Distributed Messaging Systems Guide for System Design Interviews

Table of Contents

1. Introduction
2. Messaging System Fundamentals
3. Message Queue vs Pub/Sub
4. Popular Messaging Systems
5. Core Concepts
6. Delivery Guarantees
7. Ordering Guarantees
8. Message Patterns
9. Scalability & Performance
10. Fault Tolerance & Reliability
11. Common Interview Scenarios
12. How to Use in Interviews
13. Design Decisions & Trade-offs
14. Interview Tips & Best Practices

Introduction

A **Distributed Messaging System** is a software component that enables communication between different parts of a distributed application by passing messages asynchronously.

Why Use Messaging Systems?

Key Benefits:

- **Decoupling**: Services don't need to know about each other
- **Scalability**: Easy to add consumers/producers
- **Reliability**: Messages persisted until processed
- **Asynchronous Processing**: Non-blocking operations
- **Load Leveling**: Handle traffic spikes
- **Fault Tolerance**: System continues if components fail

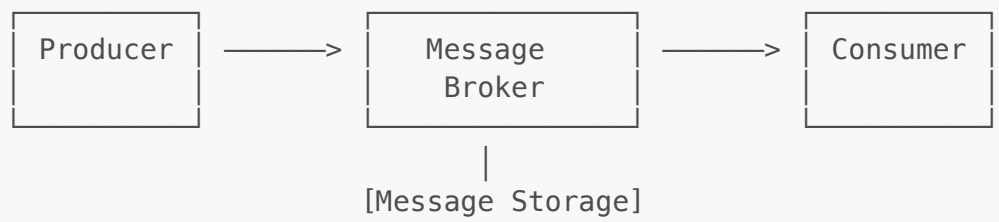
Common Use Cases

```
1. Event-Driven Architecture
   - Microservices communication
   - Real-time notifications
   - Event sourcing
   - CQRS pattern
```

```
2. Data Pipeline
   - Log aggregation
   - ETL processes
   - Stream processing
   - Real-time analytics

3. Task Queue
   - Background job processing
   - Email/SMS sending
   - Image processing
   - Report generation

4. Integration
   - Legacy system integration
   - Third-party API calls
   - Cross-platform communication
   - Service orchestration
```

Messaging System Fundamentals

Basic Architecture



Key Components

1. Producer (Publisher)

```
- Creates and sends messages
- Doesn't wait for consumer
- Can batch messages
- Handles connection failures

Responsibilities:
- Message formatting
- Serialization
- Partitioning logic
- Error handling
- Retry logic

Example (Kafka Producer):
Properties props = new Properties();
```

```
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "StringSerializer");
props.put("value.serializer", "StringSerializer");

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

ProducerRecord<String, String> record =
    new ProducerRecord<>("my-topic", "key", "value");

producer.send(record);
```

2. Message Broker

```
- Receives messages from producers
- Stores messages temporarily/permanently
- Routes messages to consumers
- Manages subscriptions
- Handles failures

Features:
- Message persistence
- Replication
- Load balancing
- Message routing
- Dead letter queue
- Monitoring & metrics

Popular Brokers:
- Apache Kafka
- RabbitMQ
- AWS SQS/SNS
- Google Pub/Sub
- Azure Service Bus
```

3. Consumer (Subscriber)

```
- Receives and processes messages
- Acknowledges processing
- Handles retries
- Maintains offset/position

Responsibilities:
- Message deserialization
- Processing logic
- Error handling
- Idempotency handling
```

```
- State management

Example (Kafka Consumer):
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "my-consumer-group");
props.put("key.deserializer", "StringDeserializer");
props.put("value.deserializer", "StringDeserializer");

Consumer<String, String> consumer =
    new KafkaConsumer<>(props);

consumer.subscribe(Arrays.asList("my-topic"));

while (true) {
    ConsumerRecords<String, String> records =
        consumer.poll(Duration.ofMillis(100));

    for (ConsumerRecord<String, String> record : records) {
        process(record);
    }
}
```

4. Message

```
Structure:
- Headers: Metadata (timestamp, ID, source)
- Key: For partitioning/routing
- Body: Actual data payload
- Attributes: Additional metadata

Example Message:
{
  "id": "msg-123",
  "timestamp": "2024-01-15T10:30:00Z",
  "source": "order-service",
  "type": "order.created",
  "key": "user-456",
  "body": {
    "orderId": "ord-789",
    "userId": "user-456",
    "items": [...],
    "total": 99.99
  }
}
```

Message Queue vs Pub/Sub

Message Queue (Point-to-Point)

```
Architecture:
┌──────────┐
│ Producer │──┐
└──────────┘  │
              ▼
┌──────────┐  ┌─────────┐  ┌──────────┐
│ Producer │─>│  Queue  │─>│ Consumer │
└──────────┘  └─────────┘  └──────────┘
              ▲
┌──────────┐  │
│ Producer │──┘
└──────────┘

Characteristics:
- Each message consumed by ONE consumer
- Load balanced across consumers
- FIFO ordering (typically)
- Competing consumers pattern
- Simple work distribution

Use Cases:
✅ Task processing
✅ Job queue
✅ Work distribution
✅ Load leveling
✅ Sequential processing

Examples:
- AWS SQS
- RabbitMQ (default)
- Azure Queue Storage
- Redis Lists
```

Pub/Sub (Publish-Subscribe)

```
Architecture:
                    ┌──────────┐
                ┌──>│ Consumer1│
                │   └──────────┘
┌──────────┐    │   ┌──────────┐
│ Publisher │────┼──>│ Consumer2│
└──────────┘    │   └──────────┘
                │   ┌──────────┐
                └──>│ Consumer3│
                    └──────────┘
```

```
Characteristics:
- Each message received by ALL subscribers
- Multiple independent consumers
- Fan-out pattern
- Topic-based routing
- Loose coupling

Use Cases:
✅ Event broadcasting
✅ Real-time notifications
✅ Data replication
✅ Logging/monitoring
✅ Cache invalidation

Examples:
- Apache Kafka
- AWS SNS
- Google Pub/Sub
- Redis Pub/Sub
- RabbitMQ (with exchanges)
```

Comparison Table

Feature	Queue	Pub/Sub
Consumers	One	Multiple
Message Copy	Single	Per sub
Coupling	Tight	Loose
Ordering	Strong	Per partition
Use Case	Work dist.	Broadcasting
Scalability	Horizontal	Horizontal
Complexity	Simple	Medium

Popular Messaging Systems

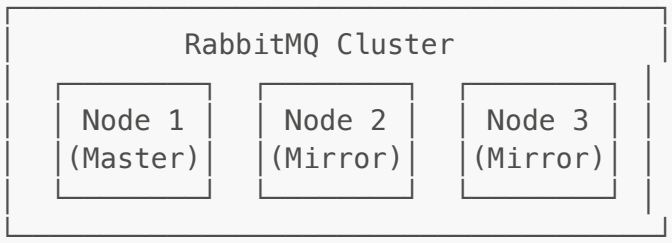
1. Apache Kafka

```
Overview:
- Distributed streaming platform
- High throughput (millions/sec)
- Persistent log-based storage
- Strong ordering guarantees
- Horizontal scalability
```

```
Architecture:
┌─────────────────────────────────────┐
│          Kafka Cluster              │
│  ┌──────────┐ ┌──────────┐ ┌──────────┐ │
│  │Broker 1  │ │Broker 2  │ │Broker 3  │ │
│  │(Leader)  │ │(Repli.)  │ │(Repli.)  │ │
│  └──────────┘ └──────────┘ └──────────┘ │
└─────────────────────────────────────┘
        ▲              ▼
   ┌──────────┐   ┌──────────┐
   │Producer  │   │ Consumer │
   └──────────┘   │  Group   │
                  └──────────┘

Key Concepts:
- Topics: Categories of messages
- Partitions: Parallel processing units
- Consumer Groups: Load balancing
- Offsets: Message position tracking
- Replication: Fault tolerance

Strengths:
✅ Extreme throughput
✅ Message replay
✅ Long-term storage
✅ Stream processing
✅ Strong ordering

Weaknesses:
❌ Complex setup
❌ Operational overhead
❌ Learning curve
❌ JVM dependency

Best For:
- Event streaming
- Log aggregation
- Real-time analytics
- Event sourcing
```

