

SISTEMAS DISTRIBUIDOS:
PRÁCTICA 1

Fun with Queues

Implementación de un SD
mediante sockets y Kafka

Ramón Juárez Cánovas
Héctor García García
2021-2022

48729547E
48790545R

Índice

1.	Introducción.....	2
2.	Componentes.	2
2.1.	Sistema central.....	2
2.1.1.	FWQ_Engine.....	2
2.1.2.	FWQ_Registry.....	6
2.1.3.	Base de datos.	9
2.2.	Servidor de Tiempos de Espera.	10
2.2.3.	FWQ_WaitingTimeServer.....	10
2.3.	Visitantes.	12
2.3.1.	FWQ_Visitor.	12
2.4.	Sensores.	15
2.4.1.	FWQ_Sensor.	15
3.	Guía de despliegue.....	16
4.	Capturas de Ejecución.	17

1. Introducción.

Se procede al desarrollo de un sistema distribuido mediante el uso de sockets y Kafka. El sistema consistirá en un total de cinco módulos independientes, implementados de forma que funcionen de manera concurrente y que se puedan interconectar entre sí en máquinas pertenecientes a la misma red. El programa está desarrollado en su totalidad en .NET (C#).

2. Componentes.

2.1. Sistema central.

El sistema central de la aplicación estará compuesto por un total de tres elementos, un módulo de registro de usuarios, un módulo que implementa la lógica del sistema y, finalmente, una base de datos en la que almacenaremos información importante para la correcta ejecución.

2.1.1. FWQ_Engine.

El Engine será el módulo encargado de almacenar la lógica del sistema. Dado que esto es así, será el elemento en el que nos encontraremos con más código. En él, contaremos con tres archivos de código que nos permiten su ejecución por consola, siendo uno de ellos una clase auxiliar empleada para el intercambio de información mediante sockets, la cual veremos repetida en todos los módulos que requieran del uso de este tipo de conexión.

El archivo que almacena la clase *Program.cs* contiene, como el resto de los ficheros con su mismo nombre que veremos durante el desarrollo del resto de componentes, un método *main* que será el que se ejecute cuando lancemos el programa. En este caso, contamos con una pequeña implementación que nos permite obtener un total de cinco parámetros (IP y puerto del Broker, número máximo de visitantes e IP y puerto del servidor de tiempos de espera), seguido de la creación de un hilo que escuchará la información que reciba del gestor de colas, así como un bucle que solicita cada X tiempo los tiempos de espera al WTS (*Waiting Time Server*).

```
static void Main(string[] args)
{
    string ipBroker;
    string puertoBroker;
    string maxVisitantes;
    string ipTS;
    string puertoTS;

    if (args.Length == 6)
    {
        ipBroker = args[1];
        puertoBroker = args[2];
        maxVisitantes = args[3];
        ipTS = args[4];
        puertoTS = args[5];

        Console.WriteLine("Obtenidos datos necesarios.");

        Engine engine = new Engine(ipBroker, puertoBroker, maxVisitantes, ipTS, puertoTS);
        Thread th1 = new Thread(engine.SolicitudAccesoKafka);
        th1.Start();

        while (true)
        {
            engine = new Engine(ipBroker, puertoBroker, maxVisitantes, ipTS, puertoTS);
            engine.StartTSConexion();
            Thread.Sleep(5 * 1000);
        }
    }
    else
    {
        Console.WriteLine("Los parámetros introducidos deben ser 5.");
    }
}
```

El núcleo de este elemento es la clase Engine, de la que comentaremos la información más relevante.

En primer lugar, tenemos los atributos de la clase. Los primeros siete atributos se emplean para almacenar información necesaria para la activación del cliente de sockets, así como el atributo `Socket s_ClienteTS`. Los atributos `maxVisitantes`, `visitantesActuales`, `numAtracciones` y `mapaData` almacenan información acerca del sistema, `pconfig` y `cconfig` contienen información para desarrollar las conexiones por Kafka tanto como productor como de consumidor y el atributo `path` almacena la ruta del fichero que emplearemos como base de datos.

```
class Engine
{
    private static ManualResetEvent connectDone = new ManualResetEvent(false);
    private static ManualResetEvent sendDone = new ManualResetEvent(false);
    private static ManualResetEvent receiveDone = new ManualResetEvent(false);

    private static String response = String.Empty;

    IPEndPoint endPointTimeServer;
    IPAddress ipAddrTimeServer;
    IPHostEntry hostTimeServer;

    int maxVisitantes;
    int visitantesActuales;

    static Socket s_ClienteTS;
    static ProducerConfig pconfig;
    static ConsumerConfig cconfig;
    static String path = Path.GetFullPath("..\\..\\..\\..\\..\\mapa.txt");

    static int numAtracciones = 5;

    String[,] mapaData;
}
```

