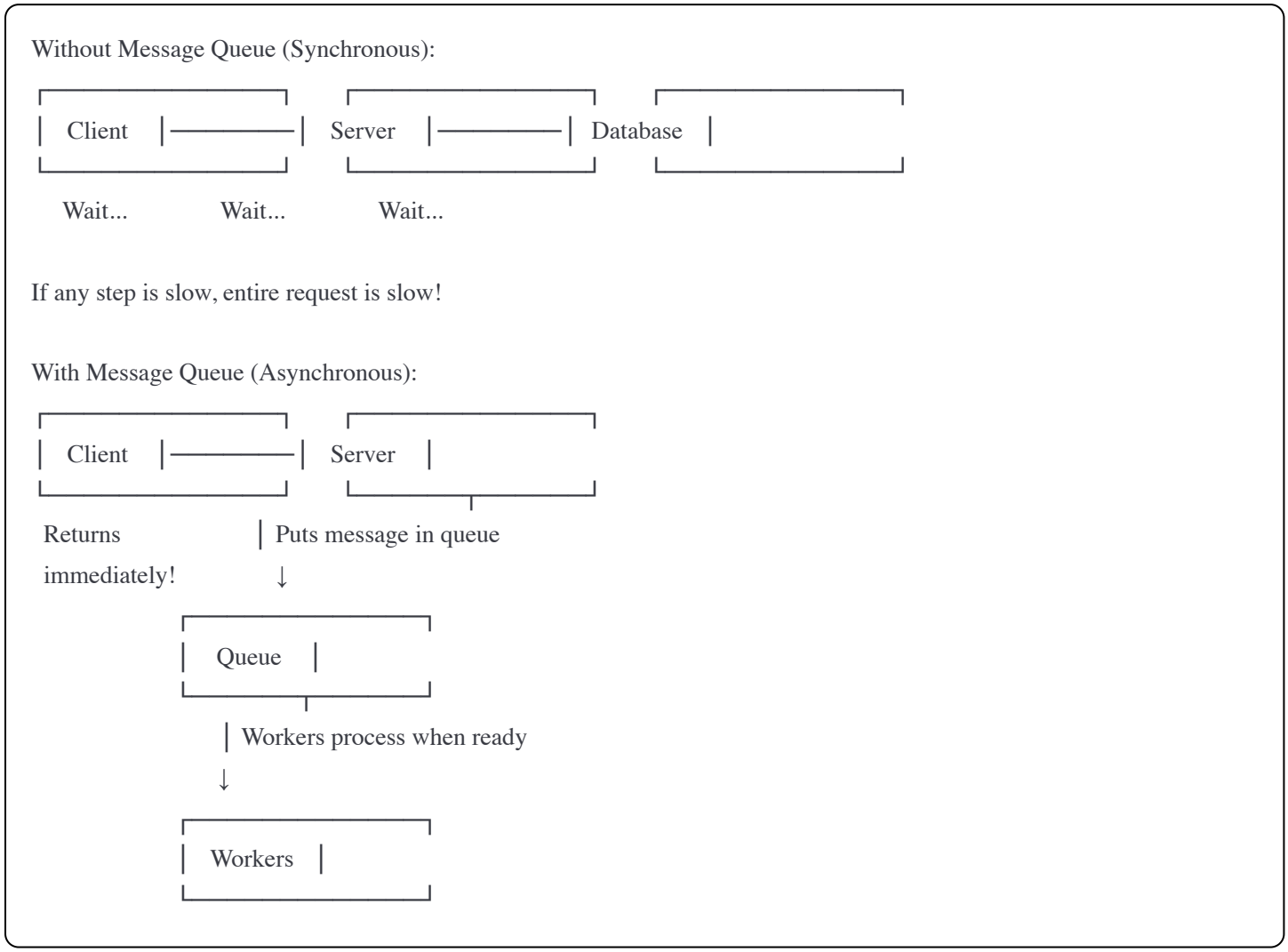
Chapter 10: Message Queues

Introduction: Why Message Queues?

Message queues enable **asynchronous communication** between services.

