Implementación del Patrón Publicador-Suscriptor para un Sistema de Intercambio y Venta de Libros Usados

Juan David Jiménez Castellanos, Sebastian David Gonzales,
Rodrigo Alberto Osorio

Asignatura:
Estructura y Arquitectura de Software

Profesor:
Luis Carlos Garzon

Universidad de Cundinamarca
Facultad de Ingeniería
Programa de Ingeniería de Sistemas

4 de octubre de 2025

Resumen

Este documento presenta la implementación del patrón arquitectónico Publicador-Suscriptor utilizando RabbitMQ para un sistema de intercambio y venta de libros usados. El patrón permite la comunicación asíncrona entre componentes del sistema, facilitando la gestión de notificaciones en tiempo real sobre nuevos libros disponibles, ofertas recibidas y actualizaciones de estado de ventas. Se implementó utilizando Python como lenguaje de programación, RabbitMQ como *message broker* y Tkinter para la interfaz gráfica. Los resultados demuestran la efectividad del patrón para desacoplar componentes y mejorar la escalabilidad del sistema.

Palabras clave: Patrón Publicador-Suscriptor, RabbitMQ, Libros Usados, Sistemas Distribuidos, Arquitectura de Software.

Índice

1. Introducción

En el contexto actual del comercio electrónico, los sistemas de intercambio y venta de productos usados han ganado significativa popularidad. Específicamente, el mercado de libros usados representa una alternativa económica y ecológica para los lectores. Sin embargo, la gestión eficiente de notificaciones en tiempo real sobre disponibilidad de libros, ofertas y actualizaciones de estado presenta desafíos técnicos significativos.

El patrón Publicador-Suscriptor emerge como una solución arquitectónica ideal para este tipo de sistemas, permitiendo la comunicación asíncrona entre componentes distribuidos. Este patrón desacopla los componentes que generan eventos (publicadores) de aquellos que los procesan (suscriptores), facilitando la escalabilidad y mantenibilidad del sistema.

RabbitMQ, como implementación del protocolo AMQP (Advanced Message Queuing Protocol), proporciona una plataforma robusta para la implementación de este patrón. Su capacidad para manejar colas de mensajes de manera confiable lo convierte en una opción adecuada para sistemas de comercio electrónico que requieren garantías de entrega de mensajes.

Este trabajo presenta la implementación del patrón Publicador-Suscriptor utilizando RabbitMQ para un sistema de intercambio y venta de libros usados, demostrando su aplicabilidad en escenarios reales de comercio electrónico.

2. Arquitectura del Sistema

2.1. Patrón Publicador-Suscriptor

El patrón Publicador-Suscriptor es un patrón de mensajería asíncrona donde los componentes del sistema se comunican a través de mensajes sin conocimiento directo entre sí. En este modelo:

- **Publicadores**: Generan mensajes sobre eventos específicos
- **Suscriptores**: Expresan interés en ciertos tipos de mensajes
- **Message Broker**: Actúa como intermediario, enrutando mensajes desde publicadores a suscriptores

2.2. Arquitectura Propuesta

La figura 1 ilustra la arquitectura propuesta para el sistema de libros usados:

