Message Queue

Adrian Monge Mairena

Jimena Castillo Campos

Escuela de Ingeniería en Computadores, Tecnológico de Costa Rica

Grupo 1, Algoritmos y Estructuras de Datos I

Issac Ramírez Herrera

28 de marzo de 2025

Message Queue

En la actualidad, los sistemas distribuidos desempeñan un papel fundamental en el desarrollo de software, permitiendo la ejecución de programas en diferentes máquinas, ya sean físicas o virtuales. Un ejemplo representativo de un sistema distribuido es Netflix, cuya arquitectura se compone de diversos servicios especializados, cada uno con una responsabilidad específica, como la transmisión de contenido multimedia, el sistema de recomendaciones y el análisis de estadísticas de uso.

La comunicación juega un papel fundamental en la ejecución de los programas, Al establecer una comunicación, una de las opciones a utilizar es la de punto a punto. A pesar de ser muy sencilla no siempre es la óptima, es por eso por lo que otra técnica utilizada en la industria es un *middleware*. Este actúa como intermediario para comunicar dos sistemas independientes uno del otros sin tener que conocer detalles específicos del otro, brindando así una mayor flexibilidad.

El presente proyecto tiene como objetivo desarrollar un *middleware* orientado a colas de mensajes denominado *MQBroker*. Este sistema se encargará de la gestión de colas y la transmisión de mensajes utilizando una topología basada en el patrón publicador/subscriptor. Para su implementación, se empleará un *socket* servidor que recibirá y gestionará las peticiones de las aplicaciones emisoras y consumidoras de mensajes. A través de este enfoque, se busca mejorar la escalabilidad y eficiencia de la comunicación en sistemas distribuidos modernos.

Tabla de Contenidos

Descripción del problema

En los sistemas distribuidos, la comunicación eficiente entre aplicaciones independientes es un desafío crítico, especialmente cuando estas operan en diferentes máquinas y deben intercambiar información de manera confiable. La comunicación punto a punto, aunque sencilla, presenta problemas de escalabilidad, ya que cada nueva conexión agrega complejidad y carga a la infraestructura. A medida que el número de aplicaciones y transacciones crece, la gestión de estas conexiones se vuelve insostenible, generando cuellos de botella, dependencia directa entre servicios y dificultad para manejar fallos o tiempos de respuesta variables. Además, la falta de un mecanismo que permita el desacoplamiento entre emisores y receptores de información dificulta la flexibilidad y evolución del sistema.

Descripción de la Solución

Primera Parte del Proyecto (MQBroker)

1. **Activación del socket:** la implementación de este requerimiento se basa en la activación de un cliente de mensajería *MQClient* que establece una conexión a un servidor en la dirección IP 127.0.0.1 y puerto 5000. El programa espera 2 segundos antes de la inicialización para asegurarse de que el MQBroker esté disponible. Se genera un identificador único *appId* para la sesión del cliente. Se crea un cliente de mensajería MQClient con los parámetros adecuados. Se define un tema *Topic* llamado *TestTopic* y un mensaje *Message* con el contenido *Hello, World!*.

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        // Agregar un retraso para dar tiempo a que MQBroker se inicie
        Thread.Sleep(2000); // 2 segundos

        Guid appId = Guid.NewGuid();
        MQClient client = new MQClient("127.0.0.1", 5000, appId);

        Topic topic = new Topic("TestTopic");
        Message message = new Message("Hello, World!");
```

1.1. Limitaciones: para esta solución algunas de las limitantes existen en la restricción de entornos locales, en la falta de reintentos en caso de fallo al establecer la conexión y que esta no maneja múltiples clientes simultáneamente.

1.2. Problemas encontrados: en algunos casos el MQBroker no está listo cuando se ejecuta el cliente.

2. **Petición Subscribe:** se recibe una solicitud con el *appId* en el formato *Guid* y el nombre del tema al que se quiere suscribir.

```
// Suscribirse al tema
if (client.Subscribe(topic))
{
    Console.WriteLine("Suscrito al tema: " + topic.Name);
}
else
{
    Console.WriteLine("Error al suscribirse al tema: " + topic.Name);
}
```

2.1. Alternativas: utilizar solo una validación para suscribir al tema seleccionado.

2.2. Limitaciones: se restringe el uso a entornos locales.

2.3. Problemas encontrados: si el MQBroker no está disponible, el programa podría fallar.

2.4. Aspectos relevantes: al utilizar *Guid. NewGuid()* para generar un identificador único de sesión, asegurando que cada cliente tenga una identidad propiedad.

3. **Petición Unsubcribe:** se recibe una solicitud con el *appId* en formato *Guid* y el nombre del tema. Después se verifica si el *appId* ya está suscrito a dicho tema, si no está suscrito indica un mensaje de error, pero si se encuentra suscrito al tema indica un mensaje cancelación de suscripción y se elimina el tema de la cola.