2. RabbitMQ

```
Overview:
- Traditional message broker
- AMQP protocol
- Flexible routing
- Multiple messaging patterns
- Easy to use

Architecture:
```




```
Key Concepts:
- Exchanges: Message routing
- Queues: Message storage
- Bindings: Routing rules
- Virtual Hosts: Multi-tenancy
- Plugins: Extensibility

Exchange Types:
1. Direct: Exact routing key match
2. Fanout: Broadcast to all
3. Topic: Pattern matching
4. Headers: Header-based routing

Strengths:
✅ Flexible routing
✅ Protocol support (AMQP, MQTT, STOMP)
✅ Management UI
✅ Plugin ecosystem
✅ Priority queues

Weaknesses:
❌ Lower throughput than Kafka
❌ No message replay
❌ Limited scalability
❌ Memory-based (default)

Best For:
- Microservices communication
- Task queues
- Complex routing
- Traditional messaging
```

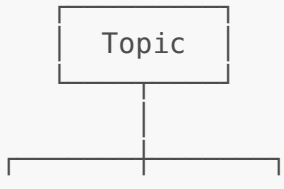
3. AWS SQS (Simple Queue Service)

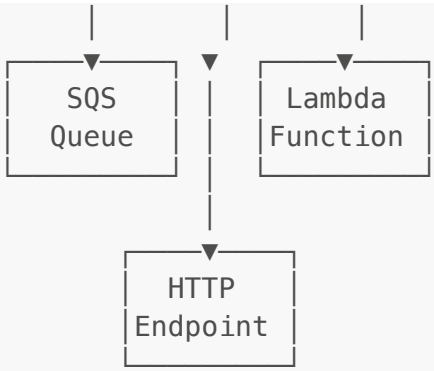
```
Overview:
- Fully managed queue service
- Serverless
- High availability
- Simple to use
- Pay per request

Types:
```

```
1. Standard Queue
   - At-least-once delivery
   - Best-effort ordering
   - Unlimited throughput

2. FIFO Queue
   - Exactly-once processing
   - Strict ordering
   - 3000 messages/sec limit

Key Features:
- Dead Letter Queue (DLQ)
- Message deduplication
- Visibility timeout
- Long polling
- Delay queues

Strengths:
✅ No management
✅ Automatic scaling
✅ High availability
✅ AWS integration
✅ Simple API

Weaknesses:
❌ AWS lock-in
❌ No message replay
❌ Limited features
❌ Cost at scale

Best For:
- AWS-based applications
- Simple queuing needs
- Decoupling services
- Batch processing
```

4. AWS SNS (Simple Notification Service)

```
Overview:
- Fully managed pub/sub
- Fan-out messaging
- Multiple protocols
- Push-based delivery

Architecture:
```






```
Supported Protocols:
- SQS
- HTTP/HTTPS
- Email/SMS
- Lambda
- Mobile Push

Strengths:
✅ Fan-out pattern
✅ Multiple destinations
✅ AWS integration
✅ Simple setup
✅ Message filtering

Weaknesses:
❌ AWS lock-in
❌ No persistence
❌ Limited features
❌ Push only

Best For:
- Event notifications
- Fan-out architecture
- Mobile push
- Email/SMS alerts
```

Comparison Matrix

Feature	Kafka	RabbitMQ	SQS	SNS
Throughput	Very High	High	High	High
Persistence	Days+	Memory	14 days	None
Ordering	Strong (part.)	Queue	FIFO option	No

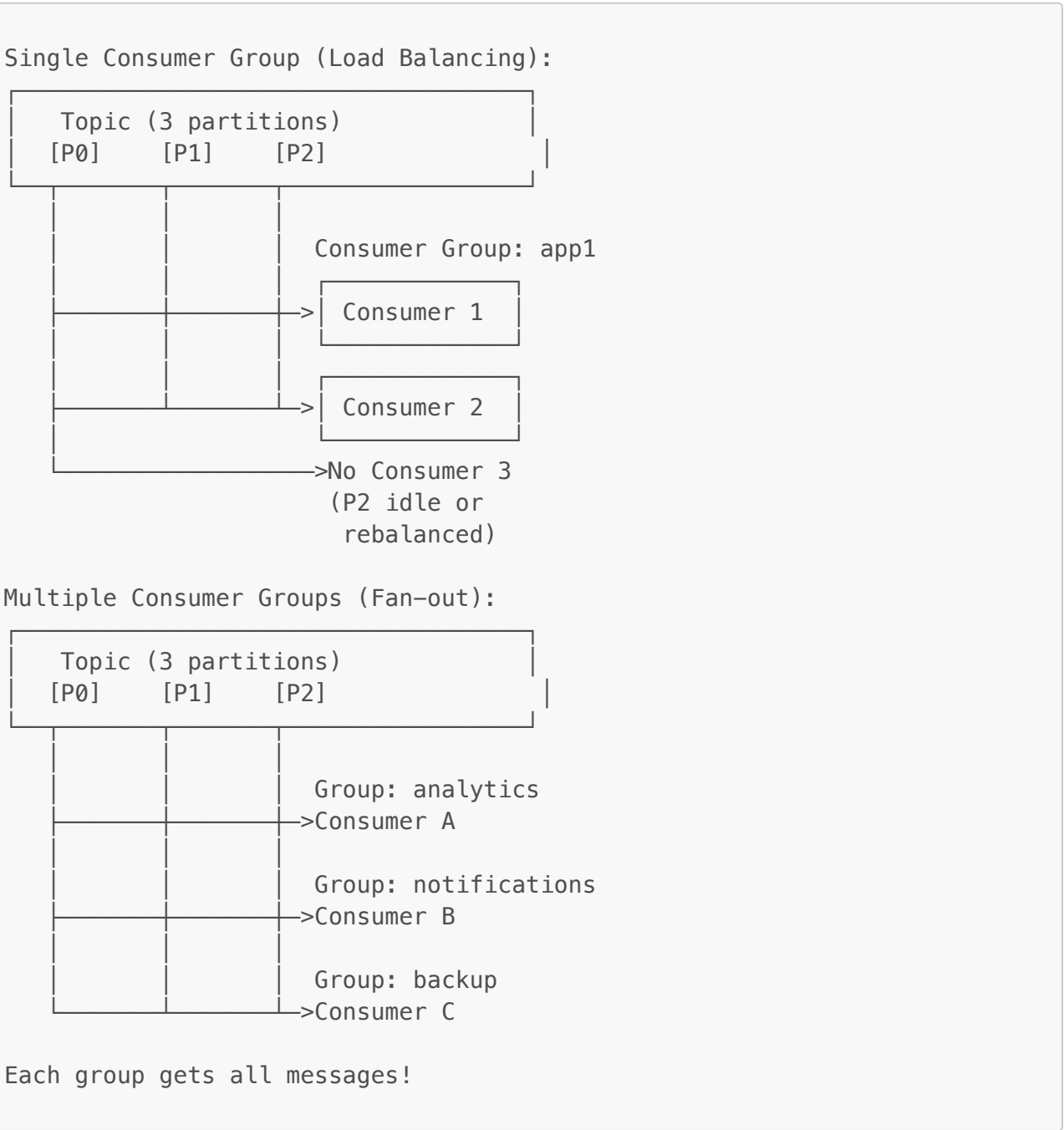
Replay	Yes	No	No	No
Scalability	Excellent	Good	Auto	Auto
Management	Complex	Medium	None	None
Latency	Low	Low	Medium	Low
Use Case	Streaming	Messaging	Queue	Fan-out

