Documentación

Pre-entrega - Obligatorio 2

Programación de Redes

Francisco Cabanillas (231918)
Matias Wilner (231073)

Licenciatura en Sistemas - Grupo M6A

Universidad ORT Uruguay

2021

Descripción general del trabajo

En la entrega anterior se construyó una aplicación cliente y una aplicación servidor, en un proyecto llamado “GameStore”, donde se simulaba a nivel general un sistema de adquisición, publicación, listado y calificación de juegos, además de aspectos relacionados al manejo de archivos.
La aplicación cliente permite realizar todo lo relativo a la interacción con el usuario que usa el sistema, mientras que la aplicación servidor permite procesar todas las solicitudes de los usuarios en torno al uso del sistema, como puede ser listar los juegos, comprar un juego, publicar un juego, entre las demás expresadas anteriormente.
El sistema permite el manejo de varias conexiones de distintos usuarios al mismo tiempo, con la posibilidad de procesar pedidos de varios clientes a la vez.

El objetivo de esta entrega fue poner en práctica los conocimientos adquiridos en la segunda parte del curso, el manejo del patrón asincrónico y el uso de la librería TCP de alto nivel TcpListener/TcpClient.

En primer lugar se sustituyó todo el uso de Threads por el modelo asincrónico de .NET, con el uso de Task y modelando en base a async y await. En segundo lugar se cambió todo lo relativo a Sockets por el uso de la librería de TcpListener/TcpClient.

Cambios realizados a la primera entrega

Uso del modelo asincrónico

Tal como fue mencionado previamente, para esta entrega se hizo uso del modelo asincrónico de .NET. El modelo asincrónico permite gestionar cómo será ejecutado el código desarrollado en una aplicación. La palabra reservada async indica que es un método que se quiere sincronizar con métodos que se ejecutarán de forma asincrónica, mientras que la palabra await indica que la ejecución de las sentencias de código escritas deben esperar por lo que prosiga a await. A continuación un ejemplo del código:

```
private async Task Rate(Header header)
{
    var bufferData = new byte[header.IDataLength];
    await ReceiveData(header.IDataLength, bufferData);

    var splittedReview string[] = (Encoding.UTF8.GetString(bufferData)).Split( separator: "*");
    var gameId = Convert.ToInt32(splittedReview[0]);
    var reviewedGame = _gamesLogic.GetById(gameId);
    var newReview = new Review(_userLogged, reviewedGame, rating: Convert.ToInt32(splittedReview[1]),
        splittedReview[2]);
    _reviewLogic.Add(newReview);
    _reviewLogic.AdjustRating(gameId);
    var response = $"{_userLogged.UserName} successfully reviewed {reviewedGame.Title}";
```

En el ejemplo de la imagen, se usa la palabra await previo al llamado del método ReceiveData. Esto quiere decir que hasta que no se termine de ejecutar dicho método, la lógica implementada a continuación no será ejecutada. Esto es básicamente porque se necesita la totalidad de la información que se obtiene a partir del método ReceiveData para continuar con la lógica de la aplicación detallada a continuación. Es pertinente aclarar que la palabra await no puede ser usada si en la firma del método donde está contenida dicha sentencia no se encuentra la palabra async.

Como se puede ver en la firma del método, se utiliza la palabra reservada Task. Una Task crea una tarea asincrónica a nivel de sistema en C#. Tanto Thread como Task se utilizan para la programación paralela, pero Task necesita de menos recursos para ser utilizada, lo que hace a la aplicación tener una mejor performance entre otras cosas.

Otro ejemplo que se puede visualizar de la clase Task es el siguiente:

```
private async Task<byte[]> Response(int command)
{
    var bufferResponse = new byte[] { };
    var headerLength = HeaderConstants.Response.Length + HeaderConstants.CommandLength +
                       HeaderConstants.DataLength;
    var buffer = new byte[headerLength];
    try
    {
        await ReceiveData(headerLength, buffer);
        var header = new Header();
```

Cuando luego de la palabra Task hay una palabra entre símbolos de mayor y menor (<>), significa que la función debe retornar un objeto de ese tipo. En el caso del ejemplo de la imagen, la función Response retorna un array de bytes.

```
public async Task ListenConnectionsAsync(TcpListener tcpListener, IServiceProvider serviceProvider)
{
    while (!_exit)
    {
        try
        {
            var tcpClientSocket = await tcpListener.AcceptTcpClientAsync().ConfigureAwait(false);
            var task = Task.Run(async () => await StartRuntime(serviceProvider, tcpClientSocket).ConfigureAwait(false));

        }
        catch (Exception e)
        {
            Console.WriteLine(e);
            _exit = true;
        }
    }

    Console.WriteLine("Exiting....");
}
```

El método Run de la clase Task permite correr el método que se especifica dentro, en este caso se llama a toda la lógica del Server.