El constructor de la clase inicializa la mayoría de estos atributos.

```
public Engine(String ipBroker, String puertoBroker, String maximoVisitantes, String ipTimeServer, String puertoTimeServer)
{
    hostTimeServer = Dns.GetHostEntry(ipTimeServer);
    ipAddrTimeServer = hostTimeServer.AddressList[0];
    int puerto = Int32.Parse(puertoTimeServer);
    endPointTimeServer = new IPEndPoint(ipAddrTimeServer, puerto);

    s_ClienteTS = new Socket(ipAddrTimeServer.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
    maxVisitantes = Int32.Parse(maximoVisitantes);
    visitantesActuales = 0;

    pconfig = new ProducerConfig
    {
        BootstrapServers = ipBroker + ":" + puertoBroker,
        SecurityProtocol = SecurityProtocol.Plaintext
    };

    cconfig = new ConsumerConfig
    {
        BootstrapServers = ipBroker + ":" + puertoBroker,
        SecurityProtocol = SecurityProtocol.Plaintext,
        GroupId = "my-group2"
    };

    StreamReader leer = new StreamReader(path);
    mapaData = new String[numAtracciones, 4];
    String cadena;
    for (int i = 0; (cadena = leer.ReadLine()) != null; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            mapaData[i, j] = cadena.Split(';')[j];
        }
    }
    leer.Close();
}
```

Los métodos *StartTSConexion()*, *StopClient()*, *ConnectCallback(IAsyncResult ar)*, *Recieve(Socket client)*, *RecieveCallback(IAsyncResult ar)*, *Send(Socket client, String data)* y *SendCallback(IAsyncResult ar)* son los métodos que se emplean para la conexión por sockets con el servidor de tiempos de espera. Estos métodos están implementados en de igual forma en los componentes que requieran el uso de sockets. La única diferencia sustancial la encontramos en el método *StartTSConexion()*, el cual almacena el *String* recibido por parte del WTS en la variable *mapaData*. Este *String* cuenta con el siguiente formato, el cual se procesará con el método *AlmacenarTiemposDeEspera(String res)*:

```
idAtraccion1;tiempoDeEspera1;\n
idAtraccion2;tiempoDeEspera2;\n
...
idAtraccionN;tiempoDeEsperaN;\n
```

```
public void StartTSConexion()
{
    // Connect to a remote device.
    try
    {
        Console.CancelKeyPress += new ConsoleCancelEventHandler(StopClient);
        // Connect to the remote endpoint.
        s_ClienteTS.BeginConnect(endpointTimeServer, new AsyncCallback(ConnectCallback), s_ClienteTS);
        connectDone.WaitOne();

        // Send test data to the remote device.
        String enviar = "Solicitud de datos.";
        Send(s_ClienteTS, enviar);
        sendDone.WaitOne();

        // Receive the response from the remote device.
        Receive(s_ClienteTS);
        receiveDone.WaitOne();

        // Write the response to the console.
        AlmacenarTiemposDeEspera(response);
        Console.WriteLine("Response received : \n{0}", response);

        // Release the socket.
        //s_ClienteTS.Shutdown(SocketShutdown.Both);
        //s_ClienteTS.Close();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.ToString());
    }
}
```

```
public void AlmacenarTiemposDeEspera(String res)
{
    String[] lineas = res.Split('\n');
    String[] datos;
    for(int i = 0; i < lineas.Length; i++)
    {
        datos = lineas[i].Split(';');
        for(int j = 0; j < numAtracciones; j++)
        {
            if (mapaData[j, 0].Equals(datos[0]))
            {
                mapaData[j, 2] = datos[1];
            }
        }
    }
}
```

Visto esto, solo nos queda ver los métodos referentes al funcionamiento del gestor de colas. En primer lugar, tenemos el método *SolicitudAccesoKafka()*, que será el método que se ejecute concurrentemente en el hilo paralelo creado en el método *main*. Este método, realizará una ejecución diferente en función de que mensaje reciba del gestor de colas de parte del módulo del visitante, pudiendo así enviar el número de visitantes actuales y el total del parque, recibiendo un aviso de que el usuario sale del parque o enviando el mapa del mismo.

```

public void SolicitudAccesoKafka()
{
    using (var consumer = new ConsumerBuilder<Null, string>(cconfig).Build())
    {
        consumer.Subscribe("visitantes");
        try
        {
            while (true)
            {
                Console.WriteLine("Visitantes actuales: " + visitantesActuales);
                var consumeResult = consumer.Consume();
                switch(consumeResult.Message.Value){
                    case "Acceso":
                        this.EnviaAforoKafka();
                        if(visitantesActuales < maxVisitantes)
                        {
                            visitantesActuales++;
                            Console.WriteLine("Visitantes actuales: " + visitantesActuales);
                        }
                        break;
                    case "Salgo":
                        visitantesActuales--;
                        Console.WriteLine("Visitantes actuales: " + visitantesActuales);
                        break;
                    case "Mapa":
                        EnviarMapaKafka();
                        break;
                    default:
                        break;
                }
                Console.WriteLine("Enviados visitantes actuales:");
                Console.WriteLine(consumeResult.Message.Value);
            }
        }
        catch (Exception)
        {
        }
    }
}

```

El método *EnviarMapaKafka()* envía el *String* generado por los métodos que emplea.

```

public void EnviarMapaKafka()
{
    String[,] mapa = ConstruirMapa();
    String mapaString = ConstruyeStringMapa(mapa);
    using (var producer = new ProducerBuilder<Null, string>(pconfig).Build())
    {
        var dr = producer.ProduceAsync("sd-events", new Message<Null, string> { Value = mapaString }).Result;
        Console.WriteLine($"Delivered \n'{dr.Value}' to: {dr.TopicPartitionOffset}");
    }
}

```

No hay mucho que comentar a cerca de los métodos de construcción del mapa, más que el mapa está representado por el carácter '.' cuando el elemento de la casilla está vacío, un número cuando hay una atracción (el tiempo de espera) y está delimitado por caracteres '#'.