```
// Desuscribirse del tema
if (client.Unsubscribe(topic))
{
    Console.WriteLine("Desuscrito del tema: " + topic.Name);
}
else
{
    Console.WriteLine("Error al desuscribirse del tema: " + topic.Name);
}
```

3.1. Alternativas: se evaluó la posibilidad de almacenar los mensajes en una base de datos para su recuperación posterior, pero esto fue descartado en esta fase del desarrollo.

3.2. Limitaciones: se restringe a entornos locales.

3.3. Problemas encontrados: si el MQBroker no está disponible, el programa podría fallar.

4. **Petición Publish:** recibe la solicitud con *appId* formato *Guid* para publicar un mensaje, además, verifica si el tema existe y de ser así coloca le mensaje en la cola de los suscriptores del tema.

```
// Publicar un mensaje en el tema
if (client.Publish(message, topic))
{
    Console.WriteLine("Mensaje publicado en el tema: " + topic.Name);
}
else
{
    Console.WriteLine("Error al publicar el mensaje en el tema: " + topic.Name);
}
```

4.1. Alternativas:

4.2. Limitaciones: no existen reintentos de conexión al MQBroker si se desconecta por algún motivo.

4.3. Problemas: si el MQBroker no está disponible, el programa podría fallar.

5. **Petición Receive:** recibe una solicitud del *appId* que quiere recibir un mensaje. Después se verifica si está suscrito al tema, de ser así verifica si existe una cola de mensajes para el tema determinado. Al finalizar las verificaciones se extrae el mensaje siguiendo la estructura FIFO y lo envía a través del socket.

```
// Recibir un mensaje del tema
try
{
    Message receivedMessage = client.Receive(topic);
    Console.WriteLine("Mensaje recibido: " + receivedMessage.Content);
}
catch (Exception ex)
{
    Console.WriteLine("Error al recibir el mensaje: " + ex.Message);
}
```

5.1. Limitaciones: la implementación no maneja múltiples clientes simultáneamente.

5.2. Problemas encontrados: no existen reintentos de conexión al MQBroker si se desconecta por algún motivo.

Segunda Parte del Proyecto (MQClient)

1. **Constructor MQClient:** para implementar el constructor de esta clase, se definieron 3 atributos privados y se estableció que el constructor recibe los valores como parámetros y los asigna a los atributos internos de la clase.

```
/// <summary>
/// Clase que representa un cliente de MQBroker.
/// </summary>
7 references
public class MQClient
{
    private string ip;
    private int port;
    private Guid appId;

    /// <summary>
    /// Constructor que crea un nuevo MQClient.
    /// </summary>
    /// <param name="ip">IP donde está escuchando el MQBroker.</param>
    /// <param name="port">Puerto donde está escuchando el MQBroker.</param>
    /// <param name="appId">AppID generado por la aplicación que utiliza la biblioteca MQClient.</param>
    3 references
    public MQClient(string ip, int port, Guid appId)
    {
        this.ip = ip;
        this.port = port;
        this.appId = appId;
    }
```

   1.1. Alternativas: se pensó en implementar atributos públicos en lugar de privados.

   1.2. Limitaciones: no se validan los atributos para asegurarse de que los datos sean válidos.

   1.3. Problemas encontrados: si la *ip* es proporcionada por el usuario sin validación, puede haber ataques como inyección de datos en logs o intentos de conexión a direcciones maliciosas.

2. **Subscribe:** El método toma un objeto *Topic* como parámetro, que encapsula el nombre del tema al que se desea suscribir. El mensaje que se envía al MQBroker sigue el formato SUBSCRIBE|{appId}|{topic.Name}. Luego, el mensaje se envía al servidor a través de un método *SendMessage*, que maneja la conexión y el envío del mensaje. Si el mensaje se envía correctamente, se devuelve true, de lo contrario, se devuelve false.

```
/// <summary>
/// Envía la petición Subscribe al MQBroker mediante un socket de tipo cliente.
/// </summary>
/// <param name="topic">Objeto Topic que encapsula el tema al que se quiere suscribir.</param>
/// <returns>True si la suscripción se realizó, false en caso contrario.</returns>
3 references
public bool Subscribe(Topic topic)
{
    string message = $"SUBSCRIBE|{appId}|{topic.Name}";
    return SendMessage(message);
}
```

2.1. Alternativas: validaciones previas a enviar el mensaje, se puede agregar una validación previa para asegurarse de que el *topic.Name* no sea nulo ni vacío.

2.2. Limitaciones: el método Subscribe no verifica ni maneja las respuestas del servidor (más allá de la comunicación exitosa). Esto podría ser importante para saber si la suscripción fue aceptada correctamente por el servidor o si hubo algún error específico.

2.3. Problemas encontrados: si el servidor responde con un error (por ejemplo, debido a un problema de autenticación o de configuración del tema), el código actual no captura ni maneja esos errores específicos.

3. **Unsubcribre:** el método toma un objeto *Topic*, que encapsula el nombre del tema del que se desea eliminar la suscribir. Se construye un mensaje en el formato UNSUBSCRIBE|{appId}|{topic.Name}. El mensaje se envía al servidor MQBroker a través del método *SendMessage*, que gestiona la conexión y el envío del mensaje. Si el mensaje se envía correctamente, el método retorna true. Si no, retorna false.

