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

Juan David Jimenez Castellanos,Sebastian David Gonzalez Hernandez, Rodrigo Alberto Osorio Gaona

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:

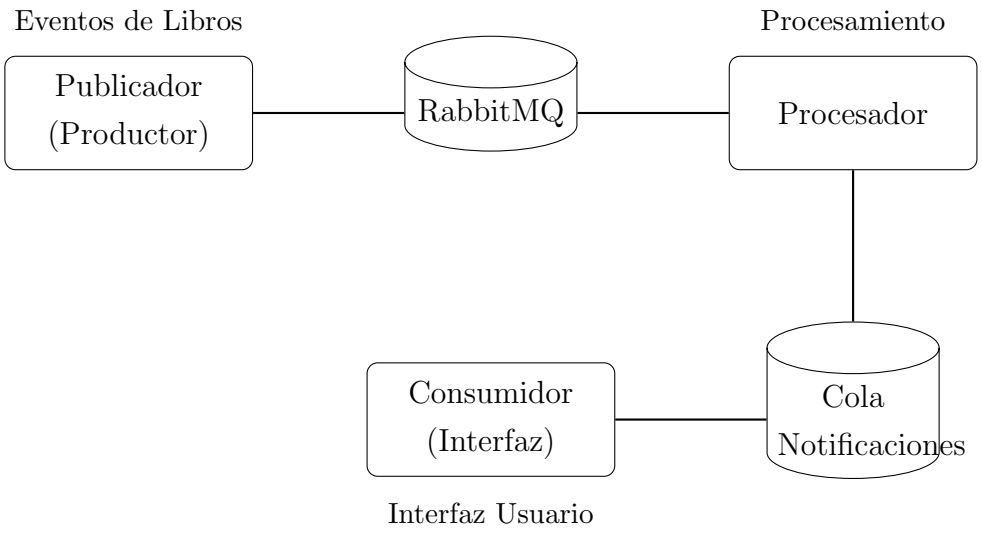

Figura 1: Arquitectura del sistema Publicador-Suscriptor para libros usados

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. Casos de Uso

3.1. Caso de Uso: Publicar Nuevo Libro

Iniciador	Vendedor
Otros actores	[leftmargin=*,nosep] ▪ Compradores interesados (reciben notificaciones) ▪ Sistema de Notificaciones (envía alertas)
Precondiciones	[leftmargin=*,nosep] ▪ El vendedor debe estar registrado y autenticado en el sistema ▪ El vendedor debe tener al menos un método de pago verificado ▪ El sistema RabbitMQ debe estar en ejecución

Cuadro 2: Caso de uso: Publicar Nuevo Libro - Información general

3.1.1. Flujo Básico

Actor	**Sistema**
1. El vendedor selecciona la opción "Publicar Libro"	
	2. El sistema muestra el formulario de publicación con campos: título, autor, género, precio, condición, descripción y fotos
3. El vendedor completa todos los campos obligatorios y hace clic en "Publicar"	
	4. El sistema valida los datos ingresados (precio positivo, título no vacío, etc.)
	5. El Publicador (Productor) genera el evento "NUEVO_LIBROçon los datos del libro y lo publica en la cola .evento_libros"de RabbitMQ
	6. El Procesador recibe el evento, lo procesa y identifica a los compradores interesados en ese género o autor
	7. El sistema publica notificaciones en la cola "notificaciones"para los usuarios interesados
	8. El sistema confirma la publicación exitosa y muestra el libro en el catálogo

Cuadro 3: Caso de uso: Publicar Nuevo Libro - Flujo básico

3.1.2. Flujos Alternativos

Flujo alternativo 1	**Validación fallida** En el paso 4, si el sistema detecta datos inválidos: [leftmargin=*,nosep] ▪ El sistema muestra mensajes de error específicos ▪ El vendedor debe corregir los datos ▪ El caso de uso continúa desde el paso 3
Flujo alternativo 2	**Publicación con fotos opcionales** En el paso 3, si el vendedor no sube fotos: [leftmargin=*,nosep] ▪ El sistema asigna una imagen por defecto del libro ▪ El sistema muestra advertencia sobre menor visibilidad ▪ El flujo continúa normalmente
Flujo alternativo 3	**RabbitMQ no disponible** En el paso 5, si RabbitMQ no está disponible: [leftmargin=*,nosep] ▪ El sistema guarda el libro en estado "pendiente de notificación" ▪ El sistema programa reintento automático ▪ El vendedor recibe confirmación condicional
Flujo alternativo 4	**Publicación rápida (campos mínimos)** El vendedor selecciona "Publicación Rápida": [leftmargin=*,nosep] ▪ Solo completa título, género y precio ▪ El sistema asigna valores por defecto para otros campos ▪ El flujo se reduce a 4 pasos principales