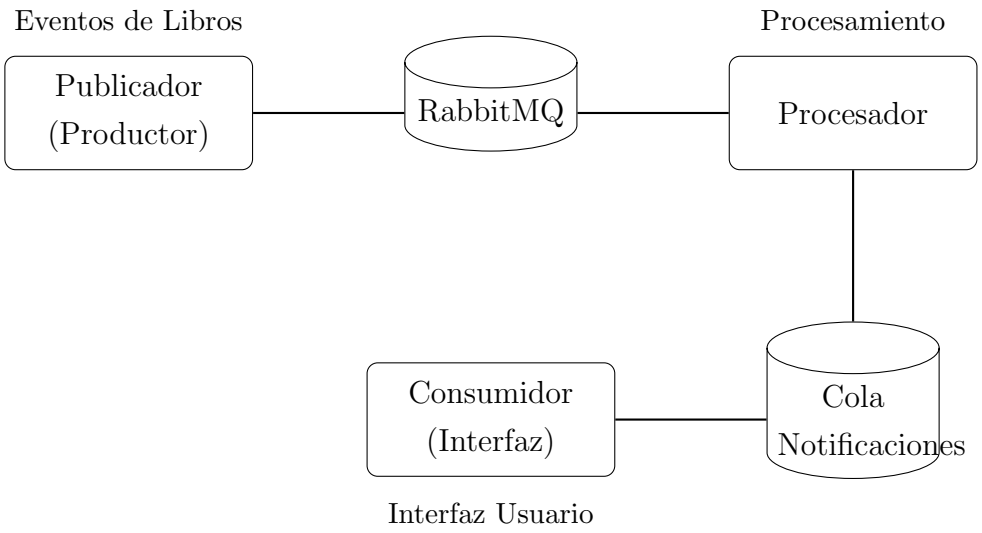

Figura 1: Arquitectura del sistema Publicador-Suscriptor para libros usados

2.3. Componentes del Sistema

La tabla 1 describe los componentes principales del sistema:

Componente	**Descripción**
Publicador (Productor)	Genera eventos cuando usuarios publican libros, reciben ofertas o completan ventas
RabbitMQ (Message Broker)	Gera las colas de mensajes y enruta eventos entre componentes
Procesador	Procesa eventos y decide a quién notificar basado en reglas de negocio
Consumidor (Interfaz)	Muestra notificaciones a usuarios y gestiona la interacción

Cuadro 1: Componentes del sistema de libros usados

3. Implementación

3.1. Configuración de RabbitMQ

Se implementó una conexión básica a RabbitMQ utilizando la biblioteca Pika para Python:

```
import pika

def conectar_rabbitmq():
    """
    Establece conexi n con RabbitMQ local
```

```
    Returns:
        canal: Canal de comunicaci n con RabbitMQ
    """
    try:
        conexion = pika.BlockingConnection(
            pika.ConnectionParameters('localhost')
        )
        canal = conexion.channel()
        print("[Sistema] Conexi n establecida con RabbitMQ")
        return canal
    except Exception as e:
        print(f"[Error] No se pudo conectar a RabbitMQ: {e}")
        return None
```

Listing 1: Conexión a RabbitMQ

3.2. Publicador de Eventos

El publicador gestiona la creación de eventos relacionados con libros:

```
import json
from datetime import datetime
from conexion import conectar_rabbitmq

def publicar_evento_libro(tipo_evento, datos_libro):
    """
    Publica un evento relacionado con libros en la cola de RabbitMQ
    """
    canal = conectar_rabbitmq()
    if canal is None:
        return

    # Declarar cola para eventos de libros
    canal.queue_declare(queue='eventos_libros', durable=True)

    # Construir mensaje
    mensaje = {
        'tipo': tipo_evento,
        'libro': datos_libro,
        'timestamp': datetime.now().isoformat(),
        'version': '1.0'
    }

    # Publicar mensaje
    canal.basic_publish(
        exchange='',
        routing_key='eventos_libros',
```

```
        body=json.dumps(mensaje),
        properties=pika.BasicProperties(
            delivery_mode=2,  # Hacer el mensaje persistente
        )
    )

    print(f"[BookTrade] Evento publicado: {tipo_evento}")
```

Listing 2: Publicador de eventos de libros

3.3. Procesador de Eventos

El procesador recibe eventos y aplica lógica de negocio para determinar las notificaciones:

```
import json
import pika
from conexion import conectar_rabbitmq

def iniciar_procesador_libros():
    """
    Inicia el procesador que escucha eventos de libros
    """
    canal = conectar_rabbitmq()
    if canal is None:
        return

    # Declarar colas
    canal.queue_declare(queue='eventos_libros', durable=True)
    canal.queue_declare(queue='notificaciones', durable=True)

    # Configurar consumo de mensajes
    canal.basic_consume(
        queue='eventos_libros',
        on_message_callback=procesar_evento_libro,
        auto_ack=False
    )

    print("[Procesador] Iniciando procesamiento de eventos...")
    canal.start_consuming()

def procesar_evento_libro(ch, method, properties, body):
    """
    Procesa un evento de libro recibido de RabbitMQ
    """
    try:
        evento = json.loads(body.decode())
```

```
        tipo_evento = evento.get('tipo')

        print(f"[Procesador] Procesando evento: {tipo_evento}")

        # L gica de procesamiento seg n tipo de evento
        if tipo_evento == 'NUEVO_LIBRO':
            notificar_usuarios_interesados(evento)
        elif tipo_evento == 'OFERTA_RECIBIDA':
            notificar_vendedor_oferta(evento)

        # Confirmar procesamiento
        ch.basic_ack(delivery_tag=method.delivery_tag)

    except Exception as e:
        print(f"[Error] No se pudo procesar evento: {e}")

def notificar_usuarios_interesados(evento):
    """
    Notifica a usuarios interesados en un nuevo libro
    """
    libro = evento['libro']
    notificacion = {
        'tipo': 'NUEVO_LIBRO_DISPONIBLE',
        'mensaje': f"       Nuevo libro: {libro['titulo']} - ${libro['
    precio']}",
        'timestamp': evento['timestamp']
    }

    enviar_notificacion(notificacion)

def enviar_notificacion(notificacion):
    """
    Env a una notificaci n a la cola de notificaciones
    """
    canal = conectar_rabbitmq()
    if canal is None:
        return

    canal.basic_publish(
        exchange='',
        routing_key='notificaciones',
        body=json.dumps(notificacion)
    )

if __name__ == "__main__":
```

```
    iniciar_procesador_libros()
```

