨 Common Pitfalls & Best Practices

1. Configuration Mistakes

❌ Blocking the Application Thread

```
// DON'T - Synchronous sends hurt performance
for (int i = 0; i < 10000; i++) {
    try {
        producer.send(record).get(); // Blocks on every send!
    } catch (Exception e) {
        // Handle error
    }
}
```

```
// DO - Use async sends with callbacks
for (int i = 0; i < 10000; i++) {
    producer.send(record, (metadata, exception) -> {
        if (exception != null) {
            // Handle error asynchronously
            handleError(exception);
        }
    });
}
producer.flush(); // Wait for all sends to complete
```

❌ Ignoring Producer Exceptions

```
// DON'T - Fire and forget without error handling
producer.send(record);
```

```
// DO - Always handle exceptions
producer.send(record, new Callback() {
    @Override
    public void onCompletion(RecordMetadata metadata, Exception exception) {
        if (exception != null) {
            if (exception instanceof RetriableException) {
                // Will be retried automatically
                logger.warn("Retriable error: {}", exception.getMessage());
            } else {
                // Non-retriable - handle immediately
                logger.error("Failed to send message", exception);
                sendToDeadLetterQueue(record);
            }
        }
    }
});
```

2. Memory and Resource Issues

❌ Unbounded Producer Memory Usage

```
// DON'T - Can cause OutOfMemoryError
Properties props = new Properties();
props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, Long.MAX_VALUE); // Dangerous!
props.put(ProducerConfig.BATCH_SIZE_CONFIG, 1048576); // 1MB batches
props.put(ProducerConfig.LINGER_MS_CONFIG, 60000); // Wait 60 seconds
```

```
// DO - Set reasonable limits
Properties props = new Properties();
props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 67108864); // 64MB limit
props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16384); // 16KB batches
props.put(ProducerConfig.LINGER_MS_CONFIG, 100); // 100ms max wait
props.put(ProducerConfig.MAX_BLOCK_MS_CONFIG, 5000); // Block max 5s when buffer
full
```

❌ Producer Resource Leaks

```
// DON'T - Forget to close producer
public void sendMessage(String message) {
    KafkaProducer<String, String> producer = new KafkaProducer<>(props);
    producer.send(new ProducerRecord<>("topic", message));
    // Producer not closed - resource leak!
}
```

```
// DO - Use try-with-resources or proper cleanup
private final KafkaProducer<String, String> producer; // Shared instance

public void sendMessage(String message) {
    producer.send(new ProducerRecord<>("topic", message));
}

@PreDestroy
public void cleanup() {
    if (producer != null) {
        producer.close(Duration.ofSeconds(10)); // Graceful close
    }
}
```

3. Ordering and Idempotence Issues

❌ Losing Message Order on Retries

```
// DON'T - Can reorder messages on retry
Properties props = new Properties();
props.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, 10);
props.put(ProducerConfig.RETRIES_CONFIG, 3);
props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, false);
```

```
// DO - Enable idempotence for ordering guarantees
Properties props = new Properties();
props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true);
// Automatically sets: acks=all, retries=MAX_VALUE, max.in.flight.requests=5
```

4. Performance Anti-patterns

❌ Creating New Producer Per Message

```
// DON'T - Very expensive
public void sendMessage(String message) {
    Properties props = createProducerConfig();
    try (KafkaProducer<String, String> producer = new KafkaProducer<>(props)) {
        producer.send(new ProducerRecord<>("topic", message));
    } // Producer closed after every message!
}
```

```
// DO - Reuse producer instances
@Component
public class MessageService {
    private final KafkaProducer<String, String> producer;

    public MessageService() {
        this.producer = new KafkaProducer<>(createProducerConfig());
    }

    public void sendMessage(String message) {
        producer.send(new ProducerRecord<>("topic", message));
    }

    @PreDestroy
    public void close() {
        producer.close();
    }
}
```

Best Practices Summary

☑ Producer Configuration Best Practices

1. **Enable idempotence** by default (`enable.idempotence=true`)
2. **Use appropriate** `acks` setting (`all` for production)
3. **Set reasonable timeouts** (`request.timeout.ms=30000`)
4. **Configure proper batching** (`batch.size=16384`, `linger.ms=5-20`)
5. **Use compression** (`compression.type=gzip` or `lz4`)

☑ Application Best Practices

1. **Reuse producer instances** - they're thread-safe
2. **Handle exceptions properly** - distinguish retriable vs non-retriable
3. **Use async sends** - don't block application threads
4. **Monitor metrics** - track success rate, latency, errors
5. **Implement circuit breakers** - handle prolonged failures gracefully