El método *EnviarAforoKafka()*, tiene una estructura idéntica a *EnviarMapaKafka()* salvo por que construye una cadena con el siguiente formato para enviar la información que requiere el visitor:

visitantesActuales: maxvisitantes:

2.1.2. FWQ_Registry.

El módulo de registro también cuenta con su *Program.cs* correspondiente con su método *main*. Este método solo recibe el puerto de escucha que va a emplear e inicializa el componente.

```
class Program
{
    //referencias
    static void Main(string[] args)
    {
        string puertoEscucha;

        if (args.Length == 2)
        {
            puertoEscucha = args[1];
            Registry r = new Registry(puertoEscucha);
            r.Start();
        } else
        {
            Console.WriteLine("Introducir puerto de escucha");
        }
    }
}
```

La clase *Registry* es la que se encargará de realizar las conexiones con el visitante para permitirle realizar las acciones pertinentes de usuario. Aquí los atributos y el constructor.

```
IPHostEntry host;
IPAddress ipAddr;
IPEndPoint endPoint;

Socket s_Servidor;
Socket s_Cliente;
static int maximoPeticiones = 10;
public static ManualResetEvent allDone = new ManualResetEvent(false);
static String path = Path.GetFullPath("..\..\..\..\usuarios.txt");
//referencia
public Registry(String puertoEscucha)
{
    host = Dns.GetHostEntry("localhost");
    ipAddr = host.AddressList[0];
    int puerto = Int32.Parse(puertoEscucha);
    endPoint = new IPEndPoint(ipAddr, puerto);

    //(para escuchar desde esa address familia, tipo de socket q usamos, protocolo por el q envia y recibe info )
    s_Servidor = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);

    //desde donde va a escuchar
    s_Servidor.Bind(endPoint);

    //nº max de conexiones q va a tener en cola antes de rechazar
    s_Servidor.Listen(maximoPeticiones);

    Console.WriteLine("Escuchando al puerto " + puertoEscucha);
}
```

En este caso, los métodos *Start()* y *ReadCallback(IAsyncResult ar)* son los más destacables a la hora de crear el servidor por sockets. El primero porque crea el servidor concurrente con ayuda del resto de métodos correspondientes y, el segundo, por ser el que lee el mensaje recibido por parte del visitante para saber que funcionalidad desea emplear el mismo.

```
public void Start()
{
    while (true)
    {
        allDone.Reset();
        Console.WriteLine("Esperando conexión...");
        s_Servidor.BeginAccept(new AsyncCallback(AcceptCallback), s_Servidor);
        allDone.WaitOne();
    }
}
```

```
public static void ReadCallback(IAsyncResult ar)
{
    String content = String.Empty;

    // Retrieve the state object and the handler socket
    // from the asynchronous state object.
    StateObject state = (StateObject)ar.AsyncState;
    Socket handler = state.workSocket;

    // Read data from the client socket.
    int bytesRead = handler.EndReceive(ar);

    if (bytesRead > 0)
    {
        // There might be more data, so store the data received so far.
        state.sb.Append(Encoding.ASCII.GetString(
            state.buffer, 0, bytesRead));

        // Check for end-of-file tag. If it is not there, read
        // more data.
        content = state.sb.ToString();
        String[] c = content.Split(';');
        String mensaje = SelectOperation(c);

        Console.WriteLine("Enviando resultados...");
        // Echo the data back to the client.
        Send(handler, mensaje);
    }
}
```

Esta decisión la tomará el método *SelectOperation(String[] caso)*.

```
public static String SelectOperation(String[] caso)
{
    String mensaje = String.Empty;
    switch (caso[0])
    {
        case "Crear perfil":
            mensaje = CrearPerfil(caso);
            break;
        case "Editar perfil":
            mensaje = EditarPerfil(caso);
            break;
        case "Iniciar sesion":
            mensaje = IniciarSesion(caso);
            break;
        default:
            break;
    }
    return mensaje;
}
```


Este método llama a su vez a *CrearPerfil()*, *EditarPerfil* o *IniciarSesion()* en función de lo que lea del cliente.

```
public static String CrearPerfil(String[] datos)
{
    String mensaje = String.Empty, password = String.Empty;
    int lineToEdit = 0;

    (lineToEdit, password) = BuscarUsuario(datos[1], path);

    StreamWriter sw = File.AppendText(path);

    if (lineToEdit != -1)
    {
        mensaje = "El usuario ya existe!";
    } else
    {
        sw.WriteLine(datos[1] + ";" + datos[2] + ";" + datos[3] + ";");
        mensaje = "Usuario creado con éxito.";
    }
    sw.Close();
    return mensaje;
}
```

```
public static String IniciarSesion(String[] datos)
{
    String mensaje = String.Empty, password = String.Empty;
    int lineToEdit = 0;

    (lineToEdit, password) = BuscarUsuario(datos[1], path);
    if (lineToEdit != -1)
    {
        if (password.Equals(datos[2]))
        {
            mensaje = "Credenciales correctas.";
        } else
        {
            mensaje = "Password incorrecta.";
        }
    } else
    {
        mensaje = "El usuario no existe!";
    }
    return mensaje;
}
```

```
public static String EditarPerfil(String[] datos)
{
    String mensaje = String.Empty, password = String.Empty;
    int lineToEdit = 0;

    (lineToEdit, password) = BuscarUsuario(datos[1], path);
    if (lineToEdit != -1)
    {
        if (password.Equals(datos[2]))
        {
            lineChanger(datos[3] + ";" + datos[4] + ";" + datos[5] + ";", path, lineToEdit);
            mensaje = "Cambios realizados con éxito.";
        } else
        {
            mensaje = "Password incorrecta.";
        }
    } else
    {
        mensaje = "El usuario no existe!";
    }
    return mensaje;
}
```