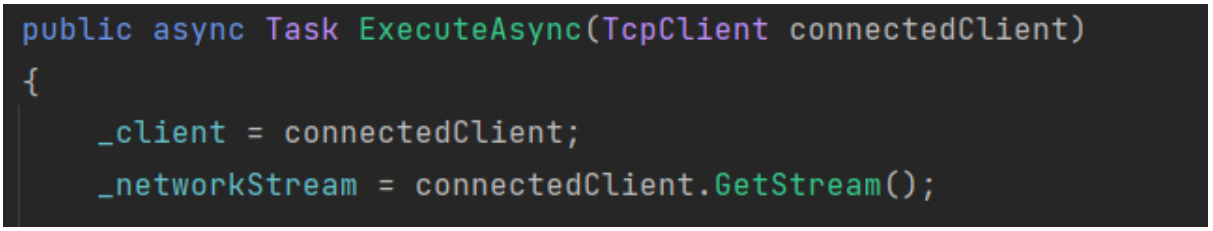
Uso de TCPListener y TCPClient

Se sustituyo el uso de Sockets por el uso de la librería TCP de alto nivel TcpClient/TcpListener. La clase TcpListener proporciona métodos simples que escuchan y aceptan solicitudes de conexión entrantes y se setea un ipEndPoint para crearla. Para conectarse con ella, se usa TcpClient.

El método Start sirve para empezar a escuchar las solicitudes de conexión entrantes. El método Stop se usa para cerrar el TcpListener.

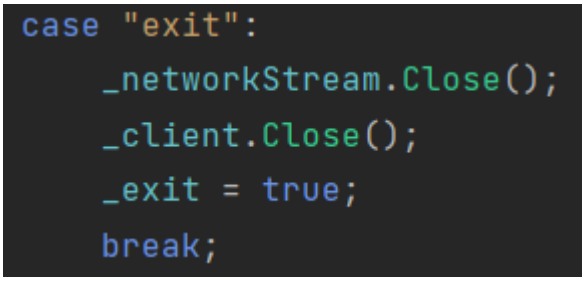
```
public void InitializeSocketServer(IServiceProvider serviceProvider)
{
    var ipEndPoint = new IPEndPoint(
        IPAddress.Parse(IpConfig),
        Port);
    var tcpListener = new TcpListener(ipEndPoint);
    tcpListener.Start(backlog: 100);
    var connections = new Connections();

    var task = Task.Run(async () => await connections.ListenConnectionsAsync(tcpListener, serviceProvider)).ConfigureAwai

    Console.WriteLine($"IpConfig: {IpConfig} - Port: {Port}");
    connections.HandleServer();
}
```

La clase TcpClient tiene métodos para conectarse, enviar y recibir datos a través de una red. Es necesario que exista un TcpListener escuchando solicitudes de conexiones entrantes para que se pueda dar la conexión. El método GetStream es usado para enviar y recibir datos a través de la clase NetworkStream.



```
public async Task ExecuteAsync(TcpClient connectedClient)
{
    _client = connectedClient;
    _networkStream = connectedClient.GetStream();
```

El método Close() de NetworkStream se usa para cerrar la secuencia actual y liberar todos los recursos relacionados a la misma.
Luego el método Close de TcpClient permite eliminar a dicho cliente y cerrar la conexión TCP.



```
case "exit":
    _networkStream.Close();
    _client.Close();
    _exit = true;
    break;
```

Otra de las cosas que se cambiaron para esta entrega fueron los métodos Request, Response y RecieveData. Estos 3 se hicieron asincrónicos, y se hizo uso de los métodos ReadAsync() y WriteAsync() de la clase NetworkStream para la transferencia de datos.

```
14 usages  Francisco +1
private async Task RequestAsync(string message, int command)
{
    var header = new Header( direction: HeaderConstants.Request, command,  datalength: message.Length);
    var data :byte[] = header.GetRequest();
    var bytesMessage :byte[] = Encoding.UTF8.GetBytes(message);
    try
    {
        await _networkStream.WriteAsync(data,  offset: 0,  count: data.Length).ConfigureAwait(false);
        await _networkStream.WriteAsync(bytesMessage,  offset: 0,  count: bytesMessage.Length).ConfigureAwait(false);
    }
    catch (IOException)
    {
        throw new ServerDisconnected();
    }
}
```

```
6 usages  Matias Wilner +1
private async Task ReceiveDataAsync(int length, byte[] buffer)
{
    var iRecv = 0;
    while (iRecv < length)
    {
        var received :int = await _networkStream // NetworkStream
            .ReadAsync(buffer,  offset: iRecv,  count: length - iRecv) // Task<int>
            .ConfigureAwait(false); // ConfiguredTaskAwaitable<int>
        iRecv += received;
    }
}
```

```
19 usages  Francisco
private async Task<byte[]> ResponseAsync(int command)
{
    var bufferResponse = new byte[] { };
    var headerLength :int = HeaderConstants.Response.Length + HeaderConstants.CommandLength +
                        HeaderConstants.DataLength;
    var buffer = new byte[headerLength];
    try
    {
        await ReceiveDataAsync(headerLength, buffer);
        var header = new Header();
        header.DecodeData(buffer);
        if (header.ICommand.Equals(command))
        {
            bufferResponse = new byte[header.IDataLength];
            await ReceiveDataAsync(header.IDataLength, bufferResponse);
        }
        else
        {
            Console.WriteLine(header.ICommand + " " + command);
        }

    }
    catch (IOException)
    {
        throw new ServerDisconnected();
    }
    catch (Exception e)
    {
        Console.WriteLine($"---- -> Message {e.Message}..");
    }
    return bufferResponse;
}
```