☑ Operational Best Practices

1. **Monitor producer metrics** - `record-send-rate`, `record-error-rate`
2. **Set up alerting** - on error rates, buffer exhaustion
3. **Plan for backpressure** - handle when Kafka is slow/down
4. **Test failure scenarios** - broker failures, network partitions
5. **Capacity planning** - monitor throughput trends

🌍 Real-World Use Cases

1. E-commerce Order Processing

```
@Service
public class OrderService {
    private final KafkaProducer<String, OrderEvent> producer;

    @Transactional
    public void processOrder(Order order) {
        // Save to database
        orderRepository.save(order);

        // Publish event for downstream services
        OrderEvent event = new OrderEvent(order.getId(), order.getUserId(),
            order.getAmount(), OrderStatus.CREATED);

        ProducerRecord<String, OrderEvent> record =
            new ProducerRecord<>("order-events", order.getUserId(), event);

        producer.send(record, (metadata, exception) -> {
            if (exception != null) {
                // Compensating action or alert
```

```
                logger.error("Failed to publish order event", exception);
                orderEventFailureHandler.handle(order, exception);
            }
        });
    }
}
```

Why Kafka for Orders:

- **Decoupling**: Order service doesn't depend on inventory, payment services
- **Reliability**: Events are persisted and can be replayed
- **Scalability**: Multiple consumers can handle different aspects

2. Real-time Analytics and Monitoring

```
@Component
public class MetricsProducer {
    private final KafkaProducer<String, MetricEvent> producer;

    @Scheduled(fixedDelay = 1000) // Every second
    public void publishSystemMetrics() {
        // CPU usage
        double cpuUsage = systemMonitor.getCpuUsage();
        publishMetric("system.cpu.usage", cpuUsage, "percentage");

        // Memory usage
        long memoryUsed = systemMonitor.getMemoryUsed();
        publishMetric("system.memory.used", memoryUsed, "bytes");

        // Request rate
        double requestRate = applicationMonitor.getRequestRate();
        publishMetric("app.requests.rate", requestRate, "requests/sec");
    }

    private void publishMetric(String metricName, double value, String unit) {
        MetricEvent event = new MetricEvent(metricName, value, unit,
            System.currentTimeMillis(), "hostname");

        // Partition by metric name for consistent routing
        ProducerRecord<String, MetricEvent> record =
            new ProducerRecord<>("system-metrics", metricName, event);

        producer.send(record); // Fire and forget for metrics
    }
}
```

Configuration for High-Volume Metrics:

```
Properties metricsProducerConfig() {
    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");

    // Optimize for throughput over reliability
    props.put(ProducerConfig.ACKS_CONFIG, "1"); // Leader ack only
    props.put(ProducerConfig.BATCH_SIZE_CONFIG, 65536); // 64KB batches
    props.put(ProducerConfig.LINGER_MS_CONFIG, 100); // Wait 100ms for batch
    props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "lz4"); // Fast compression

    return props;
}
```

3. Financial Transaction Processing

```
@Service
public class PaymentProcessor {
    private final KafkaProducer<String, PaymentEvent> producer;

    @Transactional
    public void processPayment(PaymentRequest request) {
        try {
            // Initialize transaction
            producer.initTransactions();
            producer.beginTransaction();

            // Validate payment
            PaymentValidation validation = paymentValidator.validate(request);
            if (!validation.isValid()) {
                throw new PaymentValidationException(validation.getErrors());
            }

            // Process payment with external service
            PaymentResult result = externalPaymentService.processPayment(request);

            // Record events atomically
            PaymentEvent event = new PaymentEvent(request.getPaymentId(),
                result.getStatus(), request.getAmount(),
System.currentTimeMillis());

            producer.send(new ProducerRecord<>("payment-events",
                request.getPaymentId(), event));

            // Update account balance event
            AccountEvent accountEvent = new AccountEvent(request.getAccountId(),
                AccountEventType.DEBIT, request.getAmount());

            producer.send(new ProducerRecord<>("account-events",
                request.getAccountId(), accountEvent));
```

```
            // Commit transaction
            producer.commitTransaction();

        } catch (Exception e) {
            producer.abortTransaction();
            throw new PaymentProcessingException("Payment failed", e);
        }
    }
}
```

Transactional Configuration:

```
Properties transactionalConfig() {
    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG, "payment-processor-1");
    // Exactly-once semantics automatically configured
    return props;
}
```