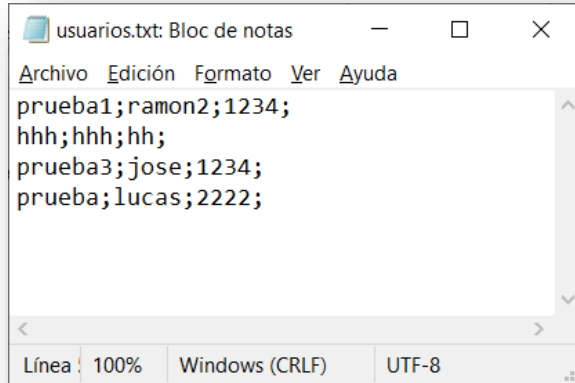
Estos, a su vez, emplean si es necesario los métodos auxiliares *BuscarUsuario(String alias, String path)* y *lineChanger(String newText, int p, int lte)* para desarrollar sus funcionalidades.

```
public static (int, String) BuscarUsuario(String alias, String path)
{
    String password = String.Empty;
    int lineToEdit = -1;
    StreamReader sr = File.OpenText(path);
    String[] splitter;
    String line;
    bool existe = false;
    for (int i = 0; (line = sr.ReadLine()) != null && !existe; i++)
    {
        splitter = line.Split(';');
        if (splitter[0].Equals(alias))
        {
            existe = true;
            lineToEdit = i;
            password = splitter[2];
        }
    }
    sr.Close();
    return (lineToEdit, password);
}
```

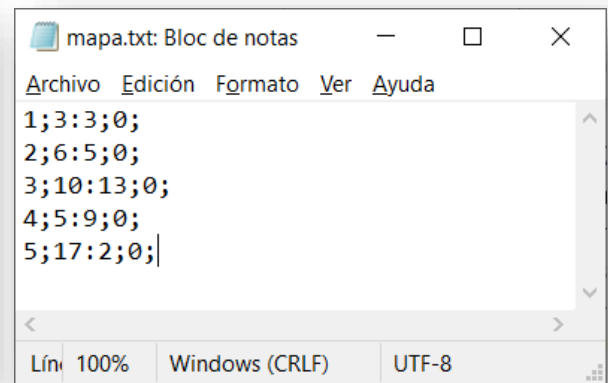
```
static void lineChanger(string newText, string p, int lte)
{
    string[] arrLine = File.ReadAllLines(p);
    arrLine[lte] = newText;
    File.WriteAllLines(p, arrLine);
}
```

2.1.3. Base de datos.

El lugar donde se almacena la información referente a los usuarios y al mapa se ha decidido que sean dos ficheros de texto simples, que guardan estos datos con un formato concreto para su posterior lectura en el código.



```
usuarios.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
prueba1;ramon2;1234;
hhh;hhh;hh;
prueba3;jose;1234;
prueba;lucas;2222;
Línea 100% Windows (CRLF) UTF-8
```



```
mapa.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
1;3;3;0;
2;6;5;0;
3;10;13;0;
4;5;9;0;
5;17;2;0;
Línea 100% Windows (CRLF) UTF-8
```

Los formatos son:

Alias;nombre;contraseña;\n para el fichero de usuarios.

IDAtraccion;posX:posY;tiempoDeCiclo; para el fichero del mapa¹.

¹: El valor final se emplea para facilitar la lectura por parte del programa. El código no almacena este dato ya que se actualiza de forma concurrente en un breve periodo de tiempo y se trataría de un proceso con un consumo de memoria innecesario para la poca trascendencia que tiene almacenar el dato.

2.2. Servidor de Tiempos de Espera.

En este módulo se alojará todo lo necesario para la creación de un servidor concurrente que envíe (de momento) los tiempos de espera en cada atracción, leyendo también información del gestor de colas enviada por los sensores para esto.

2.2.3. FWQ_WaitingTimeServer.

El *main* en este caso, recoge la IP del gestor de colas, su puerto y un puerto de escucha para montar el servidor.

```
static void Main(string[] args)
{
    string ipBroker;
    string puertoBroker;
    string puertoEscucha;

    if (args.Length == 4)
    {
        puertoEscucha = args[1];
        ipBroker = args[2];
        puertoBroker = args[3];

        TimeServer s = new TimeServer(puertoEscucha, ipBroker, puertoBroker);
        Thread th1 = new Thread(s.Start);
        th1.Start();
        s.StartConsumingKafka();
    }
    else
    {
        Console.WriteLine("Los parámetros introducidos no son suficientes");
    }
}
```