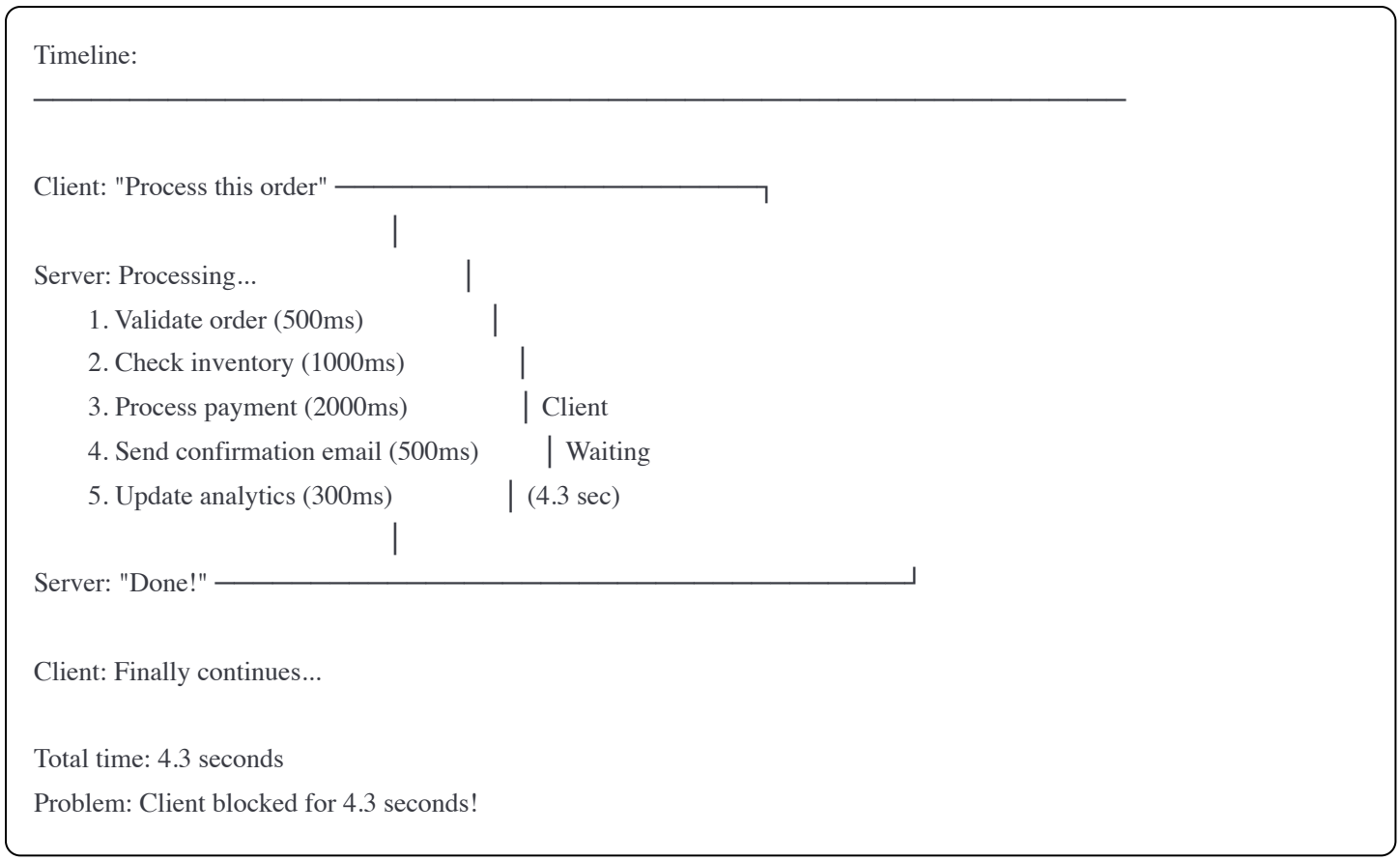


---

1. Synchronous vs Asynchronous Communication

Synchronous Communication

Concept: Caller waits for response before continuing.



Code Example:

```javascript
// Synchronous API call
app.post('/api/orders', async (req, res) => {
  try {
    // Client waits for ALL of this
    const order = await validateOrder(req.body);      // 500ms
    const inventory = await checkInventory(order);    // 1000ms
    const payment = await processPayment(order);      // 2000ms
    await sendConfirmationEmail(order);               // 500ms
    await updateAnalytics(order);                     // 300ms

    res.json({ success: true, orderId: order.id });
    // Total: 4.3 seconds before client gets response!

  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

Asynchronous Communication

Concept: Caller doesn't wait, work happens in background.

Timeline:
═══

Client: "Process this order" ──────┐
                                   │
Server: "Got it! Order ID: 123" ◄──┘ (50ms)

Client: Continues immediately!

Meanwhile, in background:

┌──┐
│ Worker 1: Check inventory (1000ms)   │
│ Worker 2: Process payment (2000ms)   │
│ Worker 3: Send email (500ms)   │
│ Worker 4: Update analytics (300ms)   │
└──┘

All happen in parallel!

Total client wait: 50ms (86x faster!)

Code Example:

javascript

```
const { Queue } = require('bull');
const orderQueue = new Queue('order-processing');

// API endpoint (returns immediately)
app.post('/api/orders', async (req, res) => {
  try {
    // Quick validation only
    const order = await validateOrderBasic(req.body);  // 50ms

    // Add to queue for background processing
    await orderQueue.add('process-order', {
      orderId: order.id,
      items: order.items,
      userId: order.userId
    });

    // Return immediately!
    res.json({
      success: true,
      orderId: order.id,
      message: 'Order is being processed'
    });

  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// Background worker (processes asynchronously)
orderQueue.process('process-order', async (job) => {
  const { orderId, items, userId } = job.data;

  // These happen in background, client already got response
  await checkInventory(items);          // 1000ms
  await processPayment(orderId);        // 2000ms
  await sendConfirmationEmail(userId);  // 500ms
  await updateAnalytics(orderId);       // 300ms

  console.log(`Order ${orderId} processed successfully`);
});
```

Comparison Table

Feature	Synchronous	Asynchronous
Response Time	Slow	Fast
Client Blocking	Yes	No
Error Handling	Immediate	Delayed
Retry Logic	Client	Queue/Worker
Scalability	Limited	High
Complexity	Simple	Complex
Debugging	Easy	Harder
Consistency	Strong	Eventual

When to use Synchronous:
✓ Need immediate response
✓ User must wait for result
✓ Simple operations
✓ Strong consistency required

Examples: Login, Payment confirmation, Search

When to use Asynchronous:
✓ Long-running tasks
✓ Can show "processing..." to user
✓ Background jobs
✓ High throughput needed

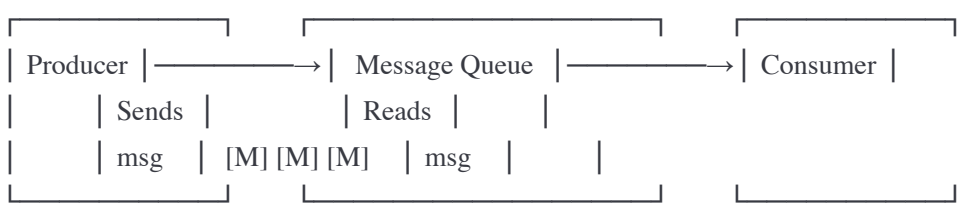
Examples: Email sending, Report generation, Image processing

---

2. Message Queue Fundamentals

Core Concepts

Basic Message Queue:



Components:
- Producer: Sends messages
- Queue: Stores messages

- Consumer: Processes messages
- Message: Data being transferred

How It Works

Step-by-Step Flow:

1. PRODUCE
   Producer: Creates message
   Message: { type: 'order', data: {...} }
   Producer: Sends to queue

2. STORE
   Queue: Receives message
   Queue: Persists to disk (durability)
   Queue: Acknowledges receipt to producer

3. CONSUME
   Consumer: Polls queue or notified
   Queue: Delivers message to consumer
   Consumer: Processes message

4. ACKNOWLEDGE
   Consumer: Sends ACK to queue
   Queue: Deletes message

If consumer fails before ACK:
   Queue: Re-delivers message to another consumer

Message Queue Benefits

1. DECOUPLING
   Services don't need to know about each other

   Without Queue:

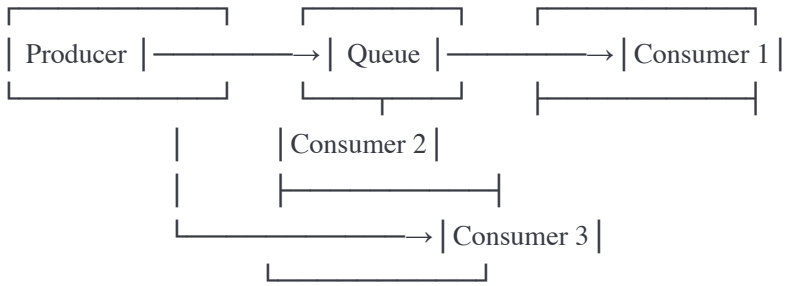
   A must know B's API, location

   With Queue:

A and B independent!

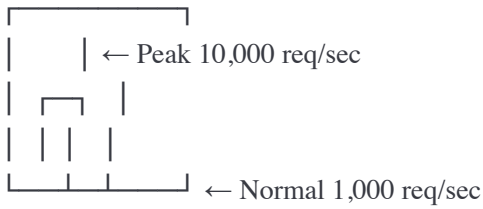
2. RELIABILITY
   Messages persist if consumer fails
   Automatic retry

3. SCALABILITY
   Add more consumers without changing producers



4. LOAD LEVELING
   Smooth out traffic spikes

   Traffic:



   Queue absorbs spike!
   Workers process at steady rate

5. ASYNC PROCESSING
   Return to user immediately
   Process in background

Message Structure

javascript

