Docker Message-Oriented Architecture

CS 404: Distributed Systems
Fall 2025
Professor: Amina Mseddi
Lab Instructor: Selim Masmoudi

April 21, 2025

Abstract

This report documents the implementation of a Docker Message-Oriented Architecture for the CS 404 Distributed Systems course. The project enhances a three-tier application (frontend, backend, database) with RabbitMQ messaging patterns to create a more resilient, loosely coupled system using asynchronous communication. The implementation focuses on the Producer/Consumer pattern for processing background operations in a Retro Gaming Catalog application.

Contents

1 Introduction

1.1 Project Overview

This project enhances a three-tier Retro Gaming Catalog application by integrating RabbitMQ message patterns to improve communication between services. The original application consisted of a frontend (HTML/CSS/JavaScript), backend (Node.js with Express), and database (MySQL). The enhancement transforms this architecture into a more resilient, loosely coupled system using asynchronous messaging.

1.2 Learning Objectives

The implementation addresses the following learning objectives:

- Application of RabbitMQ messaging patterns in a real-world distributed application
- Implementation of asynchronous communication between services
- Ensuring resilience through proper message handling and error recovery
- Containerization of a complex distributed system
- Documentation and demonstration of distributed system architecture

2 System Architecture

2.1 Architecture Overview

The enhanced system architecture implements a Producer/Consumer pattern using RabbitMQ for asynchronous message processing. The architecture consists of the following components:

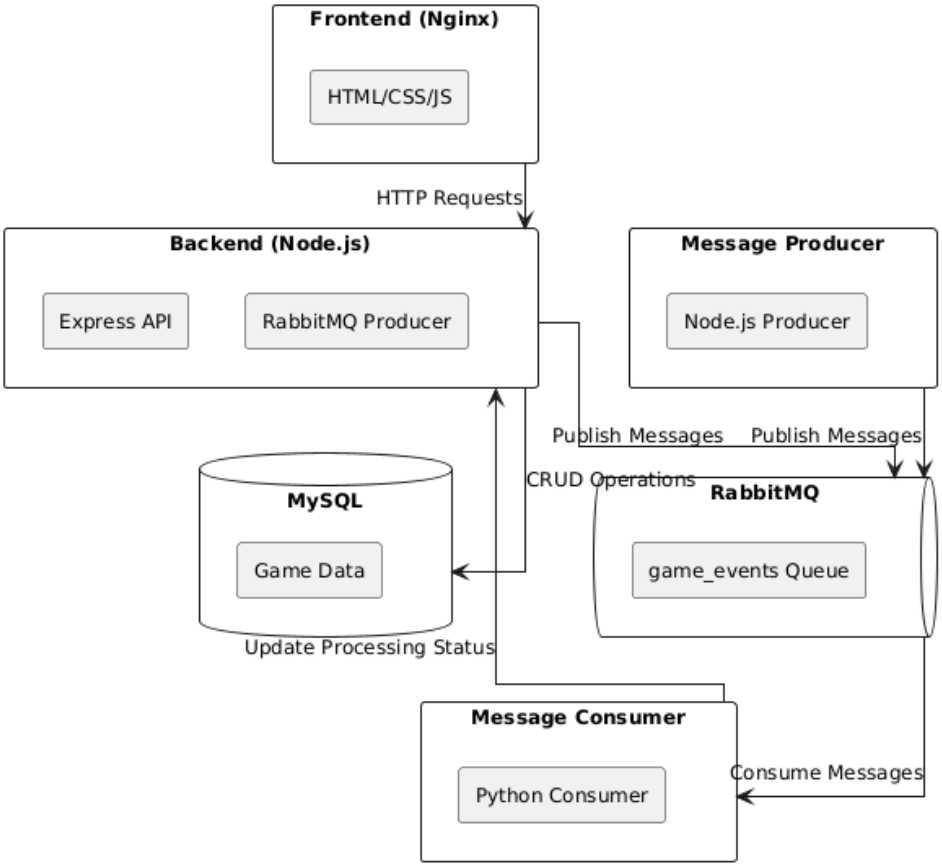

Figure 1: System Architecture Diagram

2.2 Component Description

- **Frontend (Nginx)**: Serves static HTML, CSS, and JavaScript files. Provides the user interface for the Retro Gaming Catalog.
- **Backend (Node.js)**: Implements the REST API for CRUD operations on games. Acts as a message producer, sending events to RabbitMQ when games are added, updated, or deleted.
- **Database (MySQL)**: Stores game data, including processing status.
- **RabbitMQ**: Message broker that facilitates asynchronous communication between services.
- **Message Producer**: Standalone service that can send custom messages to RabbitMQ.
- **Message Consumer**: Python service that processes messages from RabbitMQ and updates game processing status.

3 Implementation Details

3.1 RabbitMQ Integration

The implementation uses the Producer/Consumer pattern with RabbitMQ for asynchronous message processing. When a game is added, updated, or deleted in the Retro Gaming Catalog, the backend sends a message to RabbitMQ. The message consumer service processes these messages and updates the game's processing status.

3.1.1 Message Flow

1. When a game is added, updated, or deleted, the backend sends a message to the `game_events` queue in RabbitMQ.
2. The message includes the game ID, action type (create, update, delete), and relevant game data.
3. The message consumer service listens to the `game_events` queue and processes messages as they arrive.
4. After processing, the consumer updates the game's processing status in the database via the backend API.
5. The frontend displays the processing status on a dedicated page.

3.2 Docker Configuration

3.2.1 Docker Compose File

The Docker Compose file orchestrates the entire system, defining services, networks, and volumes:

Listing 1: Docker Compose File

```
version: '3.8'

services:
  # Frontend service
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    ports:
      - "80:80"
    depends_on:
      - backend
    networks:
      - app-network
    restart: unless-stopped

  # Backend service
  backend:
```

```
    build:
      context: ./backend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    depends_on:
      rabbitmq:
        condition: service_healthy
      database:
        condition: service_healthy
    environment:
      - DB_HOST=database
      - DB_USER=retro_user
      - DB_PASSWORD=retro_password
      - DB_NAME=retro_games_catalog
      - PORT=3000
      - RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
    networks:
      - app-network
    restart: unless-stopped

  # Database service
  database:
    build:
      context: ./database
      dockerfile: Dockerfile
    ports:
      - "3307:3306"  # Changed from 3306:3306 to 3307:3306
    environment:
      - MYSQL_ROOT_PASSWORD=retro_password
      - MYSQL_DATABASE=retro_games_catalog
      - MYSQL_USER=retro_user
      - MYSQL_PASSWORD=retro_password
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - app-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost", "-u",
          "retro_user", "-pretro_password"]
      interval: 10s
      timeout: 5s
      retries: 5

  # RabbitMQ service
  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"   # AMQP protocol port
      - "15672:15672" # Management UI port
```

```
    environment:
      - RABBITMQ_DEFAULT_USER=guest
      - RABBITMQ_DEFAULT_PASS=guest
    volumes:
      - rabbitmq-data:/var/lib/rabbitmq
    networks:
      - app-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "rabbitmq-diagnostics", "
         check_port_connectivity"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Message consumer service
  message-consumer:
    build:
      context: ./message-consumer
      dockerfile: Dockerfile
    depends_on:
      rabbitmq:
        condition: service_healthy
      backend:
        condition: service_started
    environment:
      - RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
      - BACKEND_URL=http://backend:3000
    networks:
      - app-network
    restart: unless-stopped

  # Message producer service
  message-producer:
    build:
      context: ./message-producer
      dockerfile: Dockerfile
    ports:
      - "3001:3001"
    depends_on:
      rabbitmq:
        condition: service_healthy
    environment:
      - RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
      - PORT=3001
    networks:
      - app-network
    restart: unless-stopped

# Define networks
networks:
```

```
  app-network:
    driver: bridge

# Define volumes
volumes:
  mysql-data:
    driver: local
  rabbitmq-data:
    driver: local
```

3.2.2 Dockerfiles

Listing 2: Frontend Dockerfile

Frontend Dockerfile

```
FROM nginx:alpine

# Copy static files
COPY . /usr/share/nginx/html/

# Copy nginx configuration
COPY nginx.conf /etc/nginx/conf.d/default.conf

# Expose port
EXPOSE 80

# Start nginx
CMD ["nginx", "-g", "daemon off;"]
```

Listing 3: Backend Dockerfile

Backend Dockerfile

```
FROM node:16-alpine

WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy application code
COPY . .

# Expose port
EXPOSE 3000

# Start the application
CMD ["node", "server.js"]
```

Listing 4: Message Consumer Dockerfile

Message Consumer Dockerfile

```
FROM python:3.9-slim

WORKDIR /app

# Install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY . .

# Run the application
CMD ["python", "app.py"]
```

Listing 5: Message Producer Dockerfile

Message Producer Dockerfile

```
FROM node:16-alpine

WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy application code
COPY . .

# Create public directory if it doesn't exist
RUN mkdir -p /app/public

# Expose port for the API
EXPOSE 3001

# Start the application
CMD ["node", "app.js"]
```

3.3 RabbitMQ Implementation

3.3.1 Backend (Producer)

The backend service acts as a message producer, sending messages to RabbitMQ when games are added, updated, or deleted:

Listing 6: Backend RabbitMQ Integration

```
// RabbitMQ connection
let rabbitConnection = null
```

```
let rabbitChannel = null

async function connectToRabbitMQ() {
  try {
    console.log(`Connecting to RabbitMQ at ${RABBITMQ_URL}...`)
    rabbitConnection = await amqp.connect(RABBITMQ_URL)
    rabbitChannel = await rabbitConnection.createChannel()

    // Declare queue with durability
    await rabbitChannel.assertQueue("game_events", { durable: true
        })

    console.log("RabbitMQ connection successful")

    // Set up connection error handling
    rabbitConnection.on("error", (err) => {
      console.error("RabbitMQ connection error:", err)
      rabbitChannel = null
      rabbitConnection = null
      setTimeout(connectToRabbitMQ, 5000)
    })

    rabbitConnection.on("close", () => {
      console.log("RabbitMQ connection closed, attempting to
         reconnect...")
      rabbitChannel = null
      rabbitConnection = null
      setTimeout(connectToRabbitMQ, 5000)
    })
  } catch (error) {
    console.error("RabbitMQ connection failed:", error)
    rabbitChannel = null
    rabbitConnection = null
    setTimeout(connectToRabbitMQ, 5000)
  }
}

// Function to send message to RabbitMQ
async function sendToQueue(queue, message) {
  try {
    if (!rabbitChannel) {
      console.error("RabbitMQ channel not available, reconnecting
         ...")
      await connectToRabbitMQ()
      if (!rabbitChannel) {
        console.error("Failed to reconnect to RabbitMQ")
        return false
      }
    }

    console.log(`Attempting to send message to queue ${queue}:`,
```

```
        message)

    const success = rabbitChannel.sendToQueue(
      queue,
      Buffer.from(JSON.stringify(message)),
      { persistent: true }, // Message will survive broker
         restarts
    )

    if (success) {
      console.log(`Message sent to queue ${queue} successfully`)
      return true
    } else {
      console.error(`Failed to send message to queue ${queue}`)
      return false
    }
  } catch (error) {
    console.error(`Error sending message to queue ${queue}:`,
       error)
    rabbitChannel = null
    rabbitConnection = null
    setTimeout(connectToRabbitMQ, 5000)
    return false
  }
}
```

3.3.2 Message Consumer

The message consumer service processes messages from RabbitMQ:

Listing 7: Message Consumer Implementation

```
def callback(ch, method, properties, body):
    """
    Process messages from the queue
    """
    try:
        # Parse the message
        message = json.loads(body)
        logger.info(f"Received message: {message}")

        # Get retry count from message headers or default to 0
        retry_count = 0
        if properties.headers and 'x-retry-count' in properties.
           headers:
            retry_count = properties.headers['x-retry-count']

        # Check message type
        if 'type' in message and message['type'] == 'custom':
            # Process custom message
            success = process_custom_message(message, retry_count)
        else:
```

```
            # Process game event
            game_id = message.get('gameId')
            action = message.get('action')
            game_data = message.get('gameData', {})

            if not game_id or not action:
                logger.error("Message missing required fields (gameId or action), cannot process")
                ch.basic_ack(delivery_tag=method.delivery_tag)
                return

            # Simulate processing time (1-3 seconds)
            processing_time = 2
            logger.info(f"Processing will take approximately {processing_time} seconds")
            time.sleep(processing_time)

            # Process the message
            success = process_game_event(game_id, action, game_data, retry_count)

        if success:
            # Acknowledge the message
            ch.basic_ack(delivery_tag=method.delivery_tag)
            logger.info(f"Processing complete for message, acknowledged")
        else:
            # Negative acknowledgment with requeue
            # Add retry count to headers
            headers = properties.headers or {}
            headers['x-retry-count'] = retry_count + 1

            # Publish the message back to the queue with updated headers
            ch.basic_publish(
                exchange='',
                routing_key=QUEUE_NAME,
                body=body,
                properties=pika.BasicProperties(
                    delivery_mode=2,  # make message persistent
                    headers=headers
                )
            )

            # Acknowledge the original message
            ch.basic_ack(delivery_tag=method.delivery_tag)
            logger.info(f"Message requeued for retry (attempt {retry_count + 1})")
```

3.4 Database Schema

The database schema includes a processing status field to track the status of game processing:

Listing 8: Database Schema

```sql
-- Create games table with processing status
CREATE TABLE IF NOT EXISTS games (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(100) NOT NULL,
    platform_id INT NOT NULL,
    genre_id INT NOT NULL,
    developer VARCHAR(100) NOT NULL,
    release_year INT NOT NULL,
    year_category VARCHAR(20) NOT NULL,
    description TEXT NOT NULL,
    image_url VARCHAR(255),
    processing_status ENUM('pending', 'processing', 'completed', '
       failed') DEFAULT 'completed',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
       CURRENT_TIMESTAMP,
    FOREIGN KEY (platform_id) REFERENCES platforms(id) ON DELETE
       CASCADE,
    FOREIGN KEY (genre_id) REFERENCES genres(id) ON DELETE CASCADE
);
```

3.5 Frontend Implementation

The frontend includes a dedicated page for monitoring processing tasks:

Listing 9: Processing Tasks Page

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
       scale=1.0">
    <title>Processing Tasks - Retro Gaming Catalog</title>
    <link rel="stylesheet" href="styles.css">
    <style>
        .task-list {
            display: grid;
            grid-template-columns: 1fr;
            gap: 15px;
            margin-top: 20px;
        }

        .task-card {
            background-color: #1a1a1a;
            border: 2px solid #33ff66;
```

```
            padding: 15px;
            display: grid;
            grid-template-columns: auto 1fr auto;
            gap: 15px;
            align-items: center;
        }

        .task-id {
            font-weight: bold;
            font-size: 1.2rem;
            color: #33ff66;
        }

        .task-title {
            font-size: 1.1rem;
        }

        .status-badge {
            padding: 5px 10px;
            border-radius: 4px;
            font-size: 0.9rem;
            font-weight: bold;
            text-align: center;
            min-width: 100px;
        }

        .status-pending {
            background-color: #ff9900;
            color: #000;
        }

        .status-processing {
            background-color: #3399ff;
            color: #000;
        }

        .status-completed {
            background-color: #33ff66;
            color: #000;
        }

        .status-failed {
            background-color: #ff3366;
            color: #fff;
        }
    </style>
</head>
<body>
    <header>
        <h1>Retro Gaming Catalog</h1>
    </header>
```

```
70
71      <div class="nav-links">
72          <a href="index.html" class="nav-link">Game Catalog</a>
73          <a href="processing.html" class="nav-link">Processing
                Tasks</a>
74      </div>
75
76      <div class="container">
77          <h2>Background Processing Tasks</h2>
78          <p>This page shows the status of game events being
                processed by our RabbitMQ message consumer. When you
                add, update, or delete a game, a message is sent to
                RabbitMQ and processed asynchronously.</p>
79
80          <div class="auto-refresh">
81              <input type="checkbox" id="auto-refresh" checked>
82              <label for="auto-refresh">Auto-refresh every 5 seconds
                    </label>
83          </div>
84
85          <button id="refresh-btn" class="refresh-btn">Refresh Now</
                button>
86
87          <div class="task-list" id="task-list">
88              <div class="loading">Loading tasks...</div>
89          </div>
90      </div>
91  </body>
92  </html>
```

4 Resilience Features

The implementation includes several resilience features to ensure the system can recover from failures:

4.1 Message Persistence

Messages are stored on disk and survive broker restarts:

Listing 10: Message Persistence

```
1  const success = rabbitChannel.sendToQueue(
2    queue,
3    Buffer.from(JSON.stringify(message)),
4    { persistent: true }, // Message will survive broker restarts
5  )
```

4.2 Message Acknowledgment

Messages are only removed from the queue after successful processing:

Listing 11: Message Acknowledgment

```
if success:
    # Acknowledge the message
    ch.basic_ack(delivery_tag=method.delivery_tag)
    logger.info(f"Processing complete for message, acknowledged")
else:
    # Negative acknowledgment with requeue
    # Add retry count to headers
    headers = properties.headers or {}
    headers['x-retry-count'] = retry_count + 1

    # Publish the message back to the queue with updated headers
    ch.basic_publish(
        exchange='',
        routing_key=QUEUE_NAME,
        body=body,
        properties=pika.BasicProperties(
            delivery_mode=2,  # make message persistent
            headers=headers
        )
    )

    # Acknowledge the original message
    ch.basic_ack(delivery_tag=method.delivery_tag)
    logger.info(f"Message requeued for retry (attempt {retry_count
        + 1})")
```

4.3 Error Handling and Retries

Failed messages are requeued for retry with a maximum retry count:

Listing 12: Error Handling and Retries

```
# If we haven't exceeded max retries, return False to trigger
   requeue
if retry_count < MAX_RETRIES:
    logger.info(f"Will retry processing game {game_id} (attempt {
       retry_count + 1}/{MAX_RETRIES})")
    return False
else:
    logger.error(f"Max retries exceeded for game {game_id}, giving
        up")

    # Update the game's processing status to 'failed'
    try:
        requests.put(
            f"{BACKEND_URL}/api/games/{game_id}/process-complete",
            json={"status": "failed"},
            timeout=5
        )
    except:
```

```
        pass  # Ignore errors when updating status to failed

    return True  # Return True to acknowledge and remove from queue
```

4.4 Connection Recovery

Services automatically reconnect to RabbitMQ if the connection is lost:

Listing 13: Connection Recovery

```
// Set up connection error handling
rabbitConnection.on("error", (err) => {
  console.error("RabbitMQ connection error:", err)
  rabbitChannel = null
  rabbitConnection = null
  setTimeout(connectToRabbitMQ, 5000)
})

rabbitConnection.on("close", () => {
  console.log("RabbitMQ connection closed, attempting to reconnect...")
  rabbitChannel = null
  rabbitConnection = null
  setTimeout(connectToRabbitMQ, 5000)
})
```

4.5 Health Checks

Docker Compose includes health checks for RabbitMQ and the database:

Listing 14: Health Checks

```
rabbitmq:
  image: rabbitmq:3-management
  # ... other configuration ...
  healthcheck:
    test: ["CMD", "rabbitmq-diagnostics", "check_port_connectivity"]
    interval: 10s
    timeout: 5s
    retries: 5

database:
  build:
    context: ./database
    dockerfile: Dockerfile
  # ... other configuration ...
  healthcheck:
    test: ["CMD", "mysqladmin", "ping", "-h", "localhost", "-u", "retro_user", "-pretro_password"]
    interval: 10s
```

```
    timeout: 5s
    retries: 5
```

5 Evidence of Operation

5.1 Running Containers

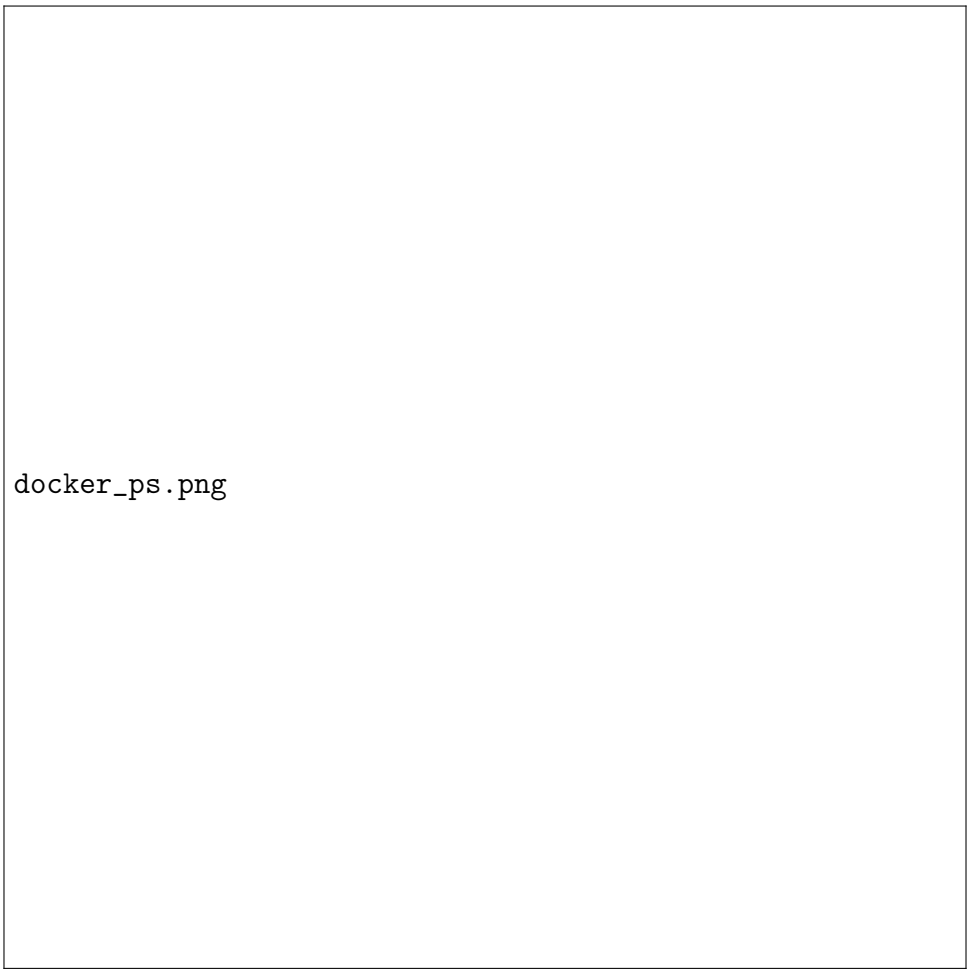

Figure 2: Running Docker Containers

5.2 RabbitMQ Management UI

Figure 3: RabbitMQ Management UI

5.3 Message Logs

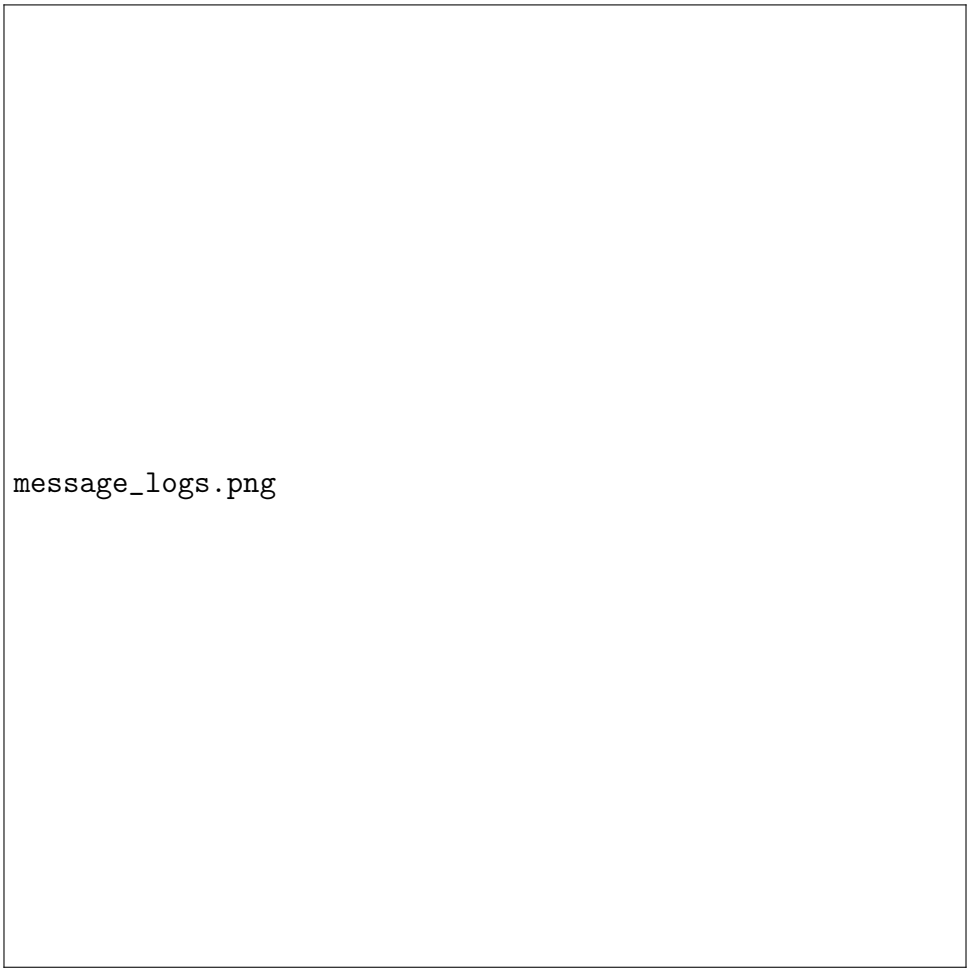

Figure 4: Message Passing Logs

5.4 Processing Tasks UI

Figure 5: Processing Tasks UI

6 Setup and Run Instructions

6.1 Prerequisites

- Docker and Docker Compose
- Git (optional, for cloning the repository)

6.2 Installation Steps

1. Clone or download the repository
2. Navigate to the project directory
3. Create a `.env` file with the following variables:

```
DB_HOST=database
DB_USER=retro_user
DB_PASSWORD=retro_password
```

```
DB_NAME=retro_games_catalog
PORT=3000
RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
```

4. Build and start the containers:

```
docker-compose up -d
```

5. Access the application:
   - Frontend: http://localhost
   - RabbitMQ Management UI: http://localhost:15672 (username: guest, password: guest)
   - Message Producer UI: http://localhost:3001

6.3 Troubleshooting

If messages are not flowing through RabbitMQ:

1. Check RabbitMQ is running: `docker-compose ps rabbitmq`
2. Verify connections in RabbitMQ Management UI: http://localhost:15672
3. Check logs: `docker-compose logs rabbitmq`
4. Check consumer logs: `docker-compose logs message-consumer`
5. Check producer logs: `docker-compose logs message-producer`
6. Restart services: `docker-compose restart rabbitmq message-consumer message-producer backend`

7 Conclusion

7.1 Summary

This project successfully enhanced a three-tier application with RabbitMQ messaging patterns to create a more resilient, loosely coupled system. The implementation focused on the Producer/Consumer pattern for processing background operations in a Retro Gaming Catalog application.

7.2 Lessons Learned

- Asynchronous messaging provides significant benefits for system resilience and scalability
- Proper error handling and recovery mechanisms are essential for distributed systems
- Docker Compose health checks ensure services start in the correct order
- Message persistence and acknowledgment are critical for reliable message delivery

7.3 Future Improvements

- Implement additional RabbitMQ patterns (Publish/Subscribe, Request/Reply)
- Add monitoring and alerting for RabbitMQ and services
- Implement message encryption for sensitive data
- Scale the system with multiple consumers for parallel processing

8 References

1. RabbitMQ Documentation: https://www.rabbitmq.com/documentation.html
2. Docker Documentation: https://docs.docker.com/
3. Node.js amqplib Documentation: https://www.npmjs.com/package/amqplib
4. Python Pika Documentation: https://pika.readthedocs.io/