A continuación, se detallan los atributos de clase y el constructor.

```
class TimeServer
{
    IPEndPoint endPoint;
    IPAddress ipAddr;
    IPEndPoint endPoint;

    Socket s_Servidor;
    Socket s_Cliente;
    static int maximoPeticiones = 10;
    public static ManualResetEvent allDone = new ManualResetEvent(false);
    static ConsumerConfig config;
    static int[] visitantesPorAtraccion = new int[5];

    public TimeServer(String puertoEscucha, String ipBroker, String puertoBroker)
    {
        host = Dns.GetHostEntry("localhost");
        ipAddr = host.AddressList[0];
        int puerto = Int32.Parse(puertoEscucha);
        endPoint = new IPEndPoint(ipAddr, puerto);

        //(para escuchar desde esa address familia, tipo de socket q usamos, protocolo por el q envia y recibe info )
        s_Servidor = new Socket(ipAddr.AddressFamily, SocketType.Stream, ProtocolType.Tcp);

        //desde donde va a escuchar
        s_Servidor.Bind(endPoint);

        //nº max de conexiones q va a tener en cola antes de rechazar
        s_Servidor.Listen(maximoPeticiones);

        config = new ConsumerConfig
        {
            BootstrapServers = ipBroker + ":" + puertoBroker,
            SecurityProtocol = SecurityProtocol.Plaintext,
            GroupId = "my-group"
        };

        for(int i = 0; i < 5; i++)
        {
            visitantesPorAtraccion[i] = 5;
        }

        Console.WriteLine("Escuchando al puerto " + puertoEscucha);
    }
}
```

El método *StartConsumingKafka()* que se ejecuta en el hilo principal, lee los datos que envían los sensores en formato *idAtraccion:numVisitantes* y los almacena.

```
public void StartConsumingKafka()
{
    using (var consumer = new ConsumerBuilder<Null, string>(config).Build())
    {
        consumer.Subscribe("sensores");
        try
        {
            while (true)
            {
                var consumeResult = consumer.Consume();
                String[] recibido = consumeResult.Message.Value.Split(":");
                int[] parseo = new int[2];
                parseo[0] = Int32.Parse(recibido[0]);
                parseo[1] = Int32.Parse(recibido[1]);
                visitantesPorAtraccion[parseo[0]-1] = parseo[1];
                Console.WriteLine(consumeResult.Message.Value);
            }
        }
        catch (Exception)
        {
            consumer.Close();
        }
    }
}
```

La variable en la que se almacena, es empleada en el método *Calculo()*, referenciado en *ReadCallback(IAsyncResult ar)* para calcular el tiempo de espera de la atracción y la preparación del *String* para enviarlo al *Engine* mediante el socket creado.

```
public static String Calculo()
{
    StringBuilder sb = new StringBuilder();
    StreamReader sr = File.OpenText("atracciones.txt");
    String[] spliter;
    //String res = "";
    String line;
    int ciclo, visitantesCiclo, resultado = 0;
    for(int i = 0; (line = sr.ReadLine()) != null; i++)
    {
        spliter = line.Split(';');
        ciclo = Int32.Parse(spliter[1]);
        visitantesCiclo = Int32.Parse(spliter[2]);
        resultado = (visitantesPorAtraccion[i] / visitantesCiclo) * ciclo;
        sb.Append(spliter[0] + ";" + resultado + "\n");
    }
    sr.Close();
    return sb.ToString();
}
```

2.3. Visitantes.

Este módulo es el que contiene la lógica de los usuarios que quieren acceder al parque. Está implementado mediante un proyecto de Windows Forms el cuál se verá más adelante.

2.3.1. FWQ_Visitor.

El método pensado para poder crear el objeto *Visitor* en el código de los formularios que lo requieran ha sido el pasar por parámetros a los constructores de estos los datos que recoge la consola como parámetros, siendo estos la IP y puerto del *Broker* y la IP y puerto del *Registry*. Cada *Form* enviará al siguiente estos datos por parámetro en el constructor, de forma que puedan inicializar un objeto *Visitor* que les permita la conexión por Kafka y sockets. Los botones de los formularios únicamente preparan la información para crear el visitante e imprimir por un *label* las respuestas recibidas.

```
static class Program
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        string ipBroker;
        string puertoBroker;
        string ipRegistry;
        string puertoRegistry;
        if(args.Length == 5)
        {
            ipBroker = args[1];
            puertoBroker = args[2];
            ipRegistry = args[3];
            puertoRegistry = args[4];

            Application.SetHighDpiMode(HighDpiMode.SystemAware);
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Inicio(ipBroker, puertoBroker, ipRegistry, puertoRegistry));
        } else
        {
            Console.WriteLine("Argumentos insuficientes");
        }
    }
}
```

Estos son los atributos y el constructor de la clase *Visitor*, similar al resto.

```
class Visitor
{
    private static ManualResetEvent connectDone = new ManualResetEvent(false);
    private static ManualResetEvent sendDone = new ManualResetEvent(false);
    private static ManualResetEvent receiveDone = new ManualResetEvent(false);

    private static String response = String.Empty;
    private static String llamador;
    private static IPHostEntry hostRegistry;
    private static IPAddress ipAddrRegistry;
    private static IPEndPoint endPointRegistry;
    private static String[] mensaje;
    private static Socket s_clienteR;
    private static ProducerConfig pconfig;
    private static ConsumerConfig cconfig;

    public Visitor(String ipBroker, String puertoBroker, String ipRegistry, String puertoRegistry, String ll, String[] m)
    {
        mensaje = m;
        llamador = ll;
        hostRegistry = Dns.GetHostEntry(ipRegistry);
        ipAddrRegistry = hostRegistry.AddressList[0];
        int puerto = Int32.Parse(puertoRegistry);
        endPointRegistry = new IPEndPoint(ipAddrRegistry, puerto);

        // (para escuchar desde esa address familia, tipo de socket q usamos, protocolo por el q envia y recibe info )
        s_clienteR = new Socket(ipAddrRegistry.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
        // s_clienteBroker = new Socket(ipAddrBroker.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
        pconfig = new ProducerConfig
        {
            BootstrapServers = ipBroker + ":" + puertoBroker,
            SecurityProtocol = SecurityProtocol.Plaintext
        };

        cconfig = new ConsumerConfig
        {
            BootstrapServers = ipBroker + ":" + puertoBroker,
            SecurityProtocol = SecurityProtocol.Plaintext,
            GroupId = "my-group3"
        };
    }
}
```