```
/// <summary>
/// Envía la petición Unsubscribe al MQBroker mediante un socket de tipo cliente.
/// </summary>
/// <param name="topic">Objeto Topic que encapsula el tema del que se quiere desuscribir.</param>
/// <returns>True si la desuscripción se realizó, false en caso contrario.</returns>
3 references
public bool Unsubscribe(Topic topic)
{
    string message = $"UNSUBSCRIBE|{appId}|{topic.Name}";
    return SendMessage(message);
}
```

3.1. Alternativas: podría añadirse una validación para comprobar que el topic.Name no esté vacío antes de intentar eliminar la suscripción.

3.2. Limitaciones: al igual que con el método Subscribe, el método *Unsubscribe* no verifica la respuesta del servidor, lo que limita la capacidad para identificar si la eliminación de la suscripción fue exitosa o si hubo un error en el servidor.

3.3. Problemas encontrados: si ocurre una excepción al enviar el mensaje (como una desconexión repentina), el cliente solo imprimirá un mensaje en la consola y retornará *false*, lo que podría no ser suficiente para la gestión adecuada de errores en una aplicación real.

4. **Publish:** el método toma dos parámetros: un objeto *Message* que encapsula el contenido del mensaje a publicar y un objeto *Topic* que representa el tema al que se publicará el mensaje. El mensaje que se envía al MQBroker sigue el formato PUBLISH|{appId}|{topic.Name}|{message.Content}. El mensaje se envía al servidor a través del método *SendMessage*. Si el mensaje se envía correctamente, se devuelve *true*; de lo contrario, se devuelve *false*.

```
/// <summary>
/// Envía la petición Publish al MQBroker mediante un socket de tipo cliente.
/// </summary>
/// <param name="message">Objeto Message que encapsula el contenido de la publicación.</param>
/// <param name="topic">Objeto Topic que encapsula el tema al que se desea publicar.</param>
/// <returns>True si el mensaje se pudo colocar, false en caso contrario.</returns>
3 references
public bool Publish(Message message, Topic topic)
{
    string msg = $"PUBLISH|{appId}|{topic.Name}|{message.Content}";
    return SendMessage(msg);
}
```

4.1. Alternativas: podría implementarse una verificación más detallada sobre la respuesta del servidor.

4.2. Limitaciones: no maneja conexiones persistentes, cada llamada a *SendMessage* establece una nueva conexión. Esto puede ser ineficiente si se realizan muchas publicaciones consecutivas, ya que la conexión se abre y se cierra para cada mensaje.

4.3. Problemas encontrados: si el sistema necesita publicar muchos mensajes en un corto período de tiempo, la implementación actual puede generar problemas de rendimiento debido a la necesidad de abrir y cerrar una nueva conexión para cada mensaje.

5. **Receive:** el método *Receive* toma un objeto *Topic* como parámetro, que encapsula el tema del cual se desea recibir un mensaje. Se construye un mensaje que se envía al MQBroker con el formato RECEIVE|{appId}|{topic.Name}. Luego, se llama a *SendMessage* para enviar el mensaje y obtener la respuesta del servidor. Si el mensaje se envía correctamente, la respuesta se divide usando el carácter "|", y si la respuesta es "OK", el contenido del mensaje recibido se encapsula en un objeto *Message*.Si el servidor responde con un error, o si no se pudo enviar el mensaje, se lanza una excepción.