Core Concepts

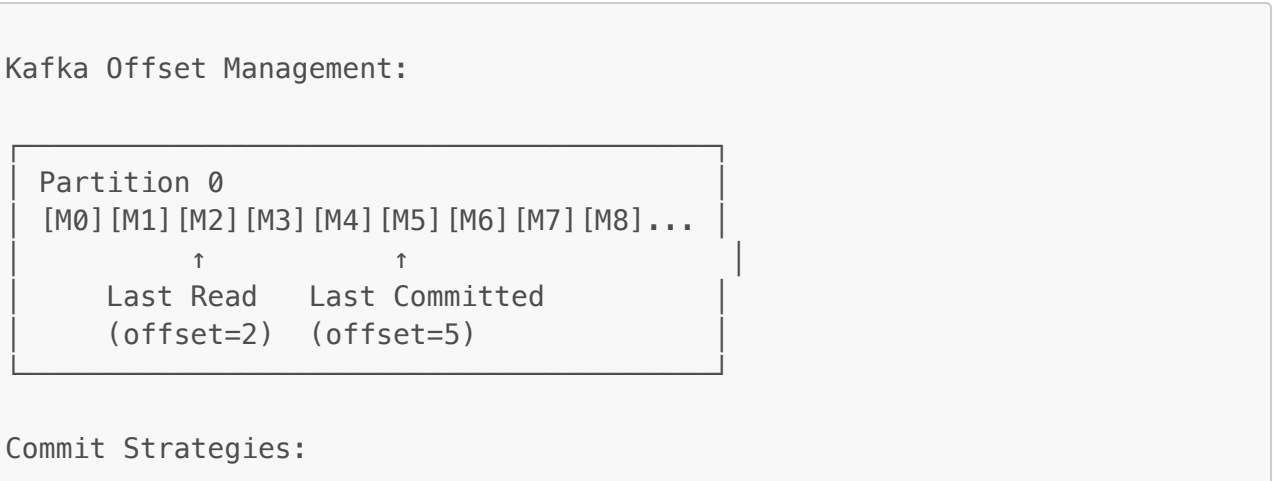
1. Topics and Partitions (Kafka)

```
Topic: Logical channel for messages

┌─────────────────────────────────────┐
│        Topic: "user-events"         │
├─────────────────────────────────────┤
│ Partition 0: [M1, M2, M5, M8, ...]  │
│ Partition 1: [M3, M6, M9, M12, ...] │
│ Partition 2: [M4, M7, M10, M11, ...]│
└─────────────────────────────────────┘

Partitioning Strategy:
1. Round-robin: No key
   - Balanced distribution
   - No ordering guarantee

2. Key-based: Hash(key) % partitions
   - Same key → same partition
   - Ordering per key
   - May cause skew

3. Custom partitioner
   - Business logic
   - Geographic routing
   - Time-based routing

Example:
// Messages with same userId go to same partition
record = new ProducerRecord(
    "user-events",
    userId,  // Key for partitioning
    event    // Value
);
```

2. Consumer Groups



3. Offsets and Acknowledgments



```
1. Auto-commit (default)
   - Periodic automatic commit
   - Simple but risky
   - May lose/duplicate messages

2. Manual commit
   - After processing
   - More control
   - Prevents data loss

3. At-least-once
   - Commit after processing
   - May process duplicates
   - Need idempotency

4. At-most-once
   - Commit before processing
   - May lose messages
   - Fast but risky

Example:
// Manual commit after processing
while (true) {
    records = consumer.poll(100);
    for (record : records) {
        process(record);  // Process first
    }
    consumer.commitSync();  // Then commit
}
```

4. Message Retention

```
Retention Policies:

1. Time-based
   - Keep for N days
   - Default: 7 days (Kafka)
   - Configurable per topic

   retention.ms = 604800000  // 7 days

2. Size-based
   - Keep last N GB
   - Prevents disk full

   retention.bytes = 1073741824  // 1GB

3. Compaction (Kafka)
   - Keep latest per key
   - Log compaction
```

```
   - Good for state

   cleanup.policy = compact

Retention Examples:

Time-based:
Day 1: [M1, M2, M3]
Day 2: [M1, M2, M3, M4, M5]
Day 3: [M1, M2, M3, M4, M5, M6]
Day 8: [M4, M5, M6, M7, M8, M9]
       (M1-M3 deleted, >7 days old)

Log Compaction:
Before: [K1:V1, K2:V2, K1:V3, K2:V4, K1:V5]
After:  [K1:V5, K2:V4]
        (Only latest value per key kept)
```

Delivery Guarantees

1. At-Most-Once

```
Definition:
Message delivered 0 or 1 times
May lose messages, never duplicates

Flow:
1. Produce message
2. DON'T wait for ack
3. Message may be lost

or

1. Consume message
2. Commit offset immediately
3. Process message
4. If processing fails, message lost

Pros:
✅ Fastest
✅ Simplest
✅ Lowest overhead

Cons:
❌ Data loss possible
❌ No guarantees

Use Cases:
- Metrics collection
```

```
- Sensor data
- Monitoring data
- Where loss acceptable

Configuration (Kafka Producer):
acks = 0  // Don't wait for ack
```

2. At-Least-Once

```
Definition:
Message delivered 1 or more times
Never lose messages, may duplicate

Flow:
1. Produce message
2. Wait for ack
3. Retry if no ack

or

1. Consume message
2. Process message
3. Commit offset
4. If processing succeeds but commit fails,
   message re-delivered

Pros:
✅ No data loss
✅ Simple to implement
✅ Good default choice

Cons:
❌ Duplicates possible
❌ Need idempotency

Use Cases:
- Event processing
- Order processing
- Payment processing
- Most business logic

Configuration (Kafka):
Producer:
  acks = all  // Wait for all replicas
  retries = MAX_INT

Consumer:
  enable.auto.commit = false
  // Manual commit after processing
```

```
Idempotency Implementation:
// Check if already processed
if (cache.contains(messageId)) {
    return;  // Skip duplicate
}
process(message);
cache.add(messageId);
commitOffset();
```

3. Exactly-Once

```
Definition:
Message delivered exactly 1 time
No loss, no duplicates

How It Works (Kafka):
1. Idempotent Producer
   - Producer assigns sequence number
   - Broker deduplicates

2. Transactional Processing
   - Atomic read-process-write
   - All or nothing

3. Transactional Consumer
   - Read and commit in transaction

Pros:
✅ Strongest guarantee
✅ No duplicates
✅ No loss

Cons:
❌ Performance overhead
❌ Complex setup
❌ Limited support

Use Cases:
- Financial transactions
- Billing systems
- Inventory management
- Critical data pipelines

Configuration (Kafka):
Producer:
  enable.idempotence = true
  transactional.id = "my-transaction-id"
  acks = all

Consumer:
```

```
  isolation.level = read_committed

Implementation:
producer.initTransactions();

try {
    producer.beginTransaction();

    // Send messages
    producer.send(record1);
    producer.send(record2);

    // Commit offsets
    producer.sendOffsetsToTransaction(
        offsets, consumerGroup);

    producer.commitTransaction();
} catch (Exception e) {
    producer.abortTransaction();
}
```

Comparison Table

Guarantee	Data Loss	Duplicates	Performance
At-Most-Once	Possible	No	Fastest
At-Least-Once	No	Possible	Fast
Exactly-Once	No	No	Slowest

```
Default Choice: At-Least-Once + Idempotency
```

Ordering Guarantees

1. No Ordering

```
Example:
Producer sends: [M1, M2, M3, M4, M5]
Consumer reads:  [M3, M1, M5, M2, M4]

Causes:
- Multiple partitions
- Network delays
- Retries
```

```
- Multiple producers

When It's OK:
- Independent messages
- Metrics/logging
- No causality needed

Systems:
- AWS SQS (Standard)
- Redis Pub/Sub
- Basic RabbitMQ
```

2. Per-Partition Ordering

```
Kafka Example:

Topic: orders (3 partitions)

Partition 0: [M1, M4, M7] → Order preserved
Partition 1: [M2, M5, M8] → Order preserved
Partition 2: [M3, M6, M9] → Order preserved

But overall: No global order

Key-based Partitioning:
// Same user → same partition → ordered
record = new ProducerRecord(
    "orders",
    userId,  // Partition by user
    order
);

Result:
- User A's orders: Always ordered
- User B's orders: Always ordered
- Between users: No order

Considerations:
✅ Scalable (parallel partitions)
✅ Ordered within key
❌ Hot partitions possible
❌ No global order
```

3. Global Ordering

```
Single Partition:

Topic: orders (1 partition)
```

```
[M1, M2, M3, M4, M5, M6, M7, ...]
  ↑
All messages in exact order

Single Consumer:
Only one consumer processes
Ensures sequential processing

Limitations:
❌ Limited throughput
❌ Single point of bottleneck
❌ No parallelism

When Needed:
- Financial transactions
- State machines
- Sequential workflows
- Strict causality

Alternative (FIFO Queue):
AWS SQS FIFO Queue
- Group ID for partitioning
- Strict order within group
- Limited to 3000 msg/sec
```

Ordering Strategies

```
Strategy 1: Partition by Entity
// All messages for same entity → same partition
key = entityId
partition = hash(key) % partitionCount

Example:
User orders → partition by userId
Bank transactions → partition by accountId

Strategy 2: Timestamp + Version
message = {
    id: "msg-123",
    timestamp: 1642098765,
    version: 5,
    data: {...}
}

Consumer:
if (message.version <= lastVersion) {
    discard();  // Out of order, already processed
}

Strategy 3: Event Sequencing
```

```
events = [
    {seq: 1, type: "order.created"},
    {seq: 2, type: "order.paid"},
    {seq: 3, type: "order.shipped"}
]

Consumer reorders if needed:
buffer = []
while (true) {
    msg = receive();
    buffer.add(msg);
    buffer.sort(by: seq);
    while (buffer[0].seq == expectedSeq) {
        process(buffer.remove(0));
        expectedSeq++;
    }
}
```

