Universidad ORT Uruguay

Obligatorio 2

Programación de Redes

Agustina Disiot 221025

Ivan Monjardin 239850

https://github.com/IvanMonjardin/ProgramacionRedesOB1

2021

Índice:

Cambios TCP a Socket

En la primera entrega se utilizó TCPListener/TCPClient por lo tanto para esta entrega se migró a Sockets.

Motivación: Se demostró que ambas clases cumplen con las mismas funcionalidades, por lo menos a nivel de nuestro obligatorio, y que la migración de una a otra se puede realizar sin mayores problemas. Tcp es una librería de mayor nivel que socket y por lo tanto Socket puede hacer todo lo que hace Tcp solo que a veces precisa más líneas de código/parámetros, por ejemplo al cerrar la conexión.

Server

Antes:

```
public class Server
{
    private readonly TcpListener tcpListener;
    public const int maxClientsInQ = 100;
    public bool acceptingConnections;
    public List<ClientHandler> clientHandlers;
    1 reference
    public Server(string serverIpAddress, string serverPort)
    {

        clientHandlers = new List<ClientHandler>();
        var ipEndPoint = new IPEndPoint(IPAddress.Parse(serverIpAddress), int.Parse(serverPort));
        tcpListener = new TcpListener(ipEndPoint);
        acceptingConnections = true;
    }
```

Despues:

```
4 references | Agustina, 28 minutes ago | 3 authors, 12 changes
public class Server
{
    private readonly Socket server;
    public const int maxClientsInQ = 100;
    public bool acceptingConnections;
    public List<ClientHandler> clientHandlers;
    1 reference | Agustina, 5 days ago | 3 authors, 4 changes
    public Server(string serverIpAddress, string serverPort)
    {

        clientHandlers = new List<ClientHandler>();
        var ipEndPoint = new IPEndPoint(IPAddress.Parse(serverIpAddress), int.Parse(serverPort));
        server = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream,
            ProtocolType.Tcp);
        server.Bind(ipEndPoint);
        acceptingConnections = true;
    }
```

ClientHandler

Antes:

```
public class ClientHandler
{
    private readonly TcpClient _acceptedTcpClient;
    private INetworkStreamHandler networkStreamHandler;
    private bool isClientConnected;

    1 reference
    public ClientHandler(TcpClient newAcceptedTcpClient)
    {
        _acceptedTcpClient = newAcceptedTcpClient;
    }
```

```
public void StopHandling()
{
    if (isClientConnected)
    {
        isClientConnected = false;
        _acceptedTcpClient.Close();
    }
}
```

Despues:

```
public class ClientHandler
{
    private readonly Socket acceptedSocketClient;
    private INetworkStreamHandler networkStreamHandler;
    private bool isClientConnected;

    1 reference | Agustina, 5 days ago | 1 author, 1 change
    public ClientHandler(Socket newAcceptedSocketClient)
    {
        acceptedSocketClient = newAcceptedSocketClient;
    }
```

```
1 reference | Agustina, 5 days ago | 2 authors, 3 changes
public void StopHandling()
{
    if (isClientConnected)
    {
        isClientConnected = false;
        acceptedSocketClient.Shutdown(SocketShutdown.Both);
        acceptedSocketClient.Close();
    }
}
```

ClientPresentation

Antes:

```
public ClientPresentation()
{
    tcpClient = ClientNetworkUtil.GetNewClientTcpEndpoint();
    serverIpEndPoint = ClientNetworkUtil.GetServerEndpoint();
    fileHandler = new FileHandler();
}
```

Despues:

```
public class ClientPresentation
{
    private NetworkStreamHandler networkStreamHandler;
    private readonly Socket socketClient;
    private readonly IPEndPoint serverIpEndPoint;
    private readonly FileHandler fileHandler;
```

```
public void StartConnection()
{
    try
    {
        socketClient.Connect(serverIpEndPoint);
    }
    catch (SocketException e)
    {
        Console.WriteLine("No se pudo conectar con el servidor");
        Environment.Exit(0);
    }
    networkStreamHandler = new NetworkStreamHandler(new NetworkStream(socketClient));
}
```

A estos cambios después se agregaron operaciones asincrónicas que se explicarán en la siguiente parte.