```
/// <summary>
/// Envia la petición Receive al MQBroker mediante un socket de tipo cliente.
/// </summary>
/// <param name="topic">Objeto Topic que encapsula el tema del que se desea obtener un mensaje.</param>
/// <returns>Objeto Message con el contenido del mensaje recibido.</returns>
/// <exception cref="Exception">Lanza una excepción si no se puede enviar el mensaje o si hay un error en la respuesta del servidor.</exception>
3 references
public Message Receive(Topic topic)
{
    string message = $"RECEIVE|{appId}|{topic.Name}";
    if (SendMessage(message, out string response))
    {
        var parts = response.Split('|');
        if (parts[0] == "OK")
        {
            return new Message(parts[1]);
        }
        throw new Exception("Respuesta de error del servidor: " + response);
    }
    throw new Exception("No se pudo enviar el mensaje");
}
```

5.1. Alternativas: se podría ampliar la verificación para manejar más casos.

5.2. Limitaciones: si el servidor MQBroker está inactivo o hay problemas de red, el método solo captura una excepción general *IOException* o *Exception*, lo que no proporciona información detallada sobre el problema.

5.3. Problemas encontrados: al capturar excepciones generales *Exception*, se pierde el detalle de las excepciones específicas que podrían ser útiles para un diagnóstico más preciso, como problemas con el formato de la respuesta o fallos de conexión.

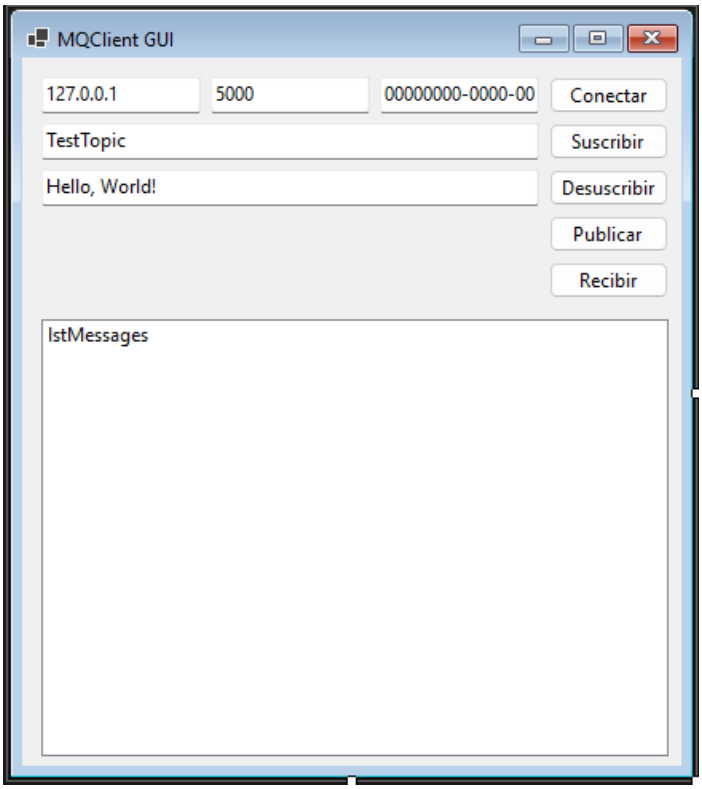
Tercera Parte del Proyecto(Interfaz Gráfica)

1. **GUI:** se implementó utilizando "Windows Forms". Se utilizan *TextBox* para capturar la dirección IP (txtIP), el puerto (txtPort), el identificador de la aplicación (txtAppID), el tema (txtTopic), y el mensaje a publicar (txtMessage). Estos controles permiten a los usuarios ingresar los datos necesarios para interactuar con el servidor MQBroker. Además, se usan *Button* para las acciones principales: Conectar (btnConnect), Suscribir (btnSubscribe), Desuscribir (btnUnsubscribe), Publicar (btnPublish), y Recibir (btnReceive). Cada botón tiene un evento *Click* asociado que ejecuta la lógica correspondiente. Cuenta con un *ListBox* para mostrar los mensajes recibidos desde el servidor.

```
namespace GUI
{
    4 references
    partial class Form1
    {
        /// <summary>
        ///  Contenedor de componentes requerido.
        /// </summary>
        private System.ComponentModel.IContainer components = null;
        private System.Windows.Forms.TextBox txtIP;
        private System.Windows.Forms.TextBox txtPort;
        private System.Windows.Forms.TextBox txtAppID;
        private System.Windows.Forms.TextBox txtTopic;
        private System.Windows.Forms.TextBox txtMessage;
        private System.Windows.Forms.Button btnConnect;
        private System.Windows.Forms.Button btnSubscribe;
        private System.Windows.Forms.Button btnUnsubscribe;
        private System.Windows.Forms.Button btnPublish;
        private System.Windows.Forms.Button btnReceive;
        private System.Windows.Forms.ListBox lstMessages;

        /// <summary>
        ///  Limpiar los recursos que se están utilizando.
        /// </summary>
        /// <param name="disposing">true si los recursos administrados deben ser eliminados; de lo contrario, false.</param>
        0 references
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        Código generado por el Diseñador de Windows Forms
    }
}
```



1.1. Alternativas: la interfaz pudo haberse implementado como una aplicación web utilizando tecnologías como ASP.NET Core.

1.2. Limitaciones: el diseño de la interfaz es bastante básico y no tiene características avanzadas en términos de experiencia de usuario.

1.3. Problemas encontrados: si el servidor MQBroker no está disponible o la conexión falla, no hay un manejo adecuado de errores en la interfaz. El usuario no sabría si algo salió mal, ya que no se muestra un mensaje claro o una notificación de error.

Diagrama UML