En este módulo, contamos con tres métodos que envían información al gestor de colas en función de cuál sea la opción que escoja el visitante dentro de la aplicación.

```
public void SolicitudAccesoKafka()
{
    using (var producer = new ProducerBuilder<Null, string>(pconfig).Build())
    {
        var dr = producer.ProduceAsync("visitantes", new Message<Null, string> { Value = "Acceso" }).Result;
        Console.WriteLine($"Delivered '{dr.Value}' to: {dr.TopicPartitionOffset}");
    }
}

1referencia
public void SalirParqueKafka()
{
    using (var producer = new ProducerBuilder<Null, string>(pconfig).Build())
    {
        var dr = producer.ProduceAsync("visitantes", new Message<Null, string> { Value = "Salgo" }).Result;
        Console.WriteLine($"Delivered '{dr.Value}' to: {dr.TopicPartitionOffset}");
    }
}

2referencias
public void SolicitarMapaKafka()
{
    using (var producer = new ProducerBuilder<Null, string>(pconfig).Build())
    {
        //while (true)
        //{
            var dr = producer.ProduceAsync("visitantes", new Message<Null, string> { Value = "Mapa" }).Result;
            Console.WriteLine($"Delivered '{dr.Value}' to: {dr.TopicPartitionOffset}");
            //RecibirMapaKafka();
            Thread.Sleep(1 * 1000);
        //}
    }
}
```

Un método que crea un mensaje u otro en función de lo mismo.

```
public static String CreaMensajeLlamada(String ll, String[] m)
{
    StringBuilder mensaje = new StringBuilder();
    switch (ll)
    {
        case "Crear perfil":
            mensaje.Append(ll + ";" + m[0] + ";" + m[1] + ";" + m[2] + ";");
            break;
        case "Editar perfil":
            mensaje.Append(ll + ";" + m[0] + ";" + m[1] + ";" + m[2] + ";" + m[3] + ";" + m[4] + ";");
            break;
        case "Iniciar sesion":
            mensaje.Append(ll + ";" + m[0] + ";" + m[1] + ";");
            break;
        default:
            break;
    }
    return mensaje.ToString();
}
```

Y dos métodos que reciben información del Kafka, uno que recibe el aforo y otro el mapa, junto con el resto de los métodos necesarios para los sockets.

```

public bool RecibirAforoKafka()
{
    bool resultado = false;
    using (var consumer = new ConsumerBuilder<Null, string>(cconfig).Build())
    {
        consumer.Subscribe("visitantes2");
        try
        {
            var consumeResult = consumer.Consume();
            String[] recibido = consumeResult.Message.Value.Split(':');
            if (Int32.Parse(recibido[0]) == Int32.Parse(recibido[1]))
            {
                resultado = false;
            }
            else
            {
                resultado = true;
            }
        }
        catch (Exception)
        {
            consumer.Close();
        }
    }
    return resultado;
}

```

```

public String RecibirMapaKafka()
{
    String mapa = String.Empty;
    using (var consumer = new ConsumerBuilder<Null, string>(cconfig).Build())
    {
        consumer.Subscribe("sd-events");
        try
        {
            //no consume correctamente
            var consumeResult = consumer.Consume();
            mapa = consumeResult.Message.Value;
            return mapa;
        }
        catch (Exception)
        {
            consumer.Close();
        }
    }
    return mapa;
}

```

2.4. Sensores.

En este módulo, se implementan los sensores, colocados en cada atracción, que envían el número de visitantes actuales que hay en cada uno.

2.4.1. FWQ_Sensor.

El módulo del sensor cuenta simplemente con un método *main* en la clase *Program* que realiza todas las funcionalidades del sensor. Para facilitar la implementación, cualquier sensor que se active detecta que hay un total de diez visitantes en la atracción que reciba como parámetro, junto con la IP y el puerto del Broker.

```
static void Main(string[] args)
{
    string ipBroker;
    string puertoBroker;
    string idAtraccion;

    if (args.Length == 4)
    {
        ipBroker = args[1];
        puertoBroker = args[2];
        idAtraccion = args[3];

        var config = new ProducerConfig
        {
            BootstrapServers = ipBroker + ":" + puertoBroker,
            SecurityProtocol = SecurityProtocol.Plaintext,
        };

        using (var producer = new ProducerBuilder<Null, String>(config).Build());

        var rand = new Random();
        int a = Int32.Parse(idAtraccion);
        int nVisitantes = 10;
        int counter = 0;
        while (true)
        {
            if(counter % 10 == 0 && nVisitantes > 0)
            {
                nVisitantes--;
            }
            String enviar = idAtraccion + ":" + nVisitantes + ":";
            using (var producer = new ProducerBuilder<Null, string>(config).Build())
            {
                var dr = producer.ProduceAsync("sensores", new Message<Null, string> { Value = enviar }).Result;
                Console.WriteLine($"Delivered '{dr.Value}' to: {dr.TopicPartitionOffset}");
            }
            rand = new Random();
            Thread.Sleep(rand.Next(1,4) * 1000);
            counter++;
        }
    }
    else
    {
        Console.WriteLine("Los parámetros introducidos deben ser 4.");
    }
}
```


3. Guía de despliegue.

Para desplegar el programa correctamente, ejecutamos los componentes en el siguiente orden:

1. Gestor de colas mediante Kafka en el PC1. Esto se realiza mediante las siguientes instrucciones en dos terminales diferentes, además de la creación de los *topics* en una nueva.