Message Patterns

1. Request-Response

```
Not native to messaging, but can be implemented:

Pattern:
┌──────────┐  Request   ┌──────────┐
│ Client   │ ─────────> │ Server   │
│          │            │          │
│          │ Response   │          │
│          │ <───────── │          │
└──────────┘            └──────────┘

Implementation:
1. Client sends request to request-queue
2. Client listens on reply-queue
3. Server processes from request-queue
4. Server sends response to reply-queue

Message Structure:
{
    correlationId: "req-123",
    replyTo: "reply-queue-abc",
    payload: {...}
}

Code Example:
// Client
correlationId = UUID.random();
replyQueue = createTempQueue();
```

```
send(requestQueue, {
    correlationId: correlationId,
    replyTo: replyQueue,
    payload: request
});

response = receive(replyQueue,
    filter: correlationId);

// Server
request = receive(requestQueue);
result = process(request.payload);

send(request.replyTo, {
    correlationId: request.correlationId,
    payload: result
});

Pros:
✅ Async RPC
✅ Decoupled
✅ Reliable

Cons:
❌ Complex
❌ Timeout handling
❌ Not ideal for messaging
```

2. Competing Consumers

```
Multiple consumers compete for messages:

┌──────────┐
│ Producer │
└──────────┘
     │
     ▼
┌──────────┐
│  Queue   │
│[M1][M2].. │
└──────────┘
  │   │   │
  ▼   ▼   ▼
┌──┐ ┌──┐ ┌──┐
│C1│ │C2│ │C3│  Consumers
└──┘ └──┘ └──┘

Load Distribution:
C1 processes: M1, M4, M7
```

```
C2 processes: M2, M5, M8
C3 processes: M3, M6, M9

Benefits:
✅ Load balancing
✅ Horizontal scaling
✅ Fault tolerance
✅ Parallel processing

Use Cases:
- Image processing
- Email sending
- Report generation
- Background jobs

Implementation:
// All consumers in same group
// compete for messages

consumer1 = new Consumer(group="workers");
consumer2 = new Consumer(group="workers");
consumer3 = new Consumer(group="workers");

Each message processed by only ONE consumer
```

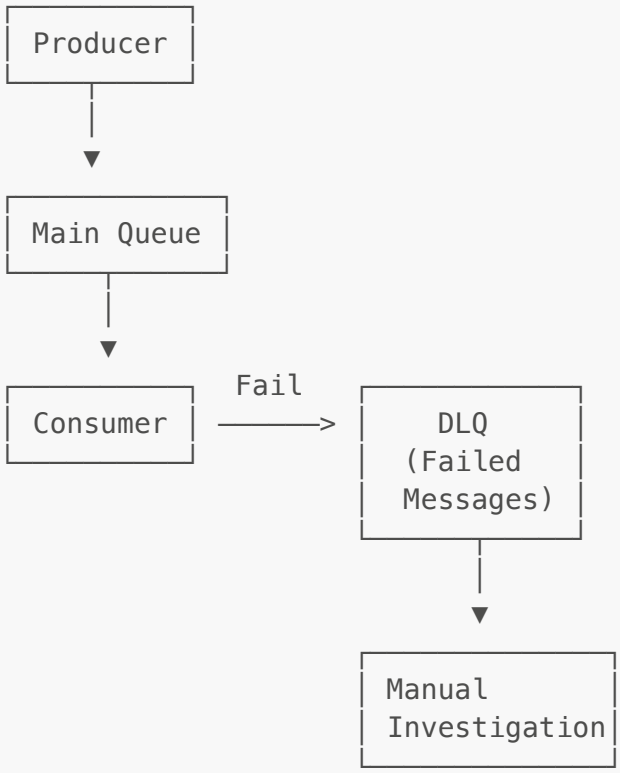
3. Publish-Subscribe

```
One message → multiple subscribers:

┌──────────────┐
│  Publisher   │
└──────┬───────┘
       │
       ▼
┌──────────────────┐
│      Topic       │
└──┬─────┬─────┬───┘
   │     │     │
   ▼     ▼     ▼
┌─────┐┌─────┐┌─────┐
│Sub1 ││Sub2 ││Sub3 │
└─────┘└─────┘└─────┘

All subscribers get all messages:
Sub1 receives: M1, M2, M3, M4...
Sub2 receives: M1, M2, M3, M4...
Sub3 receives: M1, M2, M3, M4...

Benefits:
✅ Fan-out
```

```
✅ Loose coupling
✅ Independent scaling
✅ Event-driven

Use Cases:
- Event notifications
- Real-time updates
- Data replication
- Monitoring

Implementation (Kafka):
// Different consumer groups
// each gets all messages

group1 = new Consumer(group="analytics");
group2 = new Consumer(group="notifications");
group3 = new Consumer(group="backup");

Each group independently consumes all messages
```

4. Dead Letter Queue (DLQ)

```
Pattern for handling failed messages:
```

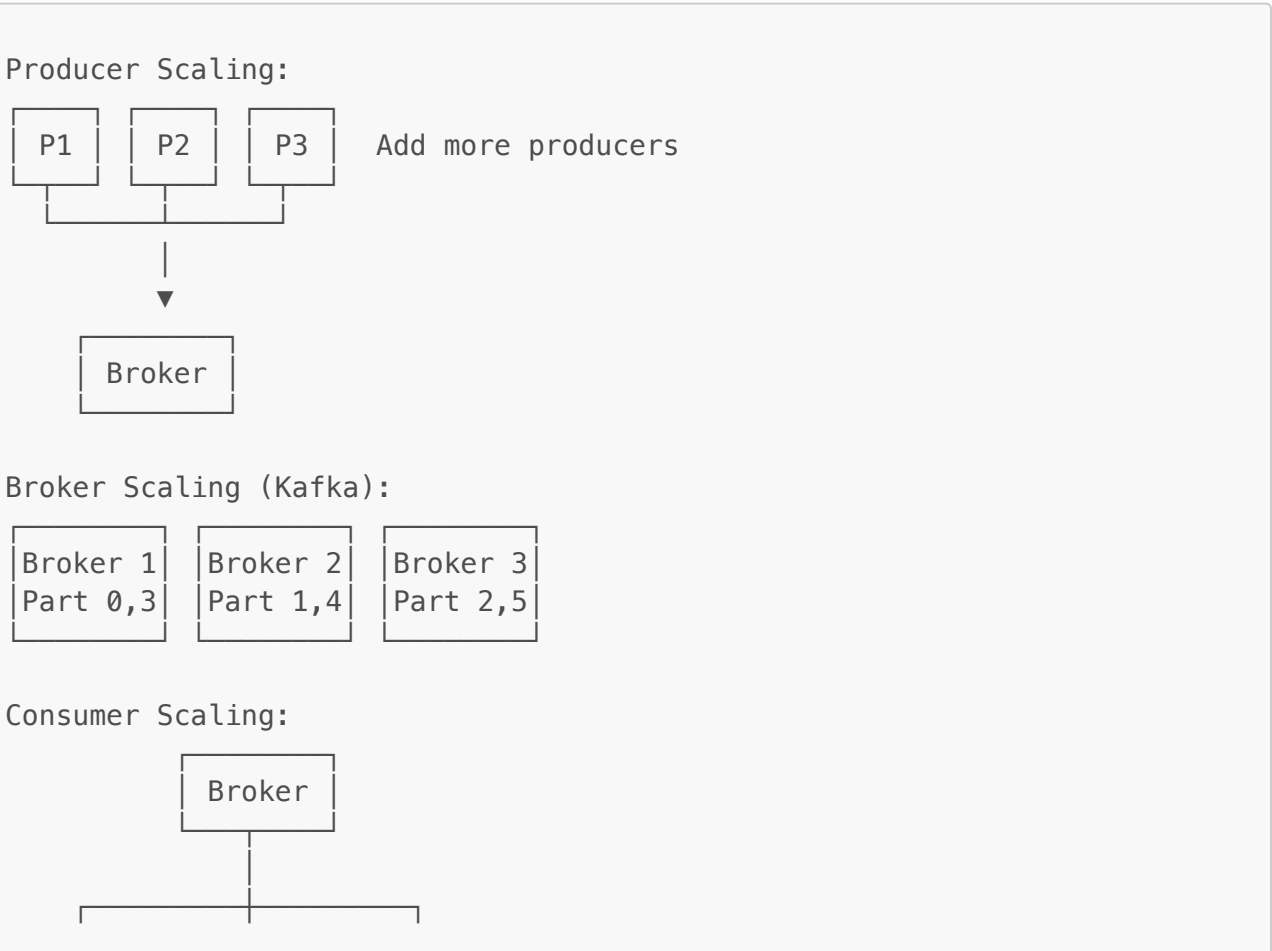



```
Reasons for DLQ:
- Processing exceptions
- Invalid message format
- Business logic failures
```

```
- Max retries exceeded

Configuration (SQS):
maxReceiveCount = 3  // Retries before DLQ

redrive_policy = {
  "deadLetterTargetArn": "arn:...:my-dlq",
  "maxReceiveCount": 3
}

Benefits:
✅ Prevents blocking
✅ Preserves failed messages
✅ Enables investigation
✅ System continues

DLQ Processing:
1. Monitor DLQ size
2. Investigate failures
3. Fix root cause
4. Reprocess or discard
```