Listing 3: Procesador de eventos

3.4. Interfaz de Usuario

Se desarrolló una interfaz gráfica utilizando Tkinter:

```
import tkinter as tk
from tkinter import ttk, scrolledtext
import json
import threading
from conexion import conectar_rabbitmq
from productor import publicar_evento_libro

class InterfazLibrosUsados:
    def __init__(self):
        self.ventana = tk.Tk()
        self.configurar_interfaz()

    def configurar_interfaz(self):
        """Configura la interfaz gr fica principal"""
        self.ventana.title("BookTrade - Plataforma de Libros Usados")
        self.ventana.geometry("700x600")
        self.ventana.configure(bg="#f8f9fa")

        self.crear_seccion_publicacion()
        self.crear_seccion_notificaciones()

    def crear_seccion_publicacion(self):
        """Crea la secci n para publicar nuevos libros"""
        frame_publicar = tk.LabelFrame(
            self.ventana,
            text="       Publicar Nuevo Libro",
            bg="#f8f9fa",
            font=("Arial", 12, "bold")
        )
        frame_publicar.pack(pady=15, padx=20, fill="x")

        # Campos del formulario
        tk.Label(frame_publicar, text="T tulo:", bg="#f8f9fa").grid(row
    =0, column=0, sticky="w", pady=5)
        self.entry_titulo = tk.Entry(frame_publicar, width=30)
        self.entry_titulo.grid(row=0, column=1, padx=10, pady=5)

        tk.Label(frame_publicar, text="G nero:", bg="#f8f9fa").grid(row
    =1, column=0, sticky="w", pady=5)
```

```
38          self.combo_genero = ttk.Combobox(
39              frame_publicar,
40              values=["Ficci n", "Ciencia Ficci n", "Fantas a", "
    Romance", "Biograf a"],
41              state="readonly",
42              width=27
43          )
44          self.combo_genero.grid(row=1, column=1, padx=10, pady=5)
45          self.combo_genero.current(0)
46
47          tk.Label(frame_publicar, text="Precio:", bg="#f8f9fa").grid(row
    =2, column=0, sticky="w", pady=5)
48          self.entry_precio = tk.Entry(frame_publicar, width=15)
49          self.entry_precio.grid(row=2, column=1, sticky="w", padx=10,
    pady=5)
50
51          # Bot n de publicaci n
52          btn_publicar = tk.Button(
53              frame_publicar,
54              text="Publicar Libro",
55              command=self.publicar_libro,
56              bg="#28a745",
57              fg="white",
58              font=("Arial", 10, "bold")
59          )
60          btn_publicar.grid(row=3, column=1, pady=15, sticky="e")
61
62      def crear_seccion_notificaciones(self):
63          """Crea la secci n para mostrar notificaciones"""
64          frame_noti = tk.LabelFrame(
65              self.ventana,
66              text="       Notificaciones en Tiempo Real",
67              bg="#f8f9fa",
68              font=("Arial", 12, "bold")
69          )
70          frame_noti.pack(pady=15, padx=20, fill="both", expand=True)
71
72          self.caja_notificaciones = scrolledtext.ScrolledText(
73              frame_noti,
74              height=20,
75              width=80,
76              font=("Consolas", 9)
77          )
78          self.caja_notificaciones.pack(pady=10, padx=10, fill="both",
    expand=True)
79
80          # Iniciar hilo para escuchar notificaciones
```

```
81          hilo_consumidor = threading.Thread(
82              target=self.escuchar_notificaciones,
83              daemon=True
84          )
85          hilo_consumidor.start()
86
87      def publicar_libro(self):
88          """Publica un nuevo libro en el sistema"""
89          datos_libro = {
90              'titulo': self.entry_titulo.get(),
91              'genero': self.combo_genero.get(),
92              'precio': self.entry_precio.get(),
93              'vendedor': 'usuario_actual'
94          }
95
96          try:
97              publicar_evento_libro('NUEVO_LIBRO', datos_libro)
98              self.mostrar_notificacion("    Libro publicado exitosamente"
    )
99
100             # Limpiar campos
101             self.entry_titulo.delete(0, tk.END)
102             self.entry_precio.delete(0, tk.END)
103
104         except Exception as e:
105             self.mostrar_notificacion(f"    Error: {str(e)}")
106
107     def escuchar_notificaciones(self):
108         """Escucha notificaciones de RabbitMQ"""
109         def callback(ch, method, properties, body):
110             try:
111                 notificacion = json.loads(body.decode())
112                 mensaje = notificacion['mensaje']
113                 self.ventana.after(0, lambda: self.mostrar_notificacion(
    mensaje))
114             except Exception as e:
115                 print(f"Error: {e}")
116
117         try:
118             canal = conectar_rabbitmq()
119             if canal:
120                 canal.queue_declare(queue='notificaciones')
121                 canal.basic_consume(
122                     queue='notificaciones',
123                     on_message_callback=callback,
124                     auto_ack=True
125                 )
```

```
                canal.start_consuming()
        except Exception as e:
            print(f"Error en consumidor: {e}")

    def mostrar_notificacion(self, mensaje):
        """Muestra una notificaci n en la caja de texto"""
        self.caja_notificaciones.insert(tk.END, f"{mensaje}\n")
        self.caja_notificaciones.see(tk.END)

    def ejecutar(self):
        """Ejecuta la aplicaci n"""
        self.ventana.mainloop()

if __name__ == "__main__":
    app = InterfazLibrosUsados()
    app.ejecutar()
```

