Universidad ORT Uruguay
Facultad de Ingeniería

Programación de redes

Sistema Vapor

Nicolás Gibbs - 227347
Germán Konopka - 238880
Martin Robatto - 240935

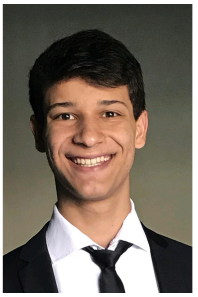
Tutores: Luis Barragué y Roberto Assandri

2021

Autoría

Nro. Estudiante	240935
Nombre:	Martín
Apellido:	Robatto
Grupo / Turno:	M6A

Nro. Estudiante	238880
Nombre:	German
Apellido:	Konopka
Grupo / Turno:	M6A



Nro. Estudiante	227347
Nombre:	Nicolás
Apellido:	Gibbs
Grupo / Turno:	M6A

Índice

Cambios respecto a la Versión anterior

En esta sección se describen los cambios realizados en el código respecto a la versión pasada. Primero, para esta versión migramos hacia los Sockets debido a que en la entrega anterior hicimos uso de la librería TCPListener/TCPClient.

Versión anterior:

```
private void Initialize()
{
    var clientIpAddress = _settingsManager.ReadSetting(ClientConfig.ClientIpConfigKey);
    var clientPort = GetPort();
    var ipEndPoint = new IPEndPoint(IPAddress.Parse(clientIpAddress), clientPort);
    _tcpClient = new TcpClient(ipEndPoint);
}
```

Versión actual:

```
private void Initialize()
{
    _clientSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
    var clientIpAddress = _settingsManager.ReadSetting(ClientConfig.ClientIpConfigKey);
    var clientPort = GetPort();
    var ipEndPoint = new IPEndPoint(IPAddress.Parse(clientIpAddress), clientPort);
    _clientSocket.Bind(ipEndPoint);
}
```

Este ejemplo muestra los cambios en el código al momento de inicializar el Socket, en la versión actual utilizamos el método Bind de Socket _clientSocket para enlazar el socket al IpEndPoint correspondiente. Antes, como se trataba de un Wrapper de alto nivel, realizando la creación del TcpClient con el IpEndPoint es suficiente.

Existen más casos (ejemplos) ubicados principalmente en la clase NetworkManager y las clases Handlers de Cliente y Servidor, pero los cambios principales son:

TCP Wrapper **(TCPClient + TCP Listener + NetworkStream)**	**Socket**
Write	Send
Read	Receive
-	Bind
Start	Listen
Connect	Connect
AcceptTcpClient	Accept
Close	Shutdown Close
Stop	Close

Además de esta migración, se realizó la transferencia del uso de Threads hacia las Tasks.
La clase Task generalmente se ejecuta de forma asincrónica. A través de lambda expressions es donde indicamos el trabajo que la tarea debe realizar.

Versión anterior:

```
public void Run()
{
    _exit = false;
    try
    {
        Initialize();
        _clientSockets = new Dictionary<TcpClient, NetworkStream>();
        var thread = new Thread(() => ListenForConnections());
        thread.IsBackground = true;
        thread.Start();
        ServerDisplay.Listening();
        ServerDisplay.Menu();
        while (!_exit)
        {
            var userInput = Console.ReadLine();
            switch (userInput)
            {
                case SHUTDOWN_SERVER:
                    ShutDown();
                    break;
                default:
                    ServerDisplay.InvalidOption();
                    break;
            }
        }
    }
    catch (Exception exception)
    {
        Console.WriteLine(exception.Message);
    }
}
```

Versión actual:

```
public void Run()
{
    _exit = false;
    try
    {
        Initialize();
        var task = Task.Run(async () => await ListenForConnectionsAsync().ConfigureAwait(false));
        ServerDisplay.Listening();
        ServerDisplay.Menu();
        while (!_exit)
        {
            var userInput = Console.ReadLine();
            switch (userInput)
            {
                case SHUTDOWN_SERVER:
                    ShutDown();
                    break;
                default:
                    ServerDisplay.InvalidOption();
                    break;
            }
        }
    }
    catch (Exception exception)
    {
        Console.WriteLine(exception.Message);
    }
}
```

Se puede observar que en la versión anterior lanzábamos un nuevo Thread para escuchar nuevas conexiones entrantes, e internamente en el método ListenForConnections() lanzábamos nuevos Thread por cada conexión aceptada con un cliente. Ahora, con el uso de Task indicamos que se ejecuta una nueva tarea en paralelo, utilizando una expresión lambda que emplea los términos async/await, para escuchar nuevas conexiones de manera asincrónica, e internamente al método ListenForConnections() se ejecutan Tasks de la misma manera por cada nueva conexión aceptada con un cliente.

Utilizamos el término async para convertir un método en asincrónico principalmente en operaciones de Entrada y Salida (I/O) en la red y el stream para los archivos. Conjuntamente con async, utilizamos el término await que envía la ejecución a segundo plano y nos devuelve el control al método que invoca. Finalmente, se destaca que seguimos la recomendación de agregar el sufijo Async al nombre del método.

Ejemplo:

```
private async Task SendDataAsync(Socket socket, byte[] data)
{
    int offset = 0;
    int size = data.Length;
    while (offset < size)
    {
        int sent = await socket.SendAsync(data, SocketFlags.None);
        if (sent == 0)
        {
            throw new SocketException();
        }
        offset += sent;
    }
}
```

Por último, es relevante mencionar que ninguna de estas migraciones tanto TCP a Socket y el uso de Task con el Await async provocó un impacto en nuestro protocolo del sistema. El protocolo nos quedó igual al de la versión anterior, en la documentación se pueden encontrar toda su descripción.