Cambios Thread a Tasks

El cambio de Thread a Tasks se realizó para utilizar el modelo asincrónico de .NET.
Este modelo facilita el manejo de código en paralelo. Permite crear funciones asíncronas, poder seguir ejecutando código y solo tener que esperar una vez que se precisa el retorno de la función o que la función termine, esto se señaliza utilizado la palabra await. Con las task, el código puede o no correr en paralelo dependiendo de lo que considere mejor .NET, no tenemos que manejar directamente nosotros los hilos, lo cual permite mayor optimización de los recursos.

Para que se pueden utilizar funciones asíncronas, las funciones que utilizaban dichas funciones también tenían que ser asincrónicas si hacían el await de los resultados. Esto se repite hasta llegar a los mains tanto del servidor como del cliente.

Archivos

Leer y escribir archivos puede llevar mucho tiempo considerando el tiempo que demora el programa en ejecutar código. Por eso, se utilizan funciones asíncronas para permitir que se ejecute código mientras se espera a que se terminen de utilizar los archivos. Más adelante se explica por qué se utilizó un await cuando se llaman a las funciones asincrónicas, y sin bien esto no aprovecha las ventajas del código en paralelo, si es necesario para nuestro protocolo.

Antes:

```
public bool FileExists(string path)
{
    return File.Exists(path);
}

3 references
public bool PathExists(string directory)
{
    return !File.Exists(directory) && Directory.Exists(directory);
}
```

Despues:

```
public async Task<bool> FileExists(string path)
{
    return await Task.Run(()=>File.Exists(path));
}

3 references | Agustina, 4 hours ago | 1 author, 2 changes
public async Task<bool> PathExists(string directory)
{
    return await Task.Run(()=>!File.Exists(directory) && Directory.Exists(directory));
}
```

No se encontraron operaciones de FileExistsAsync por eso se utilizaron directamente Tasks.

Antes:

```
public byte[] Read(string path, long offset, int length)
{
    var data = new byte[length];

    using (var fs = new FileStream(path, FileMode.Open))
    {
        fs.Position = offset;
        var bytesRead = 0;
        while (bytesRead < length)
        {
            var read = fs.Read(data, bytesRead, length - bytesRead);
            if (read == 0)
            {
                throw new Exception("Couldn't not read file");
            }
            bytesRead += read;
        }
    }

    return data;
}
```

```
public void Write(string fileName, byte[] data)
{
    if (File.Exists(fileName))
    {
        using (var fs = new FileStream(fileName, FileMode.Append))
        {
            fs.Write(data, 0, data.Length);
        }
    }
    else
    {
        using (var fs = new FileStream(fileName, FileMode.Create))
        {
            fs.Write(data, 0, data.Length);
        }
    }
}
```

Después:

```
public async Task<byte[]> Read(string path, long offset, int length)
{
    var data = new byte[length];

    using (var fs = new FileStream(path, FileMode.Open))
    {
        fs.Position = offset;
        var bytesRead = 0;
        while (bytesRead < length)
        {
            var read = await fs.ReadAsync(data, bytesRead, length - bytesRead);
            if (read == 0)
            {
                throw new Exception("Couldn't not read file");
            }
            bytesRead += read;
        }
    }

    return data;
}
```

```
public async Task Write(string fileName, byte[] data)
{
    if (File.Exists(fileName))
    {
        using (var fs = new FileStream(fileName, FileMode.Append))
        {
            await fs.WriteAsync(data, 0, data.Length);
        }
    }
    else
    {
        using (var fs = new FileStream(fileName, FileMode.Create))
        {
            await fs.WriteAsync(data, 0, data.Length);
        }
    }
}
```

Enviar y recibir datos de la red

Otro tipo de operaciones que pueden demorar aún más la ejecución del código es la espera de mandar y principalmente de recibir datos del cliente/servidor. Como no se puede controlar cuando responde el servidor/cliente, hay que asegurarse que se posible seguir corriendo código mientras tanto y que no se tranque el programa. Esto se logra mediante el async/await.