```
// Typical message format
{
  // Message ID (unique identifier)
  "id": "msg_123456789",

  // Timestamp (when created)
  "timestamp": "2024-01-20T10:00:00Z",

  // Message body (actual data)
  "body": {
    "type": "order_created",
    "orderId": "order_789",
    "userId": "user_123",
    "items": [
      { "productId": "prod_1", "quantity": 2 }
    ],
    "total": 99.99
  },

  // Metadata (attributes)
  "attributes": {
    "priority": "high",
    "source": "web_app",
    "version": "1.0"
  },

  // Delivery information
  "deliveryInfo": {
    "attempts": 1,
    "firstAttempt": "2024-01-20T10:00:00Z",
    "nextRetry": "2024-01-20T10:01:00Z"
  }
}
```

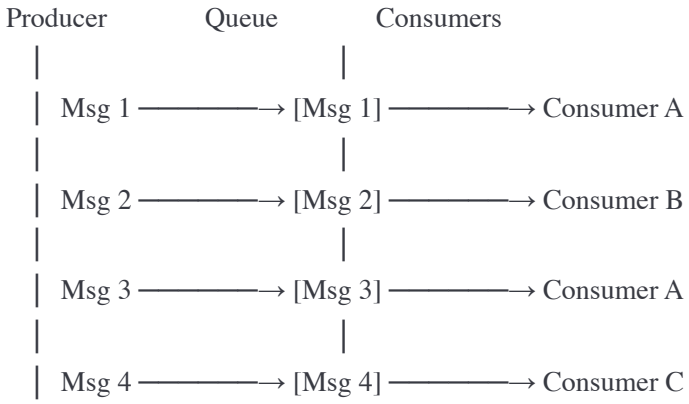
---

3. Queue vs Topic (Pub/Sub)

Queue (Point-to-Point)

Concept: One message → One consumer

Queue Model:



Characteristics:

- Each message consumed by ONE consumer only
- Load balanced across consumers
- Order preserved (usually)
- Competing consumers pattern

Use Cases:

- Task distribution (image processing)
- Job queues (background jobs)
- Load balancing (spread work)

Code Example (RabbitMQ Queue):

javascript

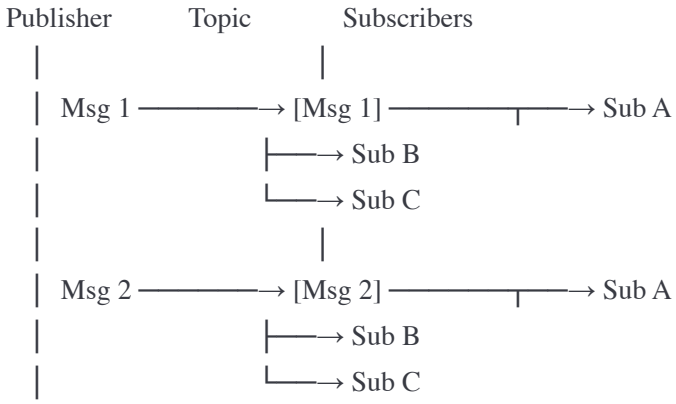
```
const amqp = require('amqplib');

// Producer
async function produceToQueue() {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();

  const queue = 'task_queue';
  await channel.assertQueue(queue, { durable: true });

  // Send messages
  for (let i = 1; i <= 10; i++) {
    const message = { taskId: i, data: `Task ${i}` };
    channel.sendToQueue(queue, Buffer.from(JSON.stringify(message)), {
      persistent: true
    });
    console.log(`Sent: Task ${i}`);
  }

  setTimeout(() => {
    connection.close();
  }, 500);
}

// Consumer 1
async function consumeFromQueue(consumerId) {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();

  const queue = 'task_queue';
  await channel.assertQueue(queue, { durable: true });

  // Fair dispatch (one message at a time)
  channel.prefetch(1);

  console.log(`Consumer ${consumerId} waiting for messages...`);

  channel.consume(queue, async (msg) => {
    const message = JSON.parse(msg.content.toString());
    console.log(`Consumer ${consumerId} received: ${message.data}`);

    // Simulate processing
    await new Promise(resolve => setTimeout(resolve, 1000));

    console.log(`Consumer ${consumerId} done with: ${message.data}`);
    channel.ack(msg);
```

```
  });
}

// Start multiple consumers
consumeFromQueue('A');
consumeFromQueue('B');
consumeFromQueue('C');

// Produce messages
produceToQueue();

// Output:
// Consumer A received: Task 1
// Consumer B received: Task 2
// Consumer C received: Task 3
// Consumer A received: Task 4 (Task 1 done)
// ...
// Each message processed by ONE consumer
```

Topic (Publish/Subscribe)

Concept: One message → Multiple subscribers

Pub/Sub Model:



Characteristics:

- Each message copied to ALL subscribers
- Fan-out pattern
- Subscribers are independent
- Each subscriber gets all messages

Use Cases:

- Event broadcasting (user signup)
- Real-time notifications
- Logging (multiple log processors)
- Cache invalidation (notify all servers)

Code Example (RabbitMQ Pub/Sub):

javascript

```
const amqp = require('amqplib');

// Publisher
async function publishEvent() {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();

  const exchange = 'user_events';
  await channel.assertExchange(exchange, 'fanout', { durable: false });

  // Publish event
  const event = {
    type: 'user_signup',
    userId: 123,
    email: 'user@example.com',
    timestamp: new Date().toISOString()
  };

  channel.publish(exchange, '', Buffer.from(JSON.stringify(event)));
  console.log('Published event:', event);

  setTimeout(() => connection.close(), 500);
}

// Subscriber (Email Service)
async function subscribeEmailService() {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();

  const exchange = 'user_events';
  await channel.assertExchange(exchange, 'fanout', { durable: false });

  // Create exclusive queue for this subscriber
  const q = await channel.assertQueue('', { exclusive: true });
  channel.bindQueue(q.queue, exchange, '');

  console.log('[Email Service] Waiting for events...');

  channel.consume(q.queue, (msg) => {
    const event = JSON.parse(msg.content.toString());
    console.log('[Email Service] Received:', event);
    console.log('[Email Service] Sending welcome email...');
  }, { noAck: true });
}

// Subscriber (Analytics Service)
```

```
async function subscribeAnalyticsService() {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();

  const exchange = 'user_events';
  await channel.assertExchange(exchange, 'fanout', { durable: false });

  const q = await channel.assertQueue('', { exclusive: true });
  channel.bindQueue(q.queue, exchange, '');

  console.log('[Analytics Service] Waiting for events...');

  channel.consume(q.queue, (msg) => {
    const event = JSON.parse(msg.content.toString());
    console.log('[Analytics Service] Received:', event);
    console.log('[Analytics Service] Updating metrics...');
  }, { noAck: true });
}