Scalability & Performance

1. Horizontal Scaling



```
    │        │        │
  ┌─▼─┐    ┌─▼─┐    ┌─▼─┐
  │C1 │    │C2 │    │C3 │  Scale consumers
  └───┘    └───┘    └───┘

Scaling Rules:
- Producers: Unlimited
- Partitions: Plan ahead
- Consumers: ≤ Partitions per group
- Brokers: Add for throughput/storage
```

2. Performance Tuning

```
Producer Optimization:

1. Batching
   batch.size = 16384  // 16KB
   linger.ms = 10      // Wait 10ms

   Trade-off:
   ✅ Higher throughput
   ❌ Higher latency

2. Compression
   compression.type = "snappy"

   Options: none, gzip, snappy, lz4, zstd

   Trade-off:
   ✅ Lower network I/O
   ❌ CPU overhead

3. Async Send
   producer.send(record, callback);

   vs Sync:
   future = producer.send(record);
   future.get();  // Blocks

Consumer Optimization:

1. Fetch Size
   max.partition.fetch.bytes = 1048576  // 1MB

2. Parallel Processing
   // Process messages in parallel
   ExecutorService executor =
       Executors.newFixedThreadPool(10);

   for (record : records) {
```

```
        executor.submit(() -> process(record));
    }

3. Batch Processing
    // Process records in batches
    records = consumer.poll(1000);
    processBatch(records);
    consumer.commitSync();

Performance Metrics:

Throughput:
- Messages/second
- MB/second
- Transactions/second

Latency:
- End-to-end latency
- Producer latency
- Consumer latency
- P50, P95, P99 percentiles

Example Kafka Performance:
- Throughput: 1M+ msg/sec per broker
- Latency: < 10ms (P99)
- Partitions: 1000+ per broker
- Retention: Days to weeks
```

3. Partitioning Strategies

```
1. Hash Partitioning (Default)
partition = hash(key) % numPartitions

Pros:
✅ Even distribution
✅ Simple
✅ Predictable

Cons:
❌ Hot partitions if skewed keys
❌ Rebalancing on partition changes

2. Range Partitioning
partition = determineRange(key)

Example:
A-M → Partition 0
N-Z → Partition 1

Pros:
```

```
✅ Logical grouping
✅ Range queries

Cons:
❌ Potential skew
❌ Manual management

3. Custom Partitioning
class CustomPartitioner implements Partitioner {
    public int partition(
        String topic, Object key,
        byte[] keyBytes, Object value,
        byte[] valueBytes, Cluster cluster) {

        // Geographic partitioning
        if (key.startsWith("US-")) return 0;
        if (key.startsWith("EU-")) return 1;
        if (key.startsWith("APAC-")) return 2;
        return 3;
    }
}

Pros:
✅ Business logic
✅ Custom requirements
✅ Optimization

Cons:
❌ Complexity
❌ Maintenance

Partition Count Considerations:
- Start with more partitions than consumers
- Rule of thumb: 1-2 partitions per broker
- Consider: throughput, retention, consumers
- Hard to change later (Kafka)
```

Fault Tolerance & Reliability

1. Replication

```
Kafka Replication:

Topic: orders (replication-factor=3)

┌─────────────────────────────────────┐
│ Partition 0                         │
├─────────────────────────────────────┤
│ Broker 1 (Leader)   : [M1, M2, M3] │
```

```
│ Broker 2 (Follower) : [M1, M2, M3] │
│ Broker 3 (Follower) : [M1, M2, M3] │
└─────────────────────────────────────┘

Leader Election:
1. Leader fails
2. Zookeeper/Kraft detects
3. ISR (In-Sync Replica) elected
4. Clients redirected

Configuration:
replication.factor = 3
min.insync.replicas = 2

acks = all  // Wait for all replicas
acks = 1    // Wait for leader only
acks = 0    // No wait

Trade-offs:
Durability vs Performance:
- acks=all: Safest, slowest
- acks=1: Balanced
- acks=0: Fastest, risky
```

2. Durability

```
Message Persistence:

Kafka (Disk):
- Writes to append-only log
- Sequential I/O (fast)
- Survives broker restart
- Configurable retention

flush.messages = 10000
flush.ms = 1000

RabbitMQ (Memory):
- Messages in RAM by default
- Can persist to disk
- Slower but durable

durable = true
delivery_mode = 2  // Persistent

SQS (Managed):
- Automatically durable
- Replicated across AZs
- No configuration needed
```

```
Durability Guarantees:
- Write to disk before ack
- Replicate before ack
- fsync for durability
- Battery-backed write cache
```

3. High Availability

```
Cluster Setup:

Kafka (3 brokers):
┌──────────┐ ┌──────────┐ ┌──────────┐
│Broker 1│ │Broker 2│ │Broker 3│
│ us-1a  │ │ us-1b  │ │ us-1c  │
└──────────┘ └──────────┘ └──────────┘

RabbitMQ (Mirrored Queues):
┌──────────┐ ┌──────────┐ ┌──────────┐
│ Node 1 │ │ Node 2 │ │ Node 3 │
│(Master)│ │(Mirror)│ │(Mirror)│
└──────────┘ └──────────┘ └──────────┘

Failure Scenarios:

1. Broker Failure
   - Replica promoted to leader
   - Clients reconnect
   - No data loss (if replicated)
   - Recovery: < 30 seconds

2. Network Partition
   - Split-brain prevention
   - Quorum required
   - Minority partition read-only

3. Disk Failure
   - Replicas serve data
   - Replace failed disk
   - Rebuild replica

Health Monitoring:
- Heartbeats
- Zookeeper/Kraft
- Metrics (lag, errors)
- Alerts

Best Practices:
✅ Odd number of nodes (3, 5)
✅ Cross-AZ deployment
✅ Regular backups
```

```
✅ Monitoring and alerts
✅ Runbooks for failures
```

4. Retry Mechanisms

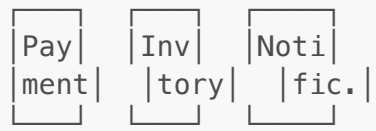
```
Producer Retries:

Configuration:
retries = Integer.MAX_VALUE
retry.backoff.ms = 100
delivery.timeout.ms = 120000

Retry Strategy:
attempt = 0
while (attempt < maxRetries) {
    try {
        send(message);
        break;
    } catch (RetriableException e) {
        attempt++;
        backoff = min(
            initialBackoff * 2^attempt,
            maxBackoff
        );
        sleep(backoff);
    }
}

Exponential Backoff:
Attempt 1: 100ms
Attempt 2: 200ms
Attempt 3: 400ms
Attempt 4: 800ms
Attempt 5: 1600ms
...
Max: 30000ms (30s)

Consumer Retries:

Dead Letter Queue Pattern:
while (true) {
    message = consume();
    attempt = 0;

    while (attempt < maxRetries) {
        try {
            process(message);
            commit();
            break;
        } catch (RetriableException e) {
```

```
            attempt++;
            if (attempt >= maxRetries) {
                sendToDLQ(message);
                commit();
            } else {
                sleep(backoffTime);
            }
        }
    }
}

Idempotency:
messageId = message.getId();

if (processedIds.contains(messageId)) {
    // Already processed, skip
    commit();
    continue;
}

process(message);
processedIds.add(messageId);
commit();
```