4. IoT Sensor Data Streaming

```
@Service
public class IoTDataProducer {
    private final KafkaProducer<String, SensorReading> producer;

    public void publishSensorReading(String sensorId, SensorReading reading) {
        // Custom partitioner routes by geographic region
        ProducerRecord<String, SensorReading> record =
            new ProducerRecord<>("sensor-data", sensorId, reading);

        // Add metadata headers
        record.headers().add("sensor-type", reading.getSensorType().getBytes());
        record.headers().add("location", reading.getLocation().getBytes());
        record.headers().add("timestamp",
            String.valueOf(reading.getTimestamp()).getBytes());

        producer.send(record, (metadata, exception) -> {
            if (exception != null) {
                // Log error but don't fail - sensor data is high volume
                logger.warn("Failed to send sensor reading for {}: {}",
                    sensorId, exception.getMessage());

                // Could implement local buffering or retry queue
                sensorFailureBuffer.add(record);
            }
        });
    }
}
```

5. Log Aggregation from Microservices

```
@Component
public class StructuredLogger {
    private final KafkaProducer<String, LogEvent> producer;

    public void logEvent(String service, LogLevel level, String message,
                         Map<String, Object> context) {
        LogEvent event = LogEvent.builder()
            .service(service)
            .level(level)
            .message(message)
            .context(context)
            .timestamp(System.currentTimeMillis())
            .hostname(getHostname())
            .build();

        // Partition by service for consistent routing
        ProducerRecord<String, LogEvent> record =
            new ProducerRecord<>("application-logs", service, event);

        producer.send(record); // Async - don't slow down application
    }
}

// Usage in application code
@RestController
public class UserController {

    @Autowired
    private StructuredLogger logger;

    @PostMapping("/users")
    public ResponseEntity<User> createUser(@RequestBody CreateUserRequest request)
{
        Map<String, Object> context = Map.of(
            "userId", request.getUserId(),
            "email", request.getEmail(),
            "source", "web-app"
        );

        logger.logEvent("user-service", LogLevel.INFO,
            "Creating new user", context);

        // ... business logic ...

        return ResponseEntity.ok(user);
    }
}
```

Version Highlights

Kafka 4.0 (September 2025) - Current Latest

- ✨ **Default KRaft Mode**: ZooKeeper completely removed
- ✨ **New Consumer Protocol (KIP-848)**: Faster rebalancing, better partition assignment
- ✨ **Enhanced Producer Batching**: Improved sticky partitioner performance
- ✨ **Java 17+ Required**: For brokers (Java 11+ for clients)

Kafka 3.x Series Producer Features

- **3.6** (Oct 2023): Producer metrics improvements, better error reporting
- **3.5** (Jun 2023): Enhanced idempotent producer stability
- **3.4** (Feb 2023): Improved transactional producer performance
- **3.3** (Oct 2022): Producer client optimizations for KRaft
- **3.2** (May 2022): Better producer error handling and retry logic
- **3.1** (Jan 2022): Producer connection pooling improvements
- **3.0** (Sep 2021): **Idempotence enabled by default**, improved batching

Key Producer Evolution

Version	Producer Feature	Impact
4.0	Enhanced batching algorithms	Better throughput
3.0	Idempotence by default	Exactly-once semantics out-of-box
2.8	Producer improvements for KRaft	Better metadata handling
2.4	Sticky partitioner	Better batching for null keys
2.1	Zstandard compression	Better compression ratios
1.1	Headers support	Metadata in messages
0.11	**Idempotent producer**	Exactly-once semantics
0.10	Message timestamps	Time-based operations
0.9	Producer rewrite	Much better performance

Current Recommendations (2025)

```
// Modern Kafka 4.0 producer configuration
public static Properties modernProducerConfig() {
    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);

    // Kafka 4.0 best practices
    props.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, true); // Default since
```

```
3.0
    props.put(ProducerConfig.ACKS_CONFIG, "all");              // Strong
durability
    props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "zstd");  // Best
compression (2.1+)

    // Optimized batching (improvements in 4.0)
    props.put(ProducerConfig.BATCH_SIZE_CONFIG, 32768);        // 32KB
    props.put(ProducerConfig.LINGER_MS_CONFIG, 10);            // 10ms

    return props;
}
```

🔗 Additional Resources

📚 Official Documentation

- Kafka Producer API Documentation
- Producer Configuration Reference
- Kafka Improvement Proposals - Producers

🎓 Learning Resources

- Confluent Producer Tutorial
- Apache Kafka: Producer Deep Dive
- Kafka Producer Best Practices

🔧 Tools & Monitoring

- JMX Metrics for Producers
- Kafka Producer Performance Testing

🐛 Troubleshooting

- Producer Error Handling
- Common Producer Issues

Last Updated: September 2025
Kafka Version: 4.0.0
Java Compatibility: 11+ (clients), 17+ (recommended)

> 💡 **Pro Tip**: Start with idempotence enabled and `acks=all` for reliability, then optimize for performance based on your specific use case requirements.