Nosotros ya teníamos una clase NetworkStreamHandler y se modificó para agregar las operaciones async. Al estar en Common la modificación se tuvo que realizar solo una vez tanto para el server como para el cliente

Antes:

```
public interface INetworkStreamHandler

{
    6 references
    void Write(byte[] data);
    3 references
    void WriteInt(int data);
    2 references
    void WriteFileSize(long data);
    6 references
    void WriteString(string data);
    4 references
    void WriteCommand(Command data);


    7 references
    byte[] Read(int length);
    5 references
    int ReadInt(int length);
    2 references
    long ReadFileSize();
    8 references
    string ReadString(int length);
    4 references
    Command ReadCommand();
}
```

Después:

```
56 references | Agustina, 3 hours ago | 2 authors, 5 changes
public interface INetworkStreamHandler

{
    6 references | Agustina, 4 hours ago | 1 author, 2 changes
    Task Write(byte[] data);
    3 references | Agustina, 4 hours ago | 1 author, 2 changes
    Task WriteInt(int data);
    2 references | Agustina, 4 hours ago | 2 authors, 2 changes
    Task WriteFileSize(long data);
    6 references | Agustina, 4 hours ago | 1 author, 2 changes
    Task WriteString(string data);
    4 references | Agustina, 4 hours ago | 1 author, 2 changes
    Task WriteCommand(Command data);


    7 references | Agustina, 3 hours ago | 1 author, 2 changes
    Task<byte[]> Read(int length);
    5 references | Agustina, 4 hours ago | 1 author, 2 changes
    Task<int> ReadInt(int length);
    2 references | Agustina, 3 hours ago | 2 authors, 2 changes
    Task<long> ReadFileSize();
    8 references | Agustina, 3 hours ago | 1 author, 2 changes
    Task<string> ReadString(int length);
    4 references | Agustina, 3 hours ago | 1 author, 2 changes
    Task<Command> ReadCommand();
}
```

Para el FileNetworkStreamHandler que se utiliza principalmente para las descarga y subida de carátulas de los juegos, la interfaz se definió con Tasks y en la implementación se utiliza FileStreamHandler que ya se mostró arriba la migración que se hizo.

Antes:

```
public interface IFileNetworkStreamHandler
{
    4 references
    string ReceiveFile(string folderPath, string fileName = "");
    4 references
    void SendFile(string fileName);
}
```

Después:

```
public interface IFileNetworkStreamHandler
{
    4 references | Agustina, 4 hours ago | 2 authors, 2 changes
    Task<string> ReceiveFile(string folderPath, string fileName = "");
    4 references | Agustina, 4 hours ago | 2 authors, 2 changes
    Task SendFile(string fileName);
}
```

Además, para la implementación de dichas operaciones se utilizarán los métodos ya existentes de Read y Write async de NetworkStream.

Antes:

```
public void Write(byte[] data)
{
    _networkStream.Write(data, 0, data.Length);
}
```

```
public byte[] Read(int length)
{
    int dataReceived = 0;
    var data = new byte[length];
    while (dataReceived < length)
    {
        var received = _networkStream.Read(data, dataReceived, length - dataReceived);
        if (received == 0)
        {
            throw new SocketException();
        }
        dataReceived += received;
    }

    return data;
}
```

Después:

```
6 references | Agustina, 4 hours ago | 1 author, 2 changes
public async Task Write(byte[] data)
{
    await _networkStream.WriteAsync(data, 0, data.Length);
}
```

```
7 references | Agustina, 4 hours ago | 1 author, 2 changes
public async Task<byte[]> Read(int length)
{
    int dataReceived = 0;
    var data = new byte[length];
    while (dataReceived < length)
    {
        var received = await _networkStream.ReadAsync(data, dataReceived, length - dataReceived);
        if (received == 0)
        {
            throw new SocketException();
        }
        dataReceived += received;
    }

    return data;
}
```

Funciones intensas de CPU

No hubo ninguna operación que precise de alta demanda de CPU. Si hubiera, se tendría que hacer asincrónica y utilizar Tasks. En dicho caso las funciones de más alto nivel que las utilicen también precisarán ser asíncronas. Esto último es lo que ya está pronto y permitirá agregar nuevas funciones asincrónicas sin mucha dificultad.

Conectarse/Desconectarse

Las operaciones de conectarse y desconectarse tanto del servidor como del cliente también pueden demorar y por lo tanto se migraron al modelo asincrónico aprovechando cuando existieran las funciones asíncronas ya implementadas.