Common Interview Scenarios

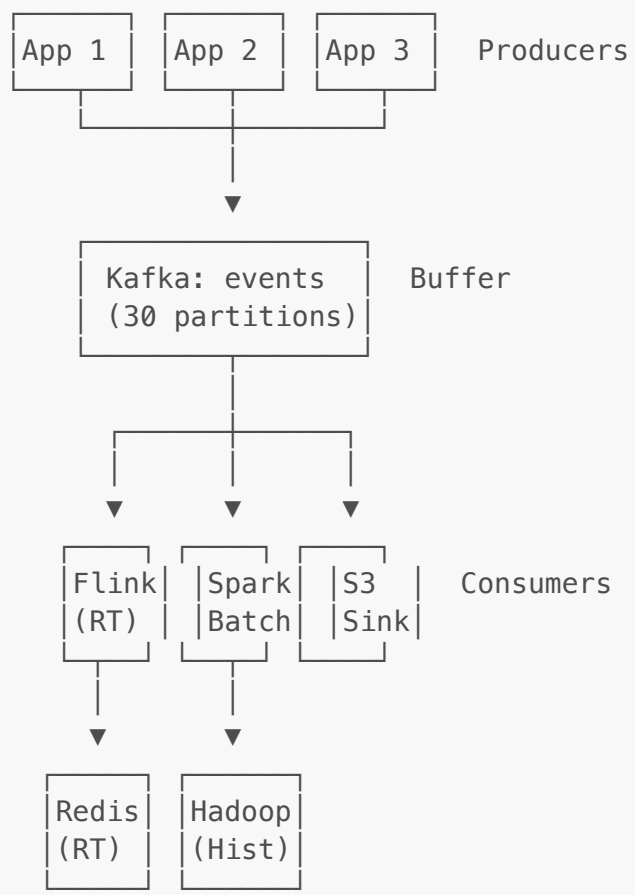
1. Order Processing System

```
Requirements:
- E-commerce orders
- Payment processing
- Inventory updates
- Order fulfillment
- Notifications

Architecture:

┌──────────────┐
│ Order        │
│ Service      │
└──────┬───────┘
       │ 1. Create Order
       ▼
┌──────────────────────┐
│ Kafka: orders        │
└──┬──────┬──────┬─────┘
   │      │      │
   │      │      │
   ▼      ▼      ▼
```




```
Flow:
1. Order created → orders topic
2. Payment service:
   - Consumes order
   - Processes payment
   - Produces to payments topic
3. Inventory service:
   - Consumes order
   - Updates inventory
   - Produces to inventory topic
4. Fulfillment service:
   - Consumes payments + inventory
   - Triggers shipping
5. Notification service:
   - Consumes all events
   - Sends emails/SMS

Key Decisions:
- Topic: orders (partitioned by userId)
- Delivery: At-least-once + idempotency
- Ordering: Per-user ordering
- Retention: 7 days
- Replication: 3 replicas

Message Format:
{
  "orderId": "ord-123",
  "userId": "user-456",
  "items": [{...}],
  "total": 99.99,
  "timestamp": "2024-01-15T10:30:00Z",
  "eventType": "order.created"
}

Challenges & Solutions:
1. Payment failure
   → Retry with exponential backoff
   → DLQ after max retries

2. Inventory out of stock
   → Compensating transaction
   → Cancel order, refund payment

3. Duplicate processing
   → Idempotency key in database
   → Check before processing
```

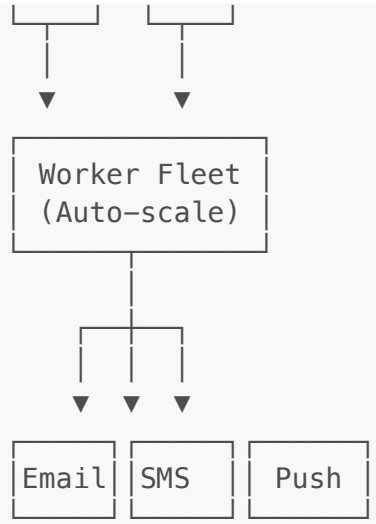
2. Real-time Analytics Pipeline

```
Requirements:
- Millions of events/second
- Real-time dashboards
- Historical analysis
- Stream processing

Architecture:




Configuration:
Topic: events
Partitions: 30
Retention: 30 days
Replication: 3
Compression: snappy

Producer:
batch.size = 32768  // 32KB
linger.ms = 100     // Batch for 100ms
compression.type = "snappy"
acks = 1            // Leader only

Consumer Groups:
1. real-time (Flink)
   - Low latency processing
```

```
   - Immediate aggregations
   - Write to Redis

2. batch (Spark)
   - Hourly aggregations
   - Complex analytics
   - Write to Hadoop

3. backup (S3 Sink)
   - Raw data backup
   - Data lake
   - Long-term storage

Scaling:
- Start: 10 partitions
- Growth: Add partitions (30)
- Consumers: 10 per group
- Throughput: 2M events/sec

Key Metrics:
- End-to-end latency: < 100ms (P95)
- Throughput: 50MB/sec
- Storage: 1.5TB/day
- Retention: 30 days = 45TB
```

3. Notification System

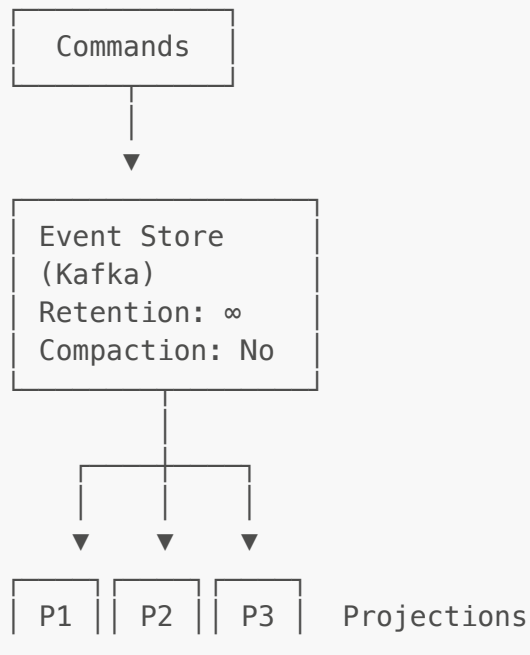
```
Requirements:
- Email, SMS, Push notifications
- Priority levels
- Retry logic
- Rate limiting
- Dead letter handling

Architecture:

┌──────────────┐
│   Services   │
│ (100+ apps)  │
└──────┬───────┘
       │
       ▼
┌──────────────┐
│ SNS: notif-req │  Fan-out
└──┬────────┬──┘
   │        │
   ▼        ▼
┌────┐   ┌────┐
│SQS │   │SQS │  Priority Queues
│High│   │Low │
```




```
Message Flow:
1. Service publishes to SNS
2. SNS fans out to SQS queues
3. Workers poll based on priority
4. Workers send notifications
5. Failed → DLQ

Message Format:
{
  "notificationId": "notif-123",
  "userId": "user-456",
  "type": "email",
  "priority": "high",
  "template": "order-confirmation",
  "data": {...},
  "maxRetries": 3,
  "timestamp": "2024-01-15T10:30:00Z"
}

Priority Handling:
High Priority Queue:
- Process immediately
- 5 workers
- Visibility timeout: 30s

Low Priority Queue:
- Process when available
- 2 workers
- Visibility timeout: 60s

Rate Limiting:
// Per user rate limit
userKey = message.userId
currentCount = redis.incr(userKey)
redis.expire(userKey, 3600)  // 1 hour

if (currentCount > 100) {
    // User exceeded limit
```

```
    requeueWithDelay(message, 3600);
    return;
}

sendNotification(message);

Retry Strategy:
attempt = message.retryCount || 0
if (attempt < 3) {
    // Exponential backoff
    delay = 60 * (2 ^ attempt)
    requeueWithDelay(message, delay);
} else {
    sendToDLQ(message);
}

Scaling:
- Workers: Auto-scale on queue depth
- Target: < 100 messages in queue
- Scale-out: Queue depth > 1000
- Scale-in: Queue depth < 100
```

4. Event Sourcing System

```
Requirements:
- Store all state changes
- Event replay
- Audit trail
- CQRS pattern
- Point-in-time recovery

Architecture:



```

```
└─────┘└─────┘└─────┘
  |     |     |
  ▼     ▼     ▼
┌─────────────────┐
│  Read Models    │
│ (Materialized)  │
└─────────────────┘

Event Stream Example:
Topic: account-events
Partition by: accountId

Events:
[
  {seq: 1, type: "AccountCreated",
   data: {accountId: "acc-1", balance: 0}},

  {seq: 2, type: "MoneyDeposited",
   data: {accountId: "acc-1", amount: 100}},

  {seq: 3, type: "MoneyWithdrawn",
   data: {accountId: "acc-1", amount: 30}},

  {seq: 4, type: "MoneyDeposited",
   data: {accountId: "acc-1", amount: 50}}
]

Current State = Replay all events:
Initial: 0
After event 2: 100
After event 3: 70
After event 4: 120

Projections:
1. Account Balance View
   - Consumes events
   - Updates balance in DB
   - Fast reads

2. Transaction History
   - All events stored
   - Audit trail
   - Compliance

3. Analytics View
   - Aggregate statistics
   - Daily balances
   - Trends

Configuration:
Topic: account-events
Retention: Infinite
Compaction: None (keep all)
```

```
Replication: 3
Partitions: 100

Benefits:
✅ Complete audit trail
✅ Time travel queries
✅ Event replay
✅ Multiple views
✅ Debugging

Challenges:
1. Storage growth
   → Cost optimization
   → Tiered storage

2. Schema evolution
   → Version events
   → Backward compatibility

3. Performance
   → Snapshots
   → CQRS separation
```

How to Use in Interviews

Interview Framework (35-40 minutes)

1. Requirements Gathering (5 min)

```
Key Questions to Ask:

Scale:
- How many messages per second?
- Message size?
- Number of producers/consumers?
- Growth projections?

Guarantees:
- Delivery guarantee needed?
  (At-most-once, At-least-once, Exactly-once)
- Ordering required?
  (No order, Per-key, Global)
- Latency requirements?
- Durability needs?

Reliability:
- Acceptable data loss?
- Downtime tolerance?
- Disaster recovery?
```

```
Example:
Interviewer: "Design a system for processing orders"

You: "Great! Let me clarify a few things:
- How many orders per second do we expect?
- Do we need strict ordering of orders per user?
- Can we tolerate message loss?
- What's our latency requirement?
- Do orders need to be processed exactly once?"
```

2. High-Level Design (10-15 min)

```
Components to Cover:

1. Choose Messaging System
   Decision Matrix:

   High Throughput + Replay → Kafka
   Simple Queue → SQS
   Complex Routing → RabbitMQ
   Fan-out → SNS/Kafka

2. Draw Architecture

   [Producers] → [Broker] → [Consumers]

   Add:
   - Topics/Queues
   - Consumer groups
   - Dead letter queues

3. Explain Flow
   - How messages are produced
   - How they're routed
   - How consumers process
   - Error handling

Example Explanation:
"I'll use Kafka here because:
1. We need high throughput (100K orders/sec)
2. Multiple teams need same data (fan-out)
3. We want message replay for analytics
4. Strong ordering per user is needed

Architecture:
- Order service produces to 'orders' topic
- 30 partitions (partition by userId)
- 3 consumer groups:
  * Payment processing
```

```
  * Inventory management
  * Notification service
- Each can scale independently"
```

3. Deep Dive (15-20 min)

```
Topics to Cover:

A. Partitioning Strategy
"I'll partition by userId because:
- Orders from same user stay ordered
- Load distributes evenly
- Hot users won't create hot partitions
  (we have many users)
- Alternative: Order by orderId but loses
  per-user ordering"

B. Delivery Guarantees
"We need at-least-once delivery:
- Can't lose orders (money involved)
- Duplicate processing acceptable if
  we handle idempotency
- Implementation:
  * acks=all on producer
  * Manual commit after processing
  * Idempotency key in database"

C. Scaling
"Horizontal scaling plan:
- Producers: Unlimited, auto-scale
- Partitions: Start with 30, can add more
- Consumers: Max 30 per group (=partitions)
- Brokers: 3 initially, add as needed
- Target: 100K orders/sec today
- Capacity: 500K orders/sec"

D. Fault Tolerance
"Reliability measures:
- Replication factor: 3
- min.insync.replicas: 2
- Multi-AZ deployment
- Dead letter queue for failures
- Monitoring and alerts
- Automatic leader election"

E. Monitoring
"Key metrics to track:
- Consumer lag (most important!)
- Throughput (msgs/sec)
- Error rate
```

```
- Latency (P50, P95, P99)
- Disk usage
- Replication lag

Alerts:
- Lag > 10000 messages
- Error rate > 1%
- Disk > 80% full"
```

4. Trade-offs & Alternatives (5 min)

```
Discuss Trade-offs:

1. Kafka vs SQS
You: "I chose Kafka over SQS because:
- Better throughput (Kafka: 1M/sec vs SQS: 100K/sec)
- Message replay needed
- Multiple consumer groups needed
- BUT: More complex to operate
- Alternative: SQS if simpler requirements"

2. Partition Count
You: "Started with 30 partitions:
- Pros: Good parallelism
- Cons: Can't reduce later
- Alternative: Start small (10), but
  repartitioning is complex
- Decision: Better to over-provision"

3. Delivery Guarantee
You: "Chose at-least-once over exactly-once:
- Pros: Better performance
- Simpler implementation
- Cons: Need idempotency
- Why: Exactly-once overhead not worth it
  when idempotency is straightforward"

4. Synchronous vs Asynchronous
You: "Async messaging vs REST:
- Pros: Decoupling, scalability, resilience
- Cons: Complexity, eventual consistency
- When sync: User needs immediate response
- When async: Background processing OK"
```

Common Interview Pitfalls to Avoid

```
❌ DON'T:
1. Jump to solution without requirements
```

```
2. Ignore trade-offs
3. Overengineer for small scale
4. Forget about monitoring
5. Ignore failure scenarios
6. Claim "no data loss" without details
7. Forget about operational complexity
8. Ignore cost considerations

✅ DO:
1. Ask clarifying questions
2. Start simple, then iterate
3. Explain your reasoning
4. Discuss alternatives
5. Consider failure modes
6. Talk about monitoring
7. Think about operations
8. Be honest about limitations
```

Key Talking Points

```
1. "I'd start with..."
   - Shows iterative thinking
   - Not over-engineering

2. "The trade-off here is..."
   - Shows depth of understanding
   - Considers alternatives

3. "At scale, we'd need..."
   - Forward thinking
   - Scalability awareness

4. "To handle failures..."
   - Reliability focus
   - Production mindset

5. "We could also..."
   - Shows flexibility
   - Multiple solutions

6. "The bottleneck would be..."
   - Performance awareness
   - System thinking

7. "For monitoring, I'd track..."
   - Operational maturity
   - Production experience

8. "This assumes...clarify please"
```

```
   - Requirements gathering
   - Not making assumptions
```

Design Decisions & Trade-offs

1. Messaging System Selection

```
Decision Matrix:

Use Kafka when:
✅ High throughput (>100K msgs/sec)
✅ Message replay needed
✅ Long retention (days/weeks)
✅ Multiple consumer groups
✅ Stream processing
✅ Event sourcing

Avoid Kafka when:
❌ Simple queue needed
❌ Low throughput (<1K msgs/sec)
❌ Quick setup required
❌ Limited ops team
❌ No replay needed

Use RabbitMQ when:
✅ Complex routing needed
✅ Traditional messaging patterns
✅ Priority queues required
✅ Request-response pattern
✅ Multiple protocols needed

Avoid RabbitMQ when:
❌ Extreme throughput needed
❌ Message replay required
❌ Long-term storage needed

Use SQS when:
✅ AWS-based system
✅ Serverless architecture
✅ Simple queue needed
✅ No ops overhead wanted
✅ FIFO or Standard sufficient

Avoid SQS when:
❌ Message replay needed
❌ Multi-cloud required
❌ Very high throughput
❌ Long retention (>14 days)
```

```
Use SNS when:
✅ Fan-out needed
✅ Multiple destinations
✅ Push notifications
✅ AWS ecosystem
✅ Simple pub/sub

Avoid SNS when:
❌ Pull-based needed
❌ Message persistence required
❌ Replay functionality wanted
```

2. Delivery Guarantee Selection

```
At-Most-Once:
Use When:
✅ Metrics/logging
✅ Data loss acceptable
✅ Performance critical
✅ Sensor data

Example: IoT sensor readings
- 1000 readings/sec per sensor
- Loss of few readings OK
- Real-time processing
- No retry needed

At-Least-Once:
Use When:
✅ Most business logic
✅ Idempotency possible
✅ No data loss tolerated
✅ Good performance needed

Example: Order processing
- Orders can't be lost
- Duplicate processing OK
- Idempotency in database
- Balance: reliable & fast

Exactly-Once:
Use When:
✅ Financial transactions
✅ No duplicates tolerated
✅ Critical data
✅ Can afford overhead

Example: Payment processing
- No duplicate charges
- Money movement
```

```
- Audit requirements
- Worth the overhead
```