Listing 4: Interfaz de usuario

4. Flujos de Trabajo

4.1. Publicación de Nuevo Libro

La figura 2 ilustra el flujo cuando un usuario publica un nuevo libro:

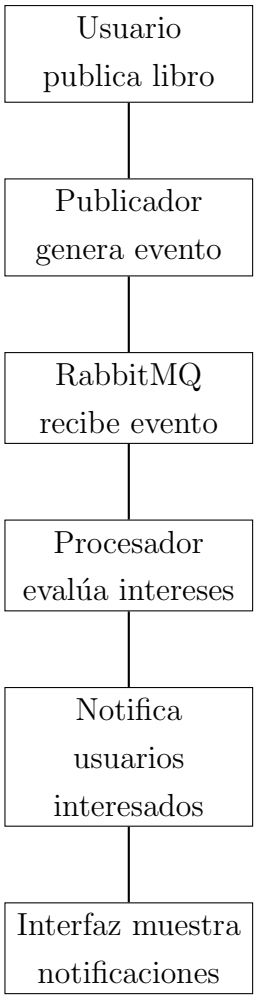

Figura 2: Flujo de publicación de nuevo libro

4.2. Tipos de Eventos

El sistema maneja los siguientes tipos de eventos principales:

- **NUEVO_LIBRO**: Cuando un usuario publica un libro disponible para venta/intercambio
- **OFERTA_RECIBIDA**: Cuando un usuario recibe una oferta por uno de sus libros
- **VENTA_COMPLETADA**: Cuando una transacción se finaliza exitosamente
- **LIBRO_AGOTADO**: Cuando un libro ya no está disponible

5. Resultados y Discusión

La implementación del patrón Publicador-Suscriptor demostró varios beneficios significativos para el sistema de libros usados:

5.1. Desacoplamiento de Componentes

El uso de RabbitMQ como intermediario permitió que los componentes del sistema evolucionen independientemente. El publicador no necesita conocer los detalles de implementación del consumidor, facilitando el mantenimiento y la escalabilidad.