Cuadro 4: Caso de uso: Publicar Nuevo Libro - Flujos alternativos

3.1.3. Poscondiciones

- El libro queda publicado y visible en el catálogo
- Los compradores interesados reciben notificaciones
- El sistema registra el evento de publicación para analytics
- El libro está disponible para ofertas y ventas

3.2. Caso de Uso: Recibir Oferta de Libro

Iniciador	Comprador
Otros actores	[leftmargin=*,nosep] - Vendedor (recibe notificación de oferta) - Sistema de Notificaciones
Precondiciones	[leftmargin=*,nosep] - El comprador debe estar registrado y autenticado - El libro debe estar disponible y activo en el catálogo - El comprador no debe ser el propietario del libro

Cuadro 5: Caso de uso: Recibir Oferta de Libro - Información general

3.2.1. Flujo Básico

Actor	**Sistema**
1. El comprador selecciona un libro y hace clic en "Hacer Oferta"	
	2. El sistema muestra el formulario de oferta con precio sugerido y campos para mensaje opcional
3. El comprador ingresa el monto de la oferta y hace clic en .ºnviar Oferta"	
	4. El sistema valida que la oferta sea mayor que 0 y que el comprador tenga fondos disponibles
	5. El Publicador genera el evento .ºFERTA_RECIBIDAz lo publica en RabbitMQ
	6. El Procesador recibe el evento y notifica al vendedor mediante la cola "notificaciones"
	7. El sistema confirma al comprador que la oferta fue enviada

Cuadro 6: Caso de uso: Recibir Oferta de Libro - Flujo básico

3.2.2. Flujos Alternativos

Flujo alternativo 1	**Oferta rechazada por validación** En el paso 4, si la oferta es inválida: [leftmargin=*,nosep] ▪ El sistema muestra error y solicita corrección ▪ El comprador puede ajustar el monto
Flujo alternativo 2	**Oferta automáticamente aceptada** Si la oferta iguala o supera el precio de venta directa: [leftmargin=*,nosep] ▪ El sistema procesa la venta automáticamente ▪ Se genera evento "VENTA_COMPLETADA"

Cuadro 7: Caso de uso: Recibir Oferta de Libro - Flujos alternativos

3.2.3. Poscondiciones

- La oferta queda registrada en el sistema
- El vendedor recibe la notificación de oferta
- El libro muestra indicador de .ºferta pendiente"
- El comprador ve la oferta en su historial

3.3. Caso de Uso: Procesar Notificaciones de Libros

Iniciador	Sistema (RabbitMQ)
Otros actores	[leftmargin=*,nosep] ▪ Usuarios (reciben notificaciones) ▪ Interfaz gráfica (muestra notificaciones)
Precondiciones	[leftmargin=*,nosep] ▪ RabbitMQ debe estar ejecutándose con las colas configuradas ▪ El procesador debe estar escuchando eventos ▪ Los usuarios deben tener preferencias de notificación configuradas

Cuadro 8: Caso de uso: Procesar Notificaciones de Libros - Información general

3.3.1. Flujo Básico

Actor	**Sistema**
	1. El Publicador genera un evento (NUEVO_LIBRO, OFERTA_RECIBIDA, etc.) y lo envía a RabbitMQ
	2. RabbitMQ almacena el evento en la cola .evento_libros"
	3. El Procesador consume el evento de la cola
	4. El Procesador analiza el tipo de evento y aplica reglas de negocio
	5. El Procesador identifica los usuarios destinatarios de la notificación
	6. El Procesador publica las notificaciones en la cola "notificaciones"
	7. El Consumidor (Interfaz) recibe las notificaciones y las muestra a los usuarios