Primero: *./bin/zookeeper-server-start.sh ./config/zookeeper.properties.*

Segundo: *./bin/kafka-server-start.sh ./config/server.properties.*

Creación de *topics*: *./bin/kafka-topics.sh --create --topic nombre-topic --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1.*

2. Servidor de Tiempos de Espera y Registro. Mediante la ejecución en la *cmd* de Windows en las respectivas carpetas. El módulo de Registro debe estar en la misma máquina que el *Engine*, pero lo ejecutaremos después. El WTS puede estar en una nueva máquina o en el mismo PC que el gestor de colas.

En las carpetas *\bin\Debug\net5.0* del proyecto:

FWQ_WaitingTimeServer.exe FWQ_WaitingTimeServer puertoEscucha ipBroker puertoBroker

FWQ_Registry.exe FWQ_Registry puertoEscucha

3. *Engine*. Mediante la ejecución en la *cmd* de Windows de la siguiente instrucción.

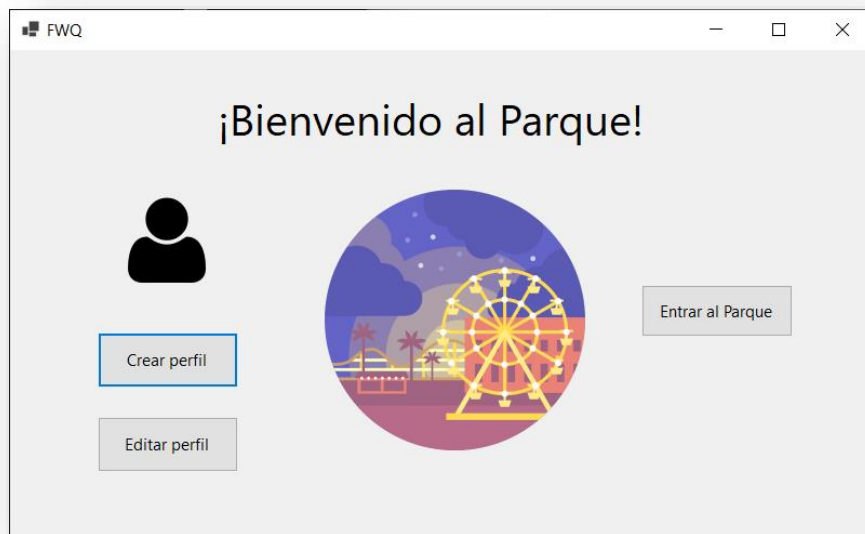
FWQ_Engine.exe FWQ_Engine ipBroker puertoBroker maxVisitantes ipTS puertoTS

4. Visitante y Sensores en PC3. Dado que los sensores no interfieren de forma directa con los visitantes, pueden establecerse en la misma máquina. Nuevamente, ejecutamos los comandos pertinentes.

FWQ_Visitor.exe FWQ_Visitor ipBroker puertoBroker ipRegistry puertoRegistry

FWQ_Sensor.exe FWQ_Sensor ipBroker puertoBroker idAtraccion

Arrancamos el visitante:



Creamos un usuario y se agrega a la BBDD:

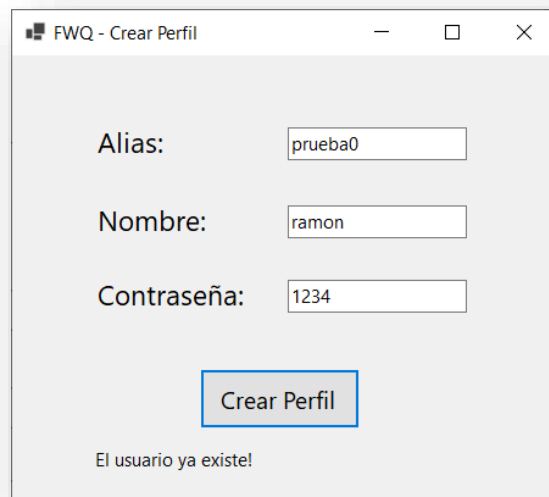
The screenshot shows a form titled 'FWQ - Crear Perfil'. It has three input fields: 'Alias' with the value 'prueba0', 'Nombre' with the value 'visitante', and 'Contraseña' with the value 'sd'. Below these fields is a button labeled 'Crear Perfil'. At the bottom of the form, a message reads 'Usuario creado con éxito.'The screenshot shows a Windows command prompt window with the title 'C:\Windows\System32\cmd.exe - FWQ_Registry.exe'. The output text is as follows:

```
Microsoft Windows [Versión 10.0.19042.1288]  
(c) Microsoft Corporation. Todos los derechos reservados.  
C:\Users\rjuar\source\repos\FWQ\FWQ\FWQ_Registry.exe  
Escuchando al puerto 9010  
Esperando conexión...  
Esperando conexión...  
Enviando resultados...  
Sent 25 bytes to client.
```