// Subscriber (Notification Service)
async function subscribeNotificationService() {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();

  const exchange = 'user_events';
  await channel.assertExchange(exchange, 'fanout', { durable: false });

  const q = await channel.assertQueue('', { exclusive: true });
  channel.bindQueue(q.queue, exchange, '');

  console.log('[Notification Service] Waiting for events...');

  channel.consume(q.queue, (msg) => {
    const event = JSON.parse(msg.content.toString());
    console.log('[Notification Service] Received:', event);
    console.log('[Notification Service] Sending push notification...');
  }, { noAck: true });
}

// Start all subscribers
subscribeEmailService();
subscribeAnalyticsService();
subscribeNotificationService();

// Publish event
setTimeout(() => publishEvent(), 1000);
```

```
// Output:
// [Email Service] Received: {user_signup...}
// [Analytics Service] Received: {user_signup...}
// [Notification Service] Received: {user_signup...}
// ALL subscribers receive the same message!
```

Queue vs Topic Comparison

Feature	Queue	Topic
Delivery	One consumer	All subscribers
Pattern	Point-to-point	Pub/Sub
Load Balancing	Yes	No
Message Copy	No	Yes (per sub)
Use Case	Task queue	Event broadcast
Example:		
3 consumers 10 messages	Each gets 33% of messages	Each gets 100% of messages

Real-World Example:

E-commerce Order Processing:

QUEUE (Order Processing):
Order created → Queue → One worker processes order
(payment, inventory, shipping)
Purpose: Distribute work, only process once

TOPIC (Order Events):
Order created → Topic → Email service (send confirmation)
→ Analytics (track sale)
→ Warehouse (prepare shipment)
→ Notification (push notification)
Purpose: Notify all interested services

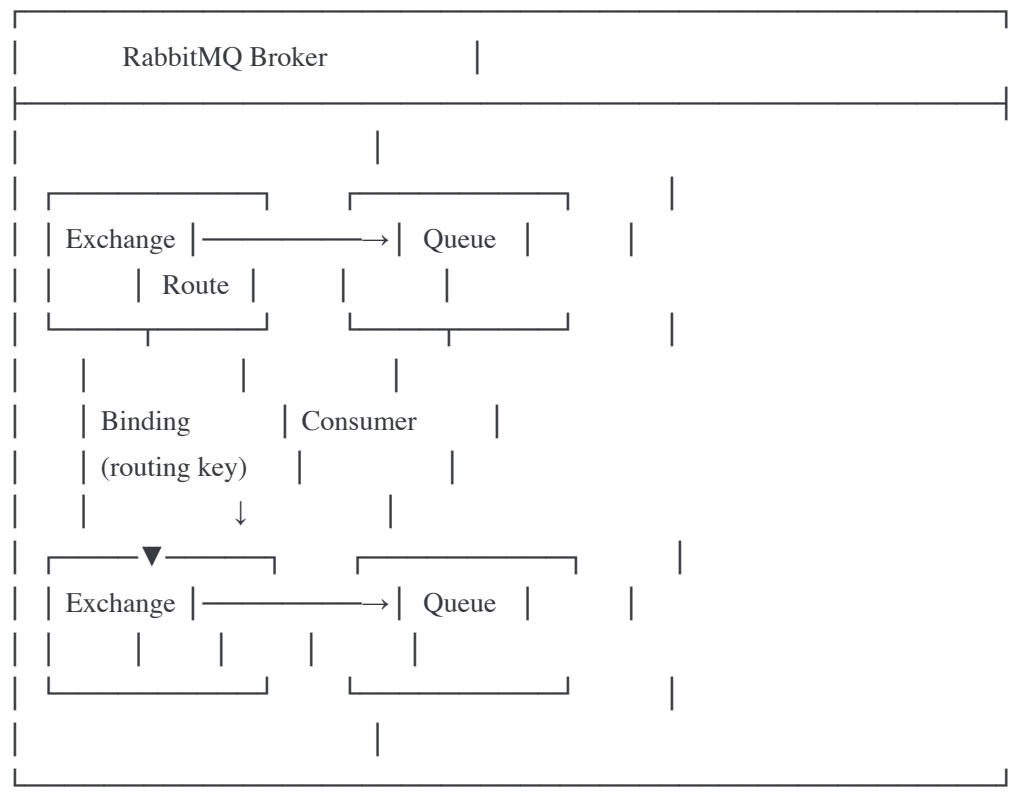
---

4. Popular Message Queues

RabbitMQ

Type: Traditional message broker **Protocol:** AMQP (Advanced Message Queuing Protocol)

Architecture:



Components:

- Producer: Sends to exchange
- Exchange: Routes messages
- Binding: Rules for routing
- Queue: Stores messages
- Consumer: Receives from queue

Exchange Types:

javascript

```
// 1. Direct Exchange (exact match)
// Use case: Send email to specific service

await channel.assertExchange('direct_logs', 'direct');
channel.publish('direct_logs', 'error', Buffer.from('Error message'));
// Only queues bound with routing key 'error' receive it

// 2. Fanout Exchange (broadcast)
// Use case: Broadcast event to all services

await channel.assertExchange('user_events', 'fanout');
channel.publish('user_events', '', Buffer.from('User signup'));
// ALL bound queues receive it

// 3. Topic Exchange (pattern match)
// Use case: Flexible routing with wildcards

await channel.assertExchange('logs', 'topic');
channel.publish('logs', 'app.error.critical', Buffer.from('Critical error'));
// Queues bound to 'app.error.*' or 'app.*.critical' receive it

// Patterns:
// * (star) - matches one word
// # (hash) - matches zero or more words

// 4. Headers Exchange (match on headers)
// Use case: Route based on message attributes

await channel.assertExchange('tasks', 'headers');
channel.publish('tasks', '', Buffer.from('Task'), {
  headers: { priority: 'high', type: 'email' }
});
// Matches queues bound with matching headers
```

Complete RabbitMQ Example:

javascript

```
const amqp = require('amqplib');

class RabbitMQService {
  constructor() {
    this.connection = null;
    this.channel = null;
  }

  async connect() {
    this.connection = await amqp.connect('amqp://localhost');
    this.channel = await this.connection.createChannel();
    console.log('Connected to RabbitMQ');
  }

  // Send to queue
  async sendToQueue(queue, message) {
    await this.channel.assertQueue(queue, { durable: true });

    this.channel.sendToQueue(
      queue,
      Buffer.from(JSON.stringify(message)),
      { persistent: true }
    );

    console.log('Sent to queue:', queue);
  }

  // Consume from queue
  async consumeFromQueue(queue, callback) {
    await this.channel.assertQueue(queue, { durable: true });
    this.channel.prefetch(1); // Process one message at a time

    this.channel.consume(queue, async (msg) => {
      if (msg) {
        const content = JSON.parse(msg.content.toString());

        try {
          await callback(content);
          this.channel.ack(msg); // Acknowledge success
        } catch (error) {
          console.error('Error processing message:', error);
          this.channel.nack(msg, false, true); // Requeue on error
        }
      }
    });
  }
```

```
  // Publish to topic
  async publish(exchange, routingKey, message) {
    await this.channel.assertExchange(exchange, 'topic', { durable: true });

    this.channel.publish(
      exchange,
      routingKey,
      Buffer.from(JSON.stringify(message)),
      { persistent: true }
    );

    console.log('Published to exchange:', exchange);
  }

  // Subscribe to topic
  async subscribe(exchange, pattern, callback) {
    await this.channel.assertExchange(exchange, 'topic', { durable: true });

    const q = await this.channel.assertQueue('', { exclusive: true });
    await this.channel.bindQueue(q.queue, exchange, pattern);

    this.channel.consume(q.queue, async (msg) => {
      if (msg) {
        const content = JSON.parse(msg.content.toString());
        await callback(content);
      }
    }, { noAck: true });
  }

  async close() {
    await this.channel.close();
    await this.connection.close();
  }
}

// Usage
async function main() {
  const mq = new RabbitMQService();
  await mq.connect();

  // Queue example
  await mq.sendToQueue('orders', { orderId: 123, total: 99.99 });

  await mq.consumeFromQueue('orders', async (order) => {
    console.log('Processing order:', order);
    // Process order...
```

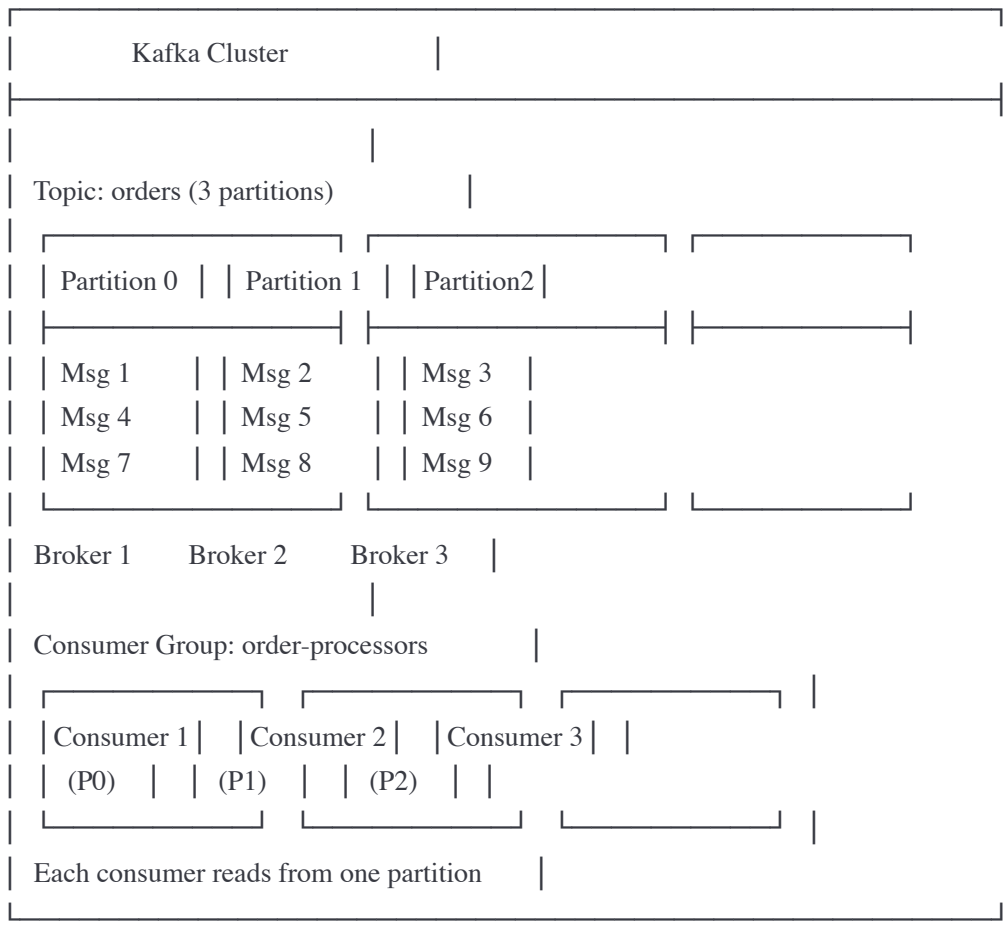
```
  });

  // Topic example
  await mq.publish('events', 'user.created', { userId: 456 });

  await mq.subscribe('events', 'user.*', async (event) => {
    console.log('User event:', event);
  });
}
```

Apache Kafka

Type: Distributed streaming platform **Protocol:** Custom binary protocol

Architecture:

Kafka Cluster:



Key Concepts:

- Topic: Category of messages
- Partition: Ordered log within topic
- Broker: Kafka server
- Consumer Group: Cooperating consumers
- Offset: Position in partition

Kafka Characteristics:

1. APPEND-ONLY LOG
   Messages never deleted immediately
   Retention: time-based or size-based

2. PARTITIONING
   Topics split into partitions
   Parallelism = number of partitions

3. REPLICATION
   Each partition replicated across brokers
   Leader handles reads/writes
   Followers replicate

4. CONSUMER GROUPS
   Each partition consumed by one consumer in group
   Enables parallel processing

5. OFFSET TRACKING
   Consumer tracks position in log
   Can replay messages (rewind offset)

Kafka Code Example:

javascript

```
const { Kafka } = require('kafkajs');

// Initialize Kafka client
const kafka = new Kafka({
  clientId: 'my-app',
  brokers: ['localhost:9092']
});

// Producer
const producer = kafka.producer();

async function produceMessages() {
  await producer.connect();

  // Send messages
  await producer.send({
    topic: 'orders',
    messages: [
      {
        key: 'user-123',  // Messages with same key go to same partition
        value: JSON.stringify({
          orderId: 1,
          userId: 123,
          total: 99.99
        })
      },
      {
        key: 'user-456',
        value: JSON.stringify({
          orderId: 2,
          userId: 456,
          total: 149.99
        })
      }
    ]
  });

  console.log('Messages sent to Kafka');
  await producer.disconnect();
}

// Consumer
const consumer = kafka.consumer({ groupId: 'order-processors' });

async function consumeMessages() {
  await consumer.connect();
```

```
  await consumer.subscribe({ topic: 'orders', fromBeginning: true });

  await consumer.run({
    eachMessage: async ({ topic, partition, message }) => {
      const order = JSON.parse(message.value.toString());

      console.log({
        partition,
        offset: message.offset,
        key: message.key.toString(),
        order
      });

      // Process order
      await processOrder(order);

      // Offset automatically committed
    }
  });
}

// Advanced: Manual offset management
async function consumeWithManualCommit() {
  await consumer.connect();
  await consumer.subscribe({ topic: 'orders', fromBeginning: false });

  await consumer.run({
    autoCommit: false,  // Manual commit
    eachMessage: async ({ topic, partition, message }) => {
      const order = JSON.parse(message.value.toString());

      try {
        await processOrder(order);

        // Manually commit offset only after successful processing
        await consumer.commitOffsets([{
          topic,
          partition,
          offset: (parseInt(message.offset) + 1).toString()
        }]);

        console.log('Order processed and committed');

      } catch (error) {
        console.error('Error processing order:', error);
        // Don't commit - message will be reprocessed
      }
```

```
    }
  });
}

// Producer with partitioning
async function produceWithCustomPartitioning() {
  await producer.connect();

  await producer.send({
    topic: 'orders',
    messages: [
      {
        key: 'user-123',
        value: JSON.stringify({ orderId: 1 }),
        partition: 0  // Explicitly specify partition
      }
    ]
  });
}

// Batch production (high throughput)
async function produceBatch() {
  await producer.connect();

  const messages = [];
  for (let i = 0; i < 1000; i++) {
    messages.push({
      key: `user-${i}`,
      value: JSON.stringify({ orderId: i })
    });
  }

  await producer.send({
    topic: 'orders',
    messages
  });

  console.log('Sent 1000 messages in batch');
}

async function processOrder(order) {
  // Simulate processing
  console.log('Processing order:', order.orderId);
}
```

Amazon SQS (Simple Queue Service)

Type: Managed queue service **Protocol:** HTTP/HTTPS

Queue Types:

1. STANDARD QUEUE
   - Unlimited throughput
   - At-least-once delivery (may duplicate)
   - Best-effort ordering (not guaranteed)
   - Use case: High throughput, duplicates ok

2. FIFO QUEUE
   - Up to 3,000 messages/sec (300 with batching off)
   - Exactly-once delivery (no duplicates)
   - Strict ordering (FIFO)
   - Use case: Order matters, no duplicates

SQS Code Example:

javascript

```
const AWS = require('aws-sdk');
const sqs = new AWS.SQS({ region: 'us-east-1' });

// Send message
async function sendMessage(queueUrl, message) {
  const params = {
    QueueUrl: queueUrl,
    MessageBody: JSON.stringify(message),
    MessageAttributes: {
      'Priority': {
        DataType: 'String',
        StringValue: 'High'
      },
      'Timestamp': {
        DataType: 'Number',
        StringValue: Date.now().toString()
      }
    },
    // For FIFO queues:
    // MessageGroupId: 'order-group',  // Messages in same group are ordered
    // MessageDeduplicationId: 'unique-id'  // Prevent duplicates
  };

  const result = await sqs.sendMessage(params).promise();
  console.log('Message sent:', result.MessageId);
  return result;
}

// Send batch (more efficient)
async function sendMessageBatch(queueUrl, messages) {
  const entries = messages.map((msg, i) => ({
    Id: `msg-${i}`,
    MessageBody: JSON.stringify(msg)
  }));

  const params = {
    QueueUrl: queueUrl,
    Entries: entries
  };

  const result = await sqs.sendMessageBatch(params).promise();
  console.log(`Sent ${result.Successful.length} messages`);
  return result;
}

// Receive and process messages
```

```
async function receiveMessages(queueUrl) {
  const params = {
    QueueUrl: queueUrl,
    MaxNumberOfMessages: 10,  // Receive up to 10 at once
    WaitTimeSeconds: 20,      // Long polling (reduce empty responses)
    VisibilityTimeout: 30     // Hide message for 30 sec while processing
  };

  const result = await sqs.receiveMessage(params).promise();

  if (!result.Messages) {
    console.log('No messages');
    return;
  }

  for (const message of result.Messages) {
    try {
      const body = JSON.parse(message.Body);
      console.log('Processing message:', body);

      // Process message
      await processMessage(body);

      // Delete message after successful processing
      await sqs.deleteMessage({
        QueueUrl: queueUrl,
        ReceiptHandle: message.ReceiptHandle
      }).promise();

      console.log('Message deleted');

    } catch (error) {
      console.error('Error processing message:', error);

      // Message will become visible again after VisibilityTimeout
      // and can be retried
    }
  }
}

// Continuous polling (worker)
async function startWorker(queueUrl) {
  console.log('Worker started');

  while (true) {
    try {
      await receiveMessages(queueUrl);
```

```
    } catch (error) {
      console.error('Worker error:', error);
      await new Promise(resolve => setTimeout(resolve, 5000)); // Wait 5 sec
    }
  }
}

// Change message visibility (extend processing time)
async function extendVisibility(queueUrl, receiptHandle) {
  await sqs.changeMessageVisibility({
    QueueUrl: queueUrl,
    ReceiptHandle: receiptHandle,
    VisibilityTimeout: 60  // Extend to 60 more seconds
  }).promise();
}

async function processMessage(message) {
  // Simulate processing
  console.log('Processing:', message);
}

// Usage
const queueUrl = 'https://sqs.us-east-1.amazonaws.com/123456789/my-queue';

// Send message
sendMessage(queueUrl, { orderId: 123, total: 99.99 });

// Start worker
startWorker(queueUrl);
```

Message Queue Comparison

```
┌──────────────────┬──────────────┬──────────────┬──────────────┐
│ Feature          │ RabbitMQ     │ Kafka        │ SQS          │
├──────────────────┼──────────────┼──────────────┼──────────────┤
│ Type             │ Broker       │ Log          │ Managed      │
│ Protocol         │ AMQP         │ Custom       │ HTTP         │
│ Throughput       │ 10K msg/s    │ 1M+ msg/s    │ Unlimited    │
│ Ordering         │ Per queue    │ Per part.    │ FIFO only    │
│ Retention        │ Until ACK    │ Time-based   │ 14 days      │
│ Replay           │ No           │ Yes          │ No           │
│ Complexity       │ Medium       │ High         │ Low          │
├──────────────────┼──────────────┼──────────────┼──────────────┤
│ Best For         │ Task queue   │ Streaming    │ AWS          │
│                  │ RPC          │ Event log    │ Simple       │
```

	Routing	Big data	Managed

When to use:
- RabbitMQ: Complex routing, traditional queues
- Kafka: High throughput, event streaming, replay
- SQS: AWS ecosystem, managed service, simple

---

5. Message Ordering and Delivery Guarantees

Delivery Guarantees

1. AT-MOST-ONCE (Fire and forget)
   Producer → Queue (no confirmation)
   Queue → Consumer (no acknowledgment)

   Risk: Messages can be lost
   Benefit: Highest performance
   Use case: Metrics, logs (some loss acceptable)

2. AT-LEAST-ONCE (Retry on failure)
   Producer → Queue → Confirmation
   Queue → Consumer → ACK
   If no ACK: Retry

   Risk: Duplicates possible
   Benefit: No loss
   Use case: Most applications (handle duplicates)

3. EXACTLY-ONCE (Deduplication)
   Producer → Queue (idempotent)
   Queue → Consumer (transactional)
   Deduplication ensures no duplicates

   Risk: More complex
   Benefit: No loss, no duplicates
   Use case: Financial transactions, payments

Message Ordering

Problem: Out-of-order delivery

Scenario: Update user profile

Message 1: Set name = "John" (sent at 10:00:00)
Message 2: Set name = "John Doe" (sent at 10:00:01)

Without ordering:
Message 2 processed first → name = "John Doe"
Message 1 processed second → name = "John" (WRONG!)

Final state: "John" (should be "John Doe")

Solutions:

1. SINGLE PARTITION/QUEUE
   All messages to same partition

   Pros: Guaranteed order
   Cons: No parallelism

2. PARTITION BY KEY
   Same key → Same partition

   Example: Kafka
   Key = user_id
   All updates for user_123 → Partition 0
   All updates for user_456 → Partition 1

   Pros: Ordered per user, still parallel
   Cons: Hotspots if some keys popular

3. SEQUENCE NUMBERS
   Add sequence to messages
   Consumer reorders before processing

   Message: { seq: 1, data: "..." }
   Message: { seq: 2, data: "..." }

   Consumer: Sort by seq, process in order

   Pros: Flexible
   Cons: Consumer complexity

4. TIMESTAMPS
   Similar to sequence numbers
   Use timestamp to order

   Risk: Clock skew between servers

Code Example (Ordered Processing):

```javascript
```

```
// Kafka: Ordered by partition key
await producer.send({
  topic: 'user-updates',
  messages: [
    {
      key: 'user-123',  // All messages with same key → same partition
      value: JSON.stringify({ action: 'set_name', value: 'John' })
    },
    {
      key: 'user-123',  // Same key → same partition → ordered!
      value: JSON.stringify({ action: 'set_name', value: 'John Doe' })
    }
  ]
});

// Consumer processes in order within partition
await consumer.run({
  eachMessage: async ({ partition, message }) => {
    const update = JSON.parse(message.value);
    // Messages for same key processed in order
    await applyUpdate(update);
  }
});

// Sequence number approach
class OrderedMessageProcessor {
  constructor() {
    this.buffers = new Map(); // userId -> message buffer
    this.nextSeq = new Map();  // userId -> expected sequence
  }

  async processMessage(message) {
    const { userId, seq, data } = message;

    if (!this.nextSeq.has(userId)) {
      this.nextSeq.set(userId, 0);
      this.buffers.set(userId, []);
    }

    const expected = this.nextSeq.get(userId);

    if (seq === expected) {
      // In order! Process immediately
      await this.process(data);
      this.nextSeq.set(userId, seq + 1);
```

```
      // Process any buffered messages now in order
      await this.processBuffered(userId);

    } else if (seq > expected) {
      // Future message, buffer it
      this.buffers.get(userId).push(message);
      console.log(`Buffered message ${seq}, expecting ${expected}`);

    } else {
      // Old message (duplicate or out of order)
      console.log(`Ignoring old message ${seq}`);
    }
  }

  async processBuffered(userId) {
    const buffer = this.buffers.get(userId);
    let expected = this.nextSeq.get(userId);

    // Sort buffer
    buffer.sort((a, b) => a.seq - b.seq);

    // Process all consecutive messages
    while (buffer.length > 0 && buffer[0].seq === expected) {
      const msg = buffer.shift();
      await this.process(msg.data);
      expected++;
      this.nextSeq.set(userId, expected);
    }
  }

  async process(data) {
    console.log('Processing:', data);
    // Actually process the data
  }
}

// Usage
const processor = new OrderedMessageProcessor();

// Messages arrive out of order
await processor.processMessage({ userId: 'user-123', seq: 0, data: 'First' });
await processor.processMessage({ userId: 'user-123', seq: 2, data: 'Third' });
// Message 2 buffered (expecting seq 1)

await processor.processMessage({ userId: 'user-123', seq: 1, data: 'Second' });
// Now processes: Second, then Third (from buffer)
```

6. Dead Letter Queues (DLQ)

Concept: Queue for messages that couldn't be processed.

Normal Flow:
Producer → Queue → Consumer (Success) ✓

Failed Flow:
Producer → Queue → Consumer (Fail)
↓ Retry 1 (Fail)
↓ Retry 2 (Fail)
↓ Retry 3 (Fail)
↓ Max retries exceeded
→ Dead Letter Queue

Dead Letter Queue holds:
- Messages that failed multiple times
- Malformed messages
- Messages causing exceptions

Why Use DLQ?

Problems without DLQ:
1. Failed messages block queue
2. Infinite retry loops
3. Lost messages (if discarded)
4. No visibility into failures

Benefits with DLQ:
1. Failed messages don't block others
2. Can investigate failures
3. Can manually retry after fixing
4. Alerts on DLQ activity

DLQ Implementation

RabbitMQ:

javascript

```
const amqp = require('amqplib');

async function setupDLQ() {
  const connection = await amqp.connect('amqp://localhost');
  const channel = await connection.createChannel();

  // Main queue with DLQ configuration
  const mainQueue = 'orders';
  const dlqExchange = 'dlx';
  const dlq = 'orders.dlq';

  // Create DLQ exchange
  await channel.assertExchange(dlqExchange, 'direct', { durable: true });

  // Create DLQ
  await channel.assertQueue(dlq, { durable: true });
  await channel.bindQueue(dlq, dlqExchange, mainQueue);

  // Create main queue with DLQ configuration
  await channel.assertQueue(mainQueue, {
    durable: true,
    arguments: {
      'x-dead-letter-exchange': dlqExchange,
      'x-dead-letter-routing-key': mainQueue,
      'x-message-ttl': 60000  // Messages expire after 60 sec if not consumed
    }
  });

  // Consumer with retry logic
  channel.consume(mainQueue, async (msg) => {
    const message = JSON.parse(msg.content.toString());
    const retryCount = (msg.properties.headers['x-retry-count'] || 0);
    const maxRetries = 3;

    try {
      // Process message
      await processOrder(message);
      channel.ack(msg);  // Success

    } catch (error) {
      console.error('Error processing message:', error);

      if (retryCount < maxRetries) {
        // Retry: Reject and requeue with incremented counter
        console.log(`Retry ${retryCount + 1}/${maxRetries}`);
```

```
        channel.nack(msg, false, false);  // Don't requeue (will go to DLQ)

        // Republish with retry count
        channel.sendToQueue(mainQueue, msg.content, {
          headers: {
            'x-retry-count': retryCount + 1
          },
          persistent: true
        });

      } else {
        // Max retries exceeded - goes to DLQ
        console.log('Max retries exceeded, sending to DLQ');
        channel.nack(msg, false, false);
      }
    }
  });

  // Monitor DLQ
  channel.consume(dlq, async (msg) => {
    const message = JSON.parse(msg.content.toString());
    console.log('Message in DLQ:', message);

    // Alert ops team
    await alertOps('Message failed permanently', message);

    // Store for investigation
    await storeFailed(message);

    channel.ack(msg);
  }, { noAck: false });
}

async function processOrder(order) {
  // Simulate processing that might fail
  if (Math.random() < 0.3) {
    throw new Error('Processing failed');
  }
  console.log('Order processed:', order);
}

async function alertOps(subject, data) {
  console.log('ALERT:', subject, data);
  // Send email, Slack message, etc.
}

async function storeFailed(message) {
```

```
  // Store in database for investigation
  console.log('Storing failed message:', message);
}
```

Kafka (using separate topic as DLQ):

```
javascript
```

```
const { Kafka } = require('kafkajs');

const kafka = new Kafka({
  clientId: 'my-app',
  brokers: ['localhost:9092']
});

const producer = kafka.producer();
const consumer = kafka.consumer({ groupId: 'order-processors' });

async function consumeWithDLQ() {
  await producer.connect();
  await consumer.connect();
  await consumer.subscribe({ topic: 'orders', fromBeginning: true });

  await consumer.run({
    eachMessage: async ({ topic, partition, message }) => {
      const order = JSON.parse(message.value.toString());
      const retryCount = parseInt(message.headers?.retryCount || 0);
      const maxRetries = 3;

      try {
        await processOrder(order);
        console.log('Order processed successfully');

      } catch (error) {
        console.error('Error processing order:', error);

        if (retryCount < maxRetries) {
          // Retry: Send back to same topic with retry count
          console.log(`Retry ${retryCount + 1}/${maxRetries}`);

          await producer.send({
            topic: 'orders',
            messages: [{
              key: message.key,
              value: message.value,
              headers: {
                retryCount: (retryCount + 1).toString(),
                originalTimestamp: message.timestamp,
                error: error.message
              }
            }]
          });

        } else {
```

```
        // Send to DLQ
        console.log('Max retries exceeded, sending to DLQ');

        await producer.send({
          topic: 'orders.dlq',
          messages: [{
            key: message.key,
            value: message.value,
            headers: {
              retryCount: retryCount.toString(),
              originalTimestamp: message.timestamp,
              finalError: error.message,
              originalTopic: topic,
              originalPartition: partition.toString()
            }
          }]
        });
      }
    }
  }
});
}

// Monitor DLQ
async function monitorDLQ() {
  const dlqConsumer = kafka.consumer({ groupId: 'dlq-monitor' });
  await dlqConsumer.connect();
  await dlqConsumer.subscribe({ topic: 'orders.dlq', fromBeginning: true });

  await dlqConsumer.run({
    eachMessage: async ({ message }) => {
      const order = JSON.parse(message.value.toString());
      const error = message.headers.finalError;

      console.log('DLQ Message:', {
        order,
        error,
        retryCount: message.headers.retryCount
      });

      // Alert and store
      await alertOps('Order failed permanently', { order, error });
      await storeInDatabase(order, error);
    }
  });
}
```

```
consumeWithDLQ();
monitorDLQ();
```

AWS SQS DLQ:

javascript

```
const AWS = require('aws-sdk');
const sqs = new AWS.SQS({ region: 'us-east-1' });

// Create DLQ
async function createDLQ() {
  const dlqResult = await sqs.createQueue({
    QueueName: 'orders-dlq'
  }).promise();

  const dlqUrl = dlqResult.QueueUrl;

  // Get DLQ ARN
  const dlqAttrs = await sqs.getQueueAttributes({
    QueueUrl: dlqUrl,
    AttributeNames: ['QueueArn']
  }).promise();

  const dlqArn = dlqAttrs.Attributes.QueueArn;

  // Create main queue with DLQ
  const mainResult = await sqs.createQueue({
    QueueName: 'orders',
    Attributes: {
      'RedrivePolicy': JSON.stringify({
        deadLetterTargetArn: dlqArn,
        maxReceiveCount: '3'  // After 3 receive attempts, send to DLQ
      })
    }
  }).promise();

  return { mainQueueUrl: mainResult.QueueUrl, dlqUrl };
}

// Process messages with automatic DLQ
async function processWithDLQ(queueUrl) {
  while (true) {
    const result = await sqs.receiveMessage({
      QueueUrl: queueUrl,
      MaxNumberOfMessages: 10,
      WaitTimeSeconds: 20,
      VisibilityTimeout: 30
    }).promise();

    if (!result.Messages) continue;

    for (const message of result.Messages) {
```

```
    try {
      const order = JSON.parse(message.Body);
      await processOrder(order);

      // Success - delete message
      await sqs.deleteMessage({
        QueueUrl: queueUrl,
        ReceiptHandle: message.ReceiptHandle
      }).promise();

    } catch (error) {
      console.error('Processing failed:', error);
      // Don't delete - message will be retried
      // After 3 failed receives, SQS automatically moves to DLQ
    }
  }
 }
}

// Monitor DLQ
async function monitorDLQ(dlqUrl) {
 while (true) {
  const result = await sqs.receiveMessage({
    QueueUrl: dlqUrl,
    MaxNumberOfMessages: 10,
    WaitTimeSeconds: 20
  }).promise();

  if (result.Messages) {
    for (const message of result.Messages) {
      const order = JSON.parse(message.Body);
      console.log('Message in DLQ:', order);

      // Alert operations
      await alertOps('Order processing failed permanently', order);

      // After investigation, can manually reprocess or delete
      await sqs.deleteMessage({
        QueueUrl: dlqUrl,
        ReceiptHandle: message.ReceiptHandle
      }).promise();
    }
  }

  await new Promise(resolve => setTimeout(resolve, 60000)); // Check every minute
```

```
  }
}
```

Key Takeaways

1. **Sync vs Async:**
   - Sync: Wait for response, simple, slow
   - Async: Return immediately, complex, fast
   - Use async for long-running tasks
2. **Message Queues:**
   - Decouple services
   - Scale independently
   - Handle failures gracefully
   - Load leveling
3. **Queue vs Topic:**
   - Queue: One consumer (load balance)
   - Topic: All subscribers (broadcast)
   - Use both in same system
4. **Message Systems:**
   - RabbitMQ: Flexible routing, traditional
   - Kafka: High throughput, event log
   - SQS: Managed, AWS-native
5. **Ordering & Guarantees:**
   - At-most-once: Fast, lossy
   - At-least-once: Duplicates ok
   - Exactly-once: Complex, expensive
   - Ordering: Partition by key
6. **Dead Letter Queues:**
   - Handle permanent failures
   - Don't lose messages
   - Alert on issues

- Enable investigation

Practice Problems

1. Design an order processing system with message queues. Which queue type and why?
2. How would you ensure exactly-once processing of payments?
3. Design a notification system that sends emails, push notifications, and SMS. Queue or topic?
4. Calculate: If processing takes 100ms and you have 10,000 messages, how many consumers needed to process in 1 minute?

Ready to continue with more chapters?