5.2. Escalabilidad

El sistema puede manejar incrementos en la carga mediante:

- Adición de múltiples instancias del procesador
- Partición de colas por tipo de evento o categoría de libro
- Balanceo de carga entre consumidores

5.3. Confiabilidad

RabbitMQ proporciona garantías de entrega de mensajes, asegurando que:

- Los eventos no se pierdan durante fallos temporales
- Los mensajes se persisten hasta su procesamiento completo
- Se puede recuperar el estado del sistema después de interrupciones

5.4. Comparación con Enfoques Síncronos

La tabla 2 compara el enfoque asíncrono con implementaciones síncronas tradicionales:

Aspecto	**Enfoque Síncrono**	**Publicador-Suscriptor**
Acoplamiento	Alto	Bajo
Escalabilidad	Limitada	Alta
Tolerancia a Fallos	Baja	Alta
Complejidad	Baja	Media-Alta
Tiempo de Respuesta	Inmediato	Asíncrono

Cuadro 2: Comparación entre enfoques síncrono y Publicador-Suscriptor

6. Conclusiones

La implementación del patrón Publicador-Suscriptor utilizando RabbitMQ demostró ser una solución efectiva para el sistema de intercambio y venta de libros usados. Los principales hallazgos incluyen:

1. El patrón permite una arquitectura altamente desacoplada, facilitando el mantenimiento y la evolución del sistema.
2. La comunicación asíncrona mejora la experiencia del usuario al proporcionar notificaciones en tiempo real sin bloquear operaciones.
3. RabbitMQ ofrece una plataforma confiable para la gestión de mensajes, con características robustas de persistencia y recuperación.
4. El sistema puede escalar horizontalmente para manejar incrementos en el volumen de transacciones.

Como trabajo futuro, se recomienda explorar:

- Integración con sistemas de pago electrónico
- Implementación de mecanismos de recomendación basados en comportamiento de usuarios
- Adición de notificaciones push para dispositivos móviles
- Implementación de análisis en tiempo real sobre patrones de compra/venta

Referencias

Richards, M. (2015). *Software Architecture Patterns*. O'Reilly Media.

Hohpe, G. y Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

Videla, A. y Williams, J.J.W. (2012). *RabbitMQ in Action: Distributed Messaging for Everyone*. Manning Publications.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Zhao, L., Wang, X. y Chen, K. (2020). E-commerce platforms for secondhand goods: A systematic literature review. *Journal of Cleaner Production*, 256, 120325.

RabbitMQ Documentation (2023). *Official RabbitMQ Documentation*. Disponible en: `https://www.rabbitmq.com/documentation.html`

Python Software Foundation (2023). *Python Documentation*. Disponible en: `https://docs.python.org/3/`

A. Instrucciones de Instalación y Ejecución

A.1. Requisitos del Sistema

- Python 3.8 o superior
- RabbitMQ 3.8 o superior
- Bibliotecas Python: pika, tkinter

A.2. Configuración

1. Instalar RabbitMQ:

```
# En Ubuntu/Debian:
sudo apt-get install rabbitmq-server
```

2. Instalar dependencias Python:

```
pip install pika
```

3. Ejecutar RabbitMQ:

```
sudo systemctl start rabbitmq-server
```

4. Ejecutar los componentes en orden:

```
# Terminal 1 - Procesador
python procesador.py

# Terminal 2 - Interfaz
python consumidor.py
```

B. Estructura del Proyecto

La estructura de archivos del proyecto es la siguiente:

```
booktrade-system/
 conexion.py          # Conexión a RabbitMQ
 productor.py         # Publicador de eventos
 procesador.py        # Procesador de eventos
 consumidor.py        # Interfaz de usuario
 requirements.txt     # Dependencias
 README.md           # Documentación
```

C. Ejemplos de Uso

C.1. Ejemplo de Publicación de Libro

```
# Desde la interfaz gráfica:
1. Ingresar título: "Cien años de soledad"
2. Seleccionar género: "Ficción"
3. Ingresar precio: "15000"
4. Click en "Publicar Libro"
```

C.2. Ejemplo de Notificación Recibida

```
[2024-01-15T10:30:00]  Nuevo libro disponible:
Cien años de soledad - $15000
```