Universidad ORT Uruguay

Obligatorio 2

Programación de Redes

Agustín Ferrari 240503

Francisco Rossi 219401

2021

Índice:

1. Descripción General: **3**
2. Modelo asincrónico .NET: 3
3. TCP Listener / TCP Client: 5

1. Descripción General:

En esta entrega se propuso cambiar los threads por un modelo asincrónico de .NET y cambiar los sockets por la librería de TCP Listener y TCP Client. Para esto nos basamos en la primera entrega, la cuál habíamos implementado Threads para el manejo de hilos y concurrencia entre varios clientes conectados a la misma vez al servidor y también utilizamos sockets como mecanismo de comunicación por medio de nuestro protocolo definido.

2. Modelo asincrónico .NET:

En esta instancia, nos propusimos realizar cambios en el manejo de varios clientes conectados a un mismo servidor de manera simultánea.
A partir de lo ya construido en el obligatorio anterior, nos enfocamos en cambiarlo orientado a un modelo que utiliza Task, en base a async y await.

Se buscó que todas las operaciones pesadas (recibir datos, enviar datos, leer archivos, escribir archivos) pasen a usar async await, para esto tuvimos que cambiar el file handler y socket handler, cambiando los métodos que realizaban estas funciones a ser sincrónicos, lo que hizo que también tengamos que cambiar la lógica del manejo de solicitudes del servidor a métodos asincrónicos para poder hacer awaits de los métodos que utilizan operaciones pesadas.

```
public override async Task HandleRequest(Header header, INetworkStreamHandler clientNetworkStreamHandler)
{
    string userName = await clientNetworkStreamHandler.ReceiveString(header.IDataLength);
    string responseMessageResult;
    if (_clientHandler.IsClientLogged(userName))
        responseMessageResult = ResponseConstants.LoginErrorAlreadyLogged;
    else
    {
        if (!_clientHandler.IsSocketInUse(clientNetworkStreamHandler))
        {
            _clientHandler.AddClient(clientNetworkStreamHandler, userName);
            _userController.TryAddUser(userName);
            responseMessageResult = ResponseConstants.LoginSuccess;
        }
        else
            responseMessageResult = ResponseConstants.LoginErrorSocketAlreadyInUse;
    }
    await clientNetworkStreamHandler.SendMessage(HeaderConstants.Response, CommandConstants.Login, responseMessageResult);
}
```

En este ejemplo podemos ver la strategy de Login en server, utilizando await para enviar y recibir datos, análogamente tuvimos que hacerlo del lado del cliente.

Otro ejemplo de los cambios realizados es:

```
private void ListenForConnections(Socket socketServer)
{
    ClientsConnectedSockets = new List<ISocketHandler>();
    while (!Exit)
    {
        try
        {
            IClientHandler clientHandler = ClientHandler.Instance;
            Socket clientConnected = socketServer.Accept();
            ISocketHandler clientConnectedHandler = new SocketHandler(clientConnected);
            ClientsConnectedSockets.Add(clientConnectedHandler);
            Console.WriteLine("Nueva conexion aceptada...");
            Thread threadcClient = new Thread(() => clientHandler.HandleClient(clientConnectedHandler));
            threadcClient.Start();
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
            Exit = true;
        }
    }
    Console.WriteLine("Exiting....");
}
```

En este ejemplo se puede observar cómo se utilizaban los threads para manejar las conexiones entrantes de los nuevos clientes al servidor. Cada vez que se enviaba la solicitud de un nuevo cliente queriendo conectarse al servidor, se creaba un thread nuevo que maneje los pedidos de acción del sistema del cliente.

```
private async Task ListenForConnections()
{
    ClientsConnectedSockets = new List<ISocketHandler>();
    while (!Exit)
    {
        try
        {
            IClientHandler clientHandler = ClientHandler.Instance;
            _tcpListener.Start(_supportedConnections);
            TcpClient tcpClient = await _tcpListener.AcceptTcpClientAsync().ConfigureAwait(false);
            _tcpListener.Stop();
            ISocketHandler clientConnectedHandler = new SocketHandler(tcpClient.GetStream());
            ClientsConnectedSockets.Add(clientConnectedHandler);
            Console.WriteLine("Nueva conexion aceptada...");
            Task.Run(async () => await clientHandler.HandleClient(clientConnectedHandler).ConfigureAwait(false));
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
            Exit = true;
        }
    }
    Console.WriteLine("Exiting....");
}
```

Se reemplazó el uso de thread con el uso de Task y async para poder ejecutar el HandleClient, manejando cada comando que el cliente ingrese en el sistema resultando en diferentes acciones.

3. TCP Listener / TCP Client:

Para la implementación habíamos realizado la entrega pasada con sockets, por lo cual pasamos a usar TCP Listener y TCP Client, para eso, dado el diseño que habíamos elegido para el obligatorio anterior fue necesario cambiar únicamente las clases que se encargaban de el manejo de los sockets. (SocketHandler, ClientSocketHandler y ServerSocketHandler).

Ejemplo:

```
public async Task ReceiveData(int Length, byte[] buffer)
{
    int iRecv = 0;
    while (iRecv < Length)
    {
        try
        {
            int localRecv = await _networkStream.ReadAsync(buffer, iRecv, Length - iRecv);
            bool connectionCloseOnRemoteEndPoint = localRecv == 0;
            if (connectionCloseOnRemoteEndPoint)
            {
                throw new SocketClientException();
            }
            iRecv += localRecv;
        }
        catch (IOException se)
        {
            Console.WriteLine(se.Message);
            return;
        }
    }
}
```

```
public void ReceiveData(int Length, byte[] buffer)
{
    var iRecv = 0;
    while (iRecv < Length)
    {
        try
        {
            int localRecv = _socket.Receive(buffer, iRecv, Length - iRecv, SocketFlags.None);
            bool connectionCloseOnRemoteEndPoint = localRecv == 0;
            if (connectionCloseOnRemoteEndPoint)
            {
                throw new SocketClientException();
            }
            iRecv += localRecv;
        }
        catch (SocketException se)
        {
            Console.WriteLine(se.Message);
            return;
        }
    }
}
```

Aca podemos ver el uso de NetworkStream (primera foto), refactorizando lo que hacíamos con Sockets, las funciones read de Socket y NetworkStream varían en que la de NetworkStream no necesita flags.

Análogamente cuando enviamos datos:

```
public async Task SendData(byte[] data)
{
    await _networkStream.WriteAsync(data, 0, data.Length);
}
```

```
public void SendData(byte[] data)
{
    int sentBytes = 0;
    while (sentBytes < data.Length)
    {
        sentBytes += _socket.Send(data, sentBytes, data.Length - sentBytes, SocketFlags.None);
    }
}
```

También tuvimos que cambiar el manejo de excepciones de Socket, ya que al utilizar NetworkStream, se lanzan diferentes excepciones, como por ejemplo IOException si intentamos recibir datos cuando el servidor se cerró.