Cuadro 9: Caso de uso: Procesar Notificaciones de Libros - Flujo básico

3.3.2. Flujos Alternativos

Flujo alternativo 1	**Usuario no conectado** Si el usuario destinatario no está conectado: [leftmargin=*,nosep] ▪ El sistema almacena la notificación para mostrarla después ▪ Se envía notificación por email si está configurado
Flujo alternativo 2	**Procesamiento por lote** Para eventos de alta frecuencia: [leftmargin=*,nosep] ▪ El procesador agrupa notificaciones similares ▪ Envía resúmenes periódicos en lugar de notificaciones individuales

Cuadro 10: Caso de uso: Procesar Notificaciones de Libros - Flujos alternativos

3.3.3. Poscondiciones

- Los usuarios reciben notificaciones relevantes en tiempo real
- El sistema registra el delivery de notificaciones
- Las notificaciones no entregadas se almacenan para reintento
- El estado de los eventos se actualiza a "procesado"

4. Implementación

4.1. 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

4.2. 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

4.3. 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')
```

```
34
35          print(f"[Procesador] Procesando evento: {tipo_evento}")
36
37          # L gica de procesamiento seg n tipo de evento
38          if tipo_evento == 'NUEVO_LIBRO':
39              notificar_usuarios_interesados(evento)
40          elif tipo_evento == 'OFERTA_RECIBIDA':
41              notificar_vendedor_oferta(evento)
42
43          # Confirmar procesamiento
44          ch.basic_ack(delivery_tag=method.delivery_tag)
45
46      except Exception as e:
47          print(f"[Error] No se pudo procesar evento: {e}")
48
49  def notificar_usuarios_interesados(evento):
50      """
51      Notifica a usuarios interesados en un nuevo libro
52      """
53      libro = evento['libro']
54      notificacion = {
55          'tipo': 'NUEVO_LIBRO_DISPONIBLE',
56          'mensaje': f"       Nuevo libro: {libro['titulo']} - ${libro['
    precio']}",
57          'timestamp': evento['timestamp']
58      }
59
60      enviar_notificacion(notificacion)
61
62  def enviar_notificacion(notificacion):
63      """
64      Env a una notificaci n a la cola de notificaciones
65      """
66      canal = conectar_rabbitmq()
67      if canal is None:
68          return
69
70      canal.basic_publish(
71          exchange='',
72          routing_key='notificaciones',
73          body=json.dumps(notificacion)
74      )
75
76  if __name__ == "__main__":
77      iniciar_procesador_libros()
```

Listing 3: Procesador de eventos

4.4. 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)
        self.combo_genero = ttk.Combobox(
            frame_publicar,
            values=["Ficci n", "Ciencia Ficci n", "Fantas a", "
    Romance", "Biograf a"],
            state="readonly",
```

```
            width=27
        )
        self.combo_genero.grid(row=1, column=1, padx=10, pady=5)
        self.combo_genero.current(0)

        tk.Label(frame_publicar, text="Precio:", bg="#f8f9fa").grid(row
=2, column=0, sticky="w", pady=5)
        self.entry_precio = tk.Entry(frame_publicar, width=15)
        self.entry_precio.grid(row=2, column=1, sticky="w", padx=10,
pady=5)

        # Bot n de publicaci n
        btn_publicar = tk.Button(
            frame_publicar,
            text="Publicar Libro",
            command=self.publicar_libro,
            bg="#28a745",
            fg="white",
            font=("Arial", 10, "bold")
        )
        btn_publicar.grid(row=3, column=1, pady=15, sticky="e")

    def crear_seccion_notificaciones(self):
        """Crea la secci n para mostrar notificaciones"""
        frame_noti = tk.LabelFrame(
            self.ventana,
            text="       Notificaciones en Tiempo Real",
            bg="#f8f9fa",
            font=("Arial", 12, "bold")
        )
        frame_noti.pack(pady=15, padx=20, fill="both", expand=True)

        self.caja_notificaciones = scrolledtext.ScrolledText(
            frame_noti,
            height=20,
            width=80,
            font=("Consolas", 9)
        )
        self.caja_notificaciones.pack(pady=10, padx=10, fill="both",
expand=True)

        # Iniciar hilo para escuchar notificaciones
        hilo_consumidor = threading.Thread(
            target=self.escuchar_notificaciones,
            daemon=True
        )
        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                 )
126                 canal.start_consuming()
127         except Exception as e:
128             print(f"Error en consumidor: {e}")
129
130     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

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 11 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 11: 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
```