Server

Antes:

```
1 reference | IvanMonjardin, 26 days ago | 3 authors, 6 changes
public void StartReceivingConnections()
{
    tcpListener.Start(maxClientsInQ);

    while (acceptingConnections)
    {
        TcpClient acceptedTcpClient = null;
        try
        {
            acceptedTcpClient = tcpListener.AcceptTcpClient();
        }
        catch (SocketException e)
        {
            Console.WriteLine("Server no longer accept request");
            acceptingConnections = false;
        }
        if (acceptingConnections)
        {
            Console.WriteLine("Accepted new client connection");
            ClientHandler newHandler = new ClientHandler(acceptedTcpClient);
            clientHandlers.Add(newHandler);
            Thread clientThread = new Thread(() => newHandler.StartHandling());
            clientThread.Start();
        }
    }
```

Despues:

```
1 reference | Agustina, 17 minutes ago | 3 authors, 9 changes
public async void StartReceivingConnections()
{
    server.Listen(maxClientsInQ);

    while (acceptingConnections)
    {
        try
        {
            Socket acceptedClient = await server.AcceptAsync().ConfigureAwait(false);
            var task = Task.Run(async()=> await handleAcceptedConection(acceptedClient));
        }
        catch (SocketException e)
        {
            Console.WriteLine("Server no longer accept request");
            acceptingConnections = false;
        }

    }


}
```

```
1 reference | Agustina, 27 minutes ago | 1 author, 1 change
public async Task handleAcceptedConection(Socket acceptedSocket)
{
    if (acceptingConnections)
    {
        Console.WriteLine("Accepted new client connection");
        ClientHandler newHandler = new ClientHandler(acceptedSocket);
        clientHandlers.Add(newHandler);
        await Task.Run(() => newHandler.StartHandling());
    }
}
```

ServerProgram

Antes:

```
static void Main(string[] args)
{
    var serverIpAddress = SettingsMgr.ReadSetting(ServerConfig.ServerIpConfigKey);
    var serverPort = SettingsMgr.ReadSetting(ServerConfig.SeverPortConfigKey);
    Console.WriteLine($"Server is starting in address {serverIpAddress} and port {serverPort}");

    Server server = new Server(serverIpAddress, serverPort);
    Console.WriteLine("Server will start accepting connections from the clients");

    Thread clientConnectionsThread = new Thread(() => server.StartReceivingConnections());
    clientConnectionsThread.Start();

    Thread serverExitPrompt = new Thread(() => server.ExitPrompt());
    serverExitPrompt.Start();

}
```

Después:

```
0 references | Agustina, 3 days ago | 3 authors, 8 changes
static async Task Main(string[] args)
{
    var serverIpAddress = SettingsMgr.ReadSetting(ServerConfig.ServerIpConfigKey);
    var serverPort = SettingsMgr.ReadSetting(ServerConfig.SeverPortConfigKey);
    Console.WriteLine($"Server is starting in address {serverIpAddress} and port {serverPort}");

    Server server = new Server(serverIpAddress, serverPort);
    StartServer(server);
    await Task.Run(() => server.ExitPrompt());
}
```

```
1 reference | Agustina, 3 days ago | 1 author, 1 change
public static async void StartServer(Server server)
{
    Console.WriteLine("Server will start accepting connections from the clients");

    await Task.Run(() => server.StartReceivingConnections());
}
```

ClientProgram:

Antes:

```
public void StartConnection()
{
    try
    {
        tcpClient.Connect(serverIpEndPoint);
    }
    catch (SocketException e)
    {
        Console.WriteLine("No se pudo conectar con el servidor");
        Environment.Exit(0);
    }
    networkStreamHandler = new NetworkStreamHandler(tcpClient.GetStream());
}
```

Después:

```
public async Task StartConnection()
{
    try
    {
        await socketClient.ConnectAsync(serverIpEndPoint);
    }
    catch (SocketException)
    {
        Console.WriteLine("No se pudo conectar con el servidor");
        Environment.Exit(0);
    }

    networkStreamHandler = new NetworkStreamHandler(new NetworkStream(socketClient));
}
```

Cambios en el protocolo

El protocolo no sufrió ningún cambio. La especificación se mantuvo igual y la manera de comunicación entre el cliente y el servidor siguió siendo: Cliente manda Request, Servidor Responde.

El cliente solo puede mandar una Request a la vez, principalmente porque se utiliza una CLI y no soporta dos menús al mismo tiempo.

Esta arquitectura de pedido-respuesta llevó a la utilización de awaits en gran parte del código para evitar que se puede aceptar/procesar varios pedidos sin que se mande la respuesta. Además, tampoco se pueden mandar distintas partes del protocolo al mismo tiempo, se tiene que seguir un orden (Req/Res, command, length, data) por lo tanto hay que esperar que se mande la primera antes de mandar la segunda.