OBLIGATORIO II Programación de Redes

2021

Universidad ORT Uruguay
Profesor: Luis Barragué Martínez

Iñaki Etchegaray - 241072
Matías González - 219329

Índice

Abstract

El objetivo de esta entrega era simplemente cambiar el código de un modelo sincrónico con threads, a uno asincrónico con tasks y async/await.

Adicionalmente, se solicitaba el cambio del uso de un TCP Wrapper al uso explícito de Sockets.

El propósito de este documento es documentar las decisiones principales al realizar estos cambios.

Arreglos de la Primera Entrega

Al comenzar con este segundo obligatorio, previo a realizar los cambios en la conexión y de async/await, el grupo decidió primero solucionar los problemas conocidos de la primera entrega.

El primer problema ocurría a la hora de desloguear un usuario de forma manual, ya que luego se corría un deslogueo forzoso pensado para los casos donde el cliente pierda conexión y este segundo intento de deslogueo daba una excepción.

Este primer problema se solucionó removiendo una parte del código que se encontraba en un finally pero que era esperado corriese solo para ciertas excepciones.

El otro problema conocido era una excepción que no estaba siendo manejada correctamente. Rápidamente nos dimos cuenta que esto se debía que a la hora de asignar la constante de error se estaba asignando la equivocada, cambiando de SERVER_ERROR a CLIENT_ERROR y esto solucionando así el problema.

TcpