Sistema de Notificaciones con RabbitMQ para el Observatorio de Datos de Calidad del Agua

Introducción

Este documento proporciona una explicación detallada del sistema de notificaciones basado en RabbitMQ desarrollado para el Observatorio de Datos de Calidad del Agua. El sistema está diseñado siguiendo el patrón de arquitectura publicador-suscriptor (pub/sub), permitiendo una comunicación asíncrona entre los componentes que generan datos y aquellos que necesitan ser notificados cuando ocurren eventos relevantes.

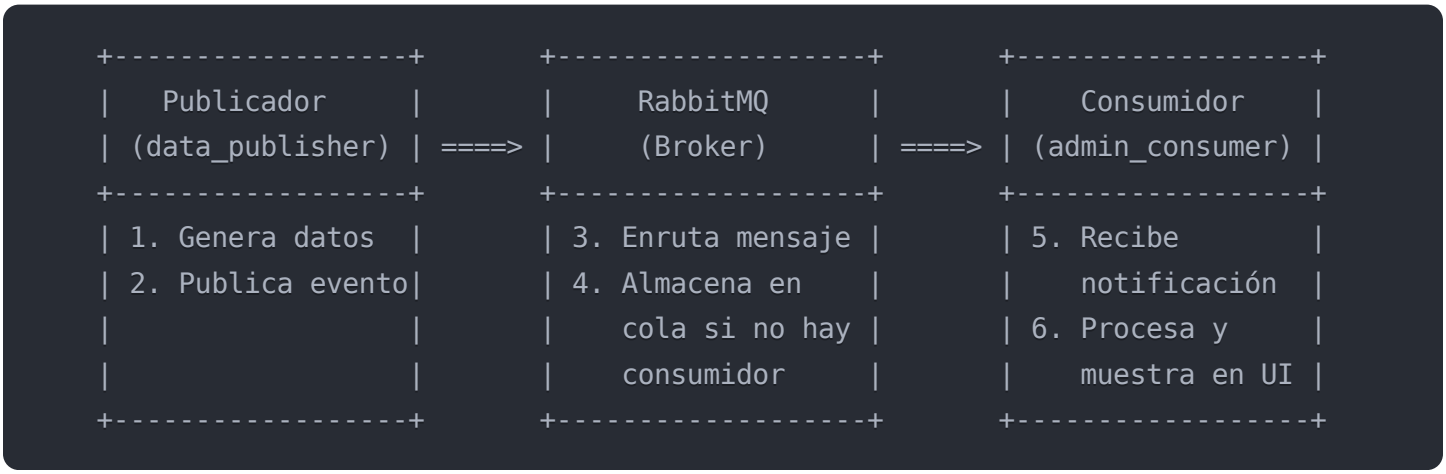
Arquitectura del Sistema

El sistema está compuesto por tres componentes principales:

1. **Utilidades de Conexión (rabbitmq_utils.py)**: Proporciona las funciones y clases base para conectarse con RabbitMQ y gestionar la serialización de mensajes.
2. **Servicio Publicador (data_publisher.py)**: Simula la carga de datos de calidad del agua y publica notificaciones cuando ocurren estos eventos.
3. **Servicio Consumidor (admin_consumer.py)**: Recibe y procesa las notificaciones, mostrándolas en una interfaz de consola para los administradores.

Diagrama de Flujo del Sistema

Copiar



Tecnologías Utilizadas

- **Python 3.8+**: Lenguaje de programación base
- **RabbitMQ 3.8+**: Sistema de mensajería que actúa como broker
- **pika**: Cliente Python para RabbitMQ
- **json**: Para serialización/deserialización de mensajes
- **threading**: Para gestionar múltiples hilos en el consumidor
- **argparse**: Para procesar argumentos de línea de comandos

Explicación Detallada de los Componentes

1. Módulo de Utilidades (rabbitmq_utils.py)

Este módulo contiene las funcionalidades básicas para interactuar con RabbitMQ y es utilizado tanto por el publicador como por el consumidor.

Constantes de Configuración

python Copiar

```python
# Constantes para la configuración
EXCHANGE_NAME = 'observatorio_events'
QUEUE_NAME = 'admin_notifications'
ROUTING_KEY = 'water.data.uploaded'
```

Estas constantes definen:

- `EXCHANGE_NAME`: El nombre del exchange en RabbitMQ donde se publicarán los mensajes
- `QUEUE_NAME`: El nombre de la cola que almacenará los mensajes
- `ROUTING_KEY`: La clave de enrutamiento que determina qué mensajes llegan a qué colas

Clase RabbitMQConnection

Esta clase encapsula la lógica de conexión con RabbitMQ:

python

Copiar

```python
class RabbitMQConnection:
    """Clase para gestionar la conexión con RabbitMQ"""

    def __init__(self, host='localhost', port=5672,
                 user='guest', password='guest', virtual_host='/'):
        """Inicializar la conexión a RabbitMQ"""
        self.host = host
        self.port = port
        self.user = user
        self.password = password
        self.virtual_host = virtual_host
        self.connection = None
        self.channel = None
```

El constructor permite configurar los parámetros de conexión, con valores por defecto que funcionan con una instalación estándar de RabbitMQ.

Método connect()

python

Copiar

```python
def connect(self):
    """Establecer conexión con RabbitMQ"""
    try:
        # Credenciales de conexión
        credentials = pika.PlainCredentials(self.user, self.password)
        # Parámetros de conexión
        parameters = pika.ConnectionParameters(
            host=self.host,
            port=self.port,
            virtual_host=self.virtual_host,
            credentials=credentials,
            heartbeat=600  # Mantener conexión activa
        )
        # Establecer conexión
        self.connection = pika.BlockingConnection(parameters)
        self.channel = self.connection.channel()

        # Declarar exchange de tipo 'topic' para enrutamiento flexible
        self.channel.exchange_declare(
            exchange=EXCHANGE_NAME,
            exchange_type='topic',
            durable=True  # Persistente ante reinicios
        )

        logger.info(f"Conectado a RabbitMQ en {self.host}:{self.port}")
        return True
    except AMQPConnectionError as e:
        logger.error(f"Error al conectar a RabbitMQ: {e}")
        return False
```

Este método:

1. Establece credenciales usando `pika.PlainCredentials`
2. Configura parámetros de conexión como host, puerto, y heartbeat
3. Crea una conexión bloqueante con RabbitMQ
4. Obtiene un canal de comunicación
5. Declara un exchange de tipo "topic", que permite un enrutamiento flexible de mensajes
6. El parámetro `durable=True` asegura que el exchange persista incluso si RabbitMQ se reinicia

Funciones de Serialización

python

Copiar

```python
def serialize_message(data):
    """Convertir datos a formato JSON para enviar como mensaje"""
    try:
        return json.dumps(data).encode('utf-8')
    except Exception as e:
        logger.error(f"Error al serializar mensaje: {e}")
        return json.dumps({"error": "Error de serialización"}).encode('utf-8')

def deserialize_message(body):
    """Convertir mensaje recibido de bytes a objeto Python"""
    try:
        return json.loads(body.decode('utf-8'))
    except Exception as e:
        logger.error(f"Error al deserializar mensaje: {e}")
        return {"error": "Error al deserializar mensaje"}
```

Estas funciones:

- Convierten datos Python a JSON y luego a bytes para enviar a RabbitMQ
- Convierten bytes recibidos de RabbitMQ a JSON y luego a objetos Python
- Incluyen manejo de errores para garantizar que no se interrumpa el flujo

2. Servicio Publicador (data_publisher.py)

Este servicio simula la carga de datos de calidad del agua en el sistema y publica notificaciones cuando ocurren estos eventos.

Simulación de Datos

python

Copiar

```python
def simulate_data_upload():
    """
    Simula la carga de datos de calidad del agua generando
    valores aleatorios para los parámetros.
    """
    # Generar ID único para el lote de datos
    batch_id = str(uuid.uuid4())

    # Seleccionar ubicación y entidad aleatoria
    location = random.choice(SAMPLE_LOCATIONS)
    entity = random.choice(REPORTING_ENTITIES)

    # Generar entre 3 y 10 mediciones aleatorias
    num_measurements = random.randint(3, 10)
    measurements = []

    # Fecha y hora actual para la carga
    timestamp = datetime.now().isoformat()

    # Generar mediciones aleatorias con lógica por tipo de parámetro
    for _ in range(num_measurements):
        parameter = random.choice(WATER_PARAMETERS)
        # [código que genera valores según el parámetro]

        # Determinar si el valor está por encima del límite permisible
        threshold_exceeded = random.random() < 0.2  # 20% de probabilidad

        measurements.append({
            "parameter": parameter,
            "value": value,
            "unit": unit,
            "threshold_exceeded": threshold_exceeded
        })

    # Componer los datos completos de la carga
    data = {
        "batch_id": batch_id,
        "timestamp": timestamp,
        "location": location,
        "reporting_entity": entity,
        "measurements": measurements,
        "metadata": {
            "device_id": f"SENSOR-{random.randint(1000, 9999)}",
            "upload_method": random.choice(["API", "Desktop App", "Field Device"]),
```

```
            "comments": "Datos simulados para prueba de concepto"
        }
    }

    return data
```

Este método simula la carga de datos generando:

1. Un ID único para el lote de datos usando UUID
2. Selección aleatoria de ubicación y entidad informante
3. Entre 3 y 10 mediciones aleatorias de parámetros de calidad del agua
4. Valores apropiados para cada tipo de parámetro (pH, temperatura, turbidez, etc.)
5. Indicación aleatoria de si el valor excede los umbrales permitidos
6. Estructura JSON completa con todos los datos necesarios

Publicación de Notificaciones

python

Copiar

```python
def publish_notification(rmq_connection, data):
    """
    Publica una notificación en RabbitMQ cuando se cargan datos.

    Args:
        rmq_connection: Conexión a RabbitMQ
        data: Datos a publicar en la notificación
    """
    try:
        # Preparar mensaje de notificación
        notification = {
            "event_type": "data_upload",
            "timestamp": datetime.now().isoformat(),
            "data": data
        }

        # Serializar mensaje
        message_body = serialize_message(notification)

        # Publicar mensaje en el exchange
        rmq_connection.channel.basic_publish(
            exchange=EXCHANGE_NAME,
            routing_key=ROUTING_KEY,
            body=message_body,
            properties=pika.BasicProperties(
                delivery_mode=2,  # Mensaje persistente
                content_type='application/json'
            )
        )

        logger.info(f"Notificación publicada: batch_id={data['batch_id']}")
        return True
    except Exception as e:
        logger.error(f"Error al publicar notificación: {e}")
        return False
```

Este método:

1. Crea una estructura de notificación con tipo de evento, timestamp y datos
2. Serializa el mensaje con la función del módulo de utilidades
3. Publica el mensaje al exchange de RabbitMQ utilizando la routing key definida
4. Configura el mensaje como persistente (`delivery_mode=2`) para asegurar que no se pierda
5. Incluye manejo de errores para registrar cualquier problema durante la publicación

Función Principal (main)

python

Copiar

```python
def main():
    """Función principal que simula cargas periódicas de datos"""
    parser = argparse.ArgumentParser(description='Simulador de carga de datos para el
    parser.add_argument('--host', default='localhost', help='Host de RabbitMQ')
    parser.add_argument('--port', type=int, default=5672, help='Puerto de RabbitMQ')
    parser.add_argument('--user', default='guest', help='Usuario de RabbitMQ')
    parser.add_argument('--password', default='guest', help='Contraseña de RabbitMQ')
    parser.add_argument('--interval', type=int, default=10,
                        help='Intervalo en segundos entre cargas de datos (default: 10
    args = parser.parse_args()

    # Crear conexión a RabbitMQ y ejecutar bucle principal
    # [código del bucle principal que simula cargas periódicas]
```

La función principal:

1. Configura un parser de argumentos para permitir personalizar la conexión y el intervalo
2. Establece una conexión con RabbitMQ
3. En un bucle infinito:
   - Simula una carga de datos
   - Publica una notificación
   - Espera el intervalo configurado
4. Maneja interrupciones de usuario (Ctrl+C) para salir limpiamente
5. Garantiza que la conexión se cierre al finalizar

3. Servicio Consumidor (admin_consumer.py)

Este servicio recibe las notificaciones de carga de datos y las muestra en una interfaz de consola para los administradores.

Clase ConsoleNotifier

python

Copiar

```python
class ConsoleNotifier:
    """
    Clase para mostrar notificaciones en la consola con formato
    y colores para mejor visualización.
    """
    # Códigos ANSI para colores en terminal
    RESET = "\033[0m"
    BOLD = "\033[1m"
    RED = "\033[91m"
    GREEN = "\033[92m"
    # [más códigos de color]

    def __init__(self):
        """Inicializa el contador de notificaciones"""
        self.notification_count = 0

    def clear_screen(self):
        """Limpia la pantalla de la consola"""
        os.system('cls' if os.name == 'nt' else 'clear')

    def print_header(self):
        """Imprime el encabezado de la aplicación"""
        self.clear_screen()
        # [código para imprimir encabezado]

    def display_notification(self, data):
        """Muestra una notificación con formato en la consola"""
        self.notification_count += 1
        self.print_header()

        # Extraer datos relevantes
        batch_id = data['data']['batch_id']
        timestamp = data['data']['timestamp']
        location = data['data']['location']
        entity = data['data']['reporting_entity']
        measurements = data['data']['measurements']

        # [código para mostrar notificación formateada]

        # Mostrar alerta si hay mediciones que exceden umbrales
        alerts = [m for m in measurements if m['threshold_exceeded']]
        if alerts:
            print(f"\n{self.BOLD}{self.RED}¡ATENCIÓN! {len(alerts)} parámetro(s) exced
            print(f"{self.YELLOW}Se requiere revisión por parte del administrador.{sel
```

Esta clase:

1. Define códigos ANSI para colorear la salida en terminal
2. Mantiene un contador de notificaciones recibidas
3. Proporciona métodos para limpiar la pantalla y mostrar un encabezado
4. Implementa la lógica para mostrar notificaciones formateadas con colores
5. Resalta en rojo los parámetros que exceden los umbrales permitidos

Configuración del Consumidor

python

Copiar

```python
def setup_consumer(rmq_connection):
    """
    Configura el consumidor para recibir notificaciones
    """
    # Crear cola para el consumidor y enlazarla al exchange
    rmq_connection.channel.queue_declare(
        queue=QUEUE_NAME,
        durable=True  # Persistente ante reinicios
    )

    # Vincular cola al exchange con la routing key
    rmq_connection.channel.queue_bind(
        exchange=EXCHANGE_NAME,
        queue=QUEUE_NAME,
        routing_key=ROUTING_KEY
    )

    # Crear instancia del notificador
    notifier = ConsoleNotifier()
    notifier.print_header()

    # Configurar callback con el notificador
    callback = lambda ch, method, properties, body: process_message(
        ch, method, properties, body, notifier
    )

    # Configurar consumidor
    rmq_connection.channel.basic_qos(prefetch_count=1)
    rmq_connection.channel.basic_consume(
        queue=QUEUE_NAME,
        on_message_callback=callback
    )

    return notifier
```

Este método:

1. Declara una cola durable para recibir mensajes (persistente ante reinicios)
2. Vincula la cola al exchange usando la routing key para recibir los mensajes relevantes
3. Crea una instancia del notificador para mostrar las notificaciones
4. Configura un callback que procesará los mensajes recibidos
5. Establece `prefetch_count=1` para no sobrecargar el consumidor
6. Configura el consumidor para recibir mensajes de la cola

Procesamiento de Mensajes

python

Copiar

```python
def process_message(ch, method, properties, body, notifier):
    """
    Callback para procesar mensajes recibidos de RabbitMQ
    """
    # Deserializar mensaje
    data = deserialize_message(body)

    # Procesar y mostrar notificación
    logger.debug(f"Mensaje recibido: {data}")
    notifier.display_notification(data)

    # Confirmar recepción del mensaje
    ch.basic_ack(delivery_tag=method.delivery_tag)
```

Este callback:

1. Deserializa el mensaje recibido
2. Registra la recepción en los logs (nivel debug)
3. Utiliza el notificador para mostrar la información en la consola
4. Confirma la recepción del mensaje con `basic_ack`, lo que elimina el mensaje de la cola

Manejo de Múltiples Hilos

python

Copiar

```python
def main():
    """Función principal del consumidor de notificaciones"""
    # [configuración de argumentos y conexión]

    try:
        # [conexión y configuración del consumidor]

        # Iniciar consumo de mensajes en un hilo separado
        def consume_messages():
            try:
                logger.info("Iniciando consumo de mensajes...")
                rmq_connection.channel.start_consuming()
            except Exception as e:
                logger.error(f"Error en consumo de mensajes: {e}")
            finally:
                if rmq_connection.channel and rmq_connection.channel.is_open:
                    rmq_connection.channel.stop_consuming()

        consumer_thread = threading.Thread(target=consume_messages)
        consumer_thread.daemon = True
        consumer_thread.start()

        # Esperar señal de cierre
        while not shutdown_flag.is_set():
            time.sleep(0.1)  # Pequeña pausa para evitar uso excesivo de CPU

    except Exception as e:
        logger.error(f"Error en el consumidor: {e}")
    finally:
        # [código para cerrar correctamente la conexión]
```

Esta implementación:

1. Crea un hilo separado para el consumo de mensajes, lo que permite que la aplicación siga respondiendo a señales de interrupción
2. Marca el hilo como "daemon" para que termine cuando el hilo principal termine
3. Utiliza una bandera (shutdown_flag) para coordinar la terminación limpia
4. Incluye manejo de excepciones en múltiples niveles para garantizar robustez
5. Asegura que los recursos se liberen correctamente en caso de error o terminación

Configuración y Ejecución del Sistema

Pasos para la Instalación

1. **Instalar RabbitMQ**:

bash Copiar

```
# Con el script de instalación que proporcionaste
./install-rabbitmq.sh
```

2. **Instalar dependencias de Python**:

bash Copiar

```
pip install pika
```

3. **Habilitar el plugin de gestión web (opcional)**:

bash Copiar

```
sudo rabbitmq-plugins enable rabbitmq_management
```

Ejecución del Sistema

1. **Iniciar el consumidor** en una terminal:

bash Copiar

```
python admin_consumer.py
```

2. **Iniciar el publicador** en otra terminal:

bash Copiar

```
python data_publisher.py --interval 5
```

Estructura de los Mensajes

json

Copiar


```
{
  "event_type": "data_upload",
  "timestamp": "2025-04-05T14:30:45.123456",
  "data": {
    "batch_id": "550e8400-e29b-41d4-a716-446655440000",
    "timestamp": "2025-04-05T14:30:40.123456",
    "location": "Río Principal - Estación Norte",
    "reporting_entity": "Secretaría de Medio Ambiente",
    "measurements": [
      {
        "parameter": "pH",
        "value": 7.2,
        "unit": "pH",
        "threshold_exceeded": false
      },
      {
        "parameter": "Turbidez",
        "value": 12.5,
        "unit": "NTU",
        "threshold_exceeded": true
      }
    ],
    "metadata": {
      "device_id": "SENSOR-1234",
      "upload_method": "API",
      "comments": "Datos simulados para prueba de concepto"
    }
  }
}
```


Consideraciones para Entornos de Producción

Escalabilidad

Para escalar este sistema en un entorno de producción, se recomiendan las siguientes mejoras:

1. **Cluster de RabbitMQ**: Implementar un cluster con al menos 3 nodos para alta disponibilidad.
2. **Balanceo de carga**: Utilizar HAProxy o un balanceador de carga similar para distribuir las conexiones entre los nodos del cluster.
3. **Consumidores especializados**:
   - Desarrollar consumidores específicos por tipo de notificación (email, SMS, dashboard web).
   - Implementar worker pools para procesamiento paralelo de mensajes.
4. **Configuraciones de desempeño**:
   - Ajustar parámetros como prefetch_count según las características de la carga.
   - Implementar reconexión automática con retroceso exponencial.

Seguridad

1. **Autenticación y autorización**:
   - Crear usuarios específicos para cada servicio con permisos limitados.
   - Implementar políticas de rotación de contraseñas.
2. **Comunicaciones seguras**:
   - Habilitar TLS para cifrar las comunicaciones entre clientes y el broker.
   - Configurar firewalls para restringir el acceso al broker de mensajes.

Monitoreo y Gestión

1. **Observabilidad**:
   - Implementar Prometheus y Grafana para monitoreo de métricas.
   - Centralizar logs con ELK Stack (Elasticsearch, Logstash, Kibana).
   - Configurar alertas para colas con acumulación de mensajes o latencias altas.
2. **Respaldo y recuperación**:
   - Configurar respaldos regulares de las definiciones de RabbitMQ.
   - Implementar estrategias de recuperación ante desastres con replicación.

Conclusión

El sistema implementado proporciona una base sólida para un mecanismo de notificaciones en tiempo real basado en el patrón publicador-suscriptor. La arquitectura desacoplada facilita la escalabilidad y la extensión con nuevos tipos de notificaciones o consumidores adicionales.

La elección de RabbitMQ como broker de mensajes proporciona características importantes como:

- Garantía de entrega de mensajes
- Persistencia ante reinicios
- Flexibilidad en el enrutamiento de mensajes
- Posibilidad de escalar horizontalmente

Esta implementación demuestra cómo pueden integrarse diferentes componentes del Observatorio de Datos de Calidad del Agua utilizando mensajería asíncrona, lo que permite una arquitectura flexible y resiliente.