The screenshot shows a Notepad window titled 'usuarios.txt: Bloc de notas'. The text content is as follows:

```
Archivo Edición Formato Ver Ayuda  
prueba1;ramon2;1234;  
hhh;hhh;hh;  
prueba3;jose;1234;  
prueba;lucas;2222;  
prueba0;visitante;sd;
```

No podemos crear un usuario ya existente:



FWQ - Crear Perfil

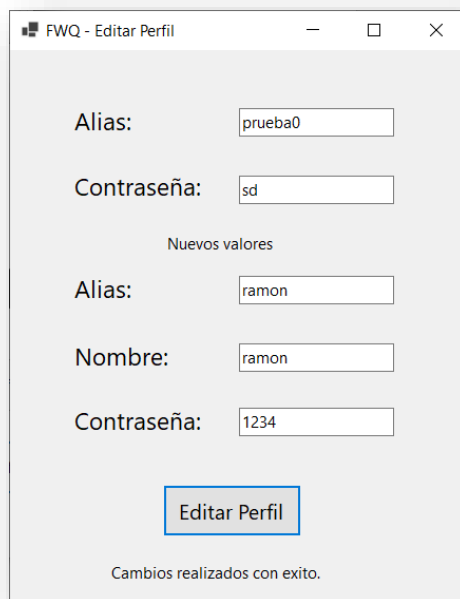
Alias:

Nombre:

Contraseña:

El usuario ya existe!

Modificamos el usuario recién creado:



FWQ - Editar Perfil

Alias:

Contraseña:

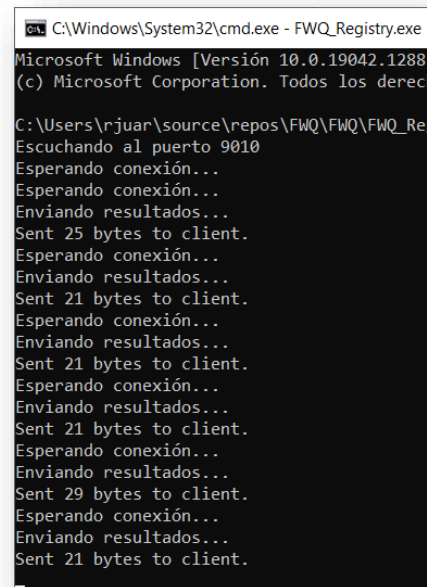
Nuevos valores

Alias:

Nombre:

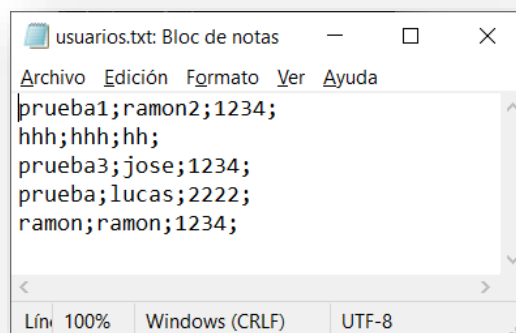
Contraseña:

Cambios realizados con éxito.



```
C:\Windows\System32\cmd.exe - FWQ_Registry.exe
Microsoft Windows [Versión 10.0.19042.1288]
(c) Microsoft Corporation. Todos los derechos reservados

C:\Users\rjruar\source\repos\FWQ\FWQ\FWQ_Regi...
Escuchando al puerto 9010
Esperando conexión...
Esperando conexión...
Enviando resultados...
Sent 25 bytes to client.
Esperando conexión...
Enviando resultados...
Sent 21 bytes to client.
Esperando conexión...
Enviando resultados...
Sent 21 bytes to client.
Esperando conexión...
Enviando resultados...
Sent 21 bytes to client.
Esperando conexión...
Enviando resultados...
Sent 21 bytes to client.
Esperando conexión...
Enviando resultados...
Sent 29 bytes to client.
Esperando conexión...
Enviando resultados...
Sent 21 bytes to client.
```



usuarios.txt: Bloc de notas

Archivo Edición Formato Ver Ayuda

prueba1;ramon2;1234;
hhh;hhh;hh;
prueba3;jose;1234;
prueba;lucas;2222;
ramon;ramon;1234;

Línea 100% Windows (CRLF) UTF-8

No podemos modificar un usuario que no existe:

FWQ - Editar Perfil

Alias:

Contraseña:

Nuevos valores

Alias:

Nombre:

Contraseña:

Editar Perfil

El usuario no existe!

Iniciamos sesión con credenciales inválidas:

FWQ - Iniciar Sesión

Alias

Contraseña

Entrar

El usuario no existe!

FWQ - Iniciar Sesión


Alias

Contraseña

Entrar

Password incorrecta.

Entramos al parque:



FWQ - Iniciar Sesión (No responde)

Alias

Contraseña

Credenciales correctas.

[illegible]

NOTA: El mensaje de No responde se debe a que el propio visitante no lee bien el mapa que le envía el *Engine*. Se acepta el inicio de sesión, pero falla al leer el mapa. El error no es de código ya que la implementación está realizada de igual forma en cualquier otro elemento que emplee Kafka en el sistema. Si la lectura fuera correcta, se lanzaría otro formulario con el mapa que ha recibido el visitante.