3. Ordering vs Throughput

```
Trade-off:

Global Ordering:
Throughput: LOW (single partition)
Latency: MEDIUM
Scalability: NONE
Use Case: Financial ledger

Example:
Topic: transactions (1 partition)
Throughput: ~10K msgs/sec
Latency: 10-20ms
Scaling: Vertical only

Per-Key Ordering:
Throughput: HIGH (many partitions)
Latency: LOW
Scalability: EXCELLENT
Use Case: User events

Example:
Topic: user-events (30 partitions)
Throughput: 500K msgs/sec
Latency: 5-10ms
Scaling: Horizontal

No Ordering:
Throughput: VERY HIGH (many partitions + parallelism)
Latency: VERY LOW
Scalability: MAXIMUM
Use Case: Metrics, logs

Example:
Topic: metrics (100 partitions)
Throughput: 2M msgs/sec
Latency: < 5ms
Scaling: Unlimited

Decision Framework:
Need strict global order? → Single partition
Need per-entity order? → Partition by entity
No ordering needed? → Max partitions for throughput
```

Interview Tips & Best Practices

Key Concepts to Master

```
1. Messaging Fundamentals
   - Producer/Consumer/Broker
   - Topics vs Queues
   - Pub/Sub vs Point-to-Point
   - Message structure

2. Delivery Guarantees
   - At-most-once
   - At-least-once
   - Exactly-once
   - Implementation details

3. Ordering Guarantees
   - No ordering
   - Per-partition ordering
   - Global ordering
   - Trade-offs

4. Scaling Patterns
   - Horizontal scaling
   - Partitioning strategies
   - Consumer groups
   - Replication

5. Reliability
   - Fault tolerance
   - Replication
   - Dead letter queues
   - Retry mechanisms
```

Common Mistakes to Avoid

```
❌ DON'T:
1. Use global ordering when not needed (kills scalability)
2. Ignore idempotency with at-least-once delivery
3. Over-provision partitions excessively
4. Forget about monitoring and alerting
5. Choose exactly-once when at-least-once + idempotency works
6. Ignore message size (can cause performance issues)
7. Forget about schema evolution
8. Overlook operational complexity
9. Ignore cost implications
10. Assume infinite retention is free

✅ DO:
```

```
1. Start with at-least-once + idempotency (good default)
2. Partition by entity for per-entity ordering
3. Plan partition count carefully (hard to change)
4. Implement proper monitoring from day 1
5. Use DLQ for failed messages
6. Consider message compression
7. Version your message schemas
8. Think about operations and maintenance
9. Estimate costs based on throughput
10. Set appropriate retention policies
```

Interview Red Flags (What Not to Say)

```
❌ "We'll never lose messages"
   (without explaining how)

❌ "Kafka is always the best choice"
   (context matters)

❌ "We don't need monitoring"
   (always need it)

❌ "Exactly-once is easy"
   (it's complex)

❌ "We'll just add more partitions later"
   (hard with Kafka)

❌ "Message order doesn't matter"
   (without understanding requirements)

❌ "We'll handle that in production"
   (shows lack of planning)

❌ "I don't know the trade-offs"
   (shows surface-level knowledge)
```

Strong Interview Answers

```
✅ "I'd use at-least-once with idempotency because..."
   (shows understanding of trade-offs)

✅ "The bottleneck would be X because..."
   (shows system thinking)

✅ "We could use Kafka or SQS. Kafka if... SQS if..."
   (shows multiple solutions)
```

```
✅ "For monitoring, I'd track consumer lag, throughput..."
   (shows production mindset)

✅ "The trade-off between ordering and throughput is..."
   (shows deep understanding)

✅ "To handle this failure scenario, we'd..."
   (shows reliability focus)

✅ "Let me clarify the requirements first..."
   (shows requirements gathering)

✅ "This assumes X. If Y instead, we'd do Z..."
   (shows flexibility)
```

Quick Reference Checklist

Messaging System Design Checklist

```
Requirements Phase:
□ Clarify throughput needs (msgs/sec)
□ Understand latency requirements
□ Determine delivery guarantees needed
□ Identify ordering requirements
□ Assess durability needs
□ Consider retention period
□ Understand scaling projections

System Selection:
□ Choose appropriate messaging system
□ Consider operational complexity
□ Evaluate cost implications
□ Check team expertise
□ Verify cloud/on-prem fit

Architecture Design:
□ Define topics/queues structure
□ Plan partitioning strategy
□ Design consumer groups
□ Plan for dead letter queues
□ Consider message schema
□ Design retry mechanism

Reliability:
□ Configure replication
□ Set up health monitoring
□ Implement failure handling
□ Plan disaster recovery
□ Set up alerts
```

```
□ Document runbooks

Performance:
□ Configure batching
□ Enable compression
□ Optimize partition count
□ Tune consumer settings
□ Plan for scaling
□ Set retention policies

Security:
□ Configure authentication
□ Set up authorization
□ Enable encryption in transit
□ Enable encryption at rest
□ Implement access controls
□ Audit logging

Monitoring:
□ Track consumer lag
□ Monitor throughput
□ Track error rates
□ Monitor latency (P50, P95, P99)
□ Set up dashboards
□ Configure alerts

Operations:
□ Document architecture
□ Create runbooks
□ Plan capacity
□ Estimate costs
□ Train team
□ Plan maintenance windows
```

Conclusion

Distributed messaging systems are fundamental to modern distributed applications. During system design interviews, remember to:

Core Principles

1. **Understand Requirements First**
    - Throughput and latency needs
    - Delivery and ordering guarantees
    - Durability and retention
    - Scaling requirements
2. **Choose the Right System**

- Kafka for high throughput and replay
- RabbitMQ for complex routing
- SQS/SNS for AWS-based simplicity
- Consider operational complexity

3. **Design for Reliability**
   - Appropriate delivery guarantees
   - Implement idempotency
   - Use dead letter queues
   - Plan for failures
4. **Think About Scale**
   - Partition strategy
   - Consumer groups
   - Horizontal scaling
   - Performance tuning
5. **Plan Operations**
   - Monitoring and alerting
   - Capacity planning
   - Cost estimation
   - Team training

Key Trade-offs to Remember

```
Throughput vs Ordering:
- More partitions = higher throughput
- Global ordering = lower throughput
- Balance based on requirements

Delivery Guarantees:
- At-most-once: Fast but lossy
- At-least-once: Balanced, need idempotency
- Exactly-once: Slow but guaranteed

System Choice:
- Kafka: Complex but powerful
- SQS: Simple but limited
- RabbitMQ: Flexible but moderate scale

Synchronous vs Asynchronous:
- Sync: Immediate feedback, coupling
- Async: Decoupling, eventual consistency
```

Interview Success Framework

```
1. Ask Questions (5 min)
   - Scale requirements
   - Delivery guarantees
   - Ordering needs
   - Latency requirements

2. High-Level Design (10 min)
   - Choose messaging system
   - Draw architecture
   - Explain data flow

3. Deep Dive (20 min)
   - Partitioning strategy
   - Delivery guarantees
   - Scaling approach
   - Failure handling
   - Monitoring

4. Trade-offs (5 min)
   - Compare alternatives
   - Explain decisions
   - Discuss limitations
```

Final Tips

Remember:

- There's no perfect solution, only trade-offs
- Start simple, then iterate
- Consider operational complexity
- Think about failure scenarios
- Monitor everything
- Document decisions

Common Interview Topics:

- Order processing systems
- Real-time analytics pipelines
- Notification systems
- Event sourcing
- Log aggregation
- Task queues

Key Metrics:

- Consumer lag (most important!)
- Throughput (msgs/sec, MB/sec)
- Latency (P50, P95, P99)
- Error rate

- Availability

Good luck with your interviews!

Additional Resources

Books

- "Kafka: The Definitive Guide" by Neha Narkhede
- "Designing Data-Intensive Applications" by Martin Kleppmann
- "Enterprise Integration Patterns" by Gregor Hohpe

Online Resources

- Apache Kafka Documentation
- RabbitMQ Tutorials
- AWS SQS/SNS Documentation
- Confluent Blog (Kafka)
- Martin Fowler's Blog (Event-Driven Architecture)

Key Concepts to Study

- Event-driven architecture
- Event sourcing
- CQRS pattern
- Saga pattern
- Outbox pattern
- Change data capture (CDC)

Hands-on Practice

- Set up local Kafka cluster
- Build producer/consumer applications
- Experiment with partitioning
- Test failure scenarios
- Monitor with Kafka tools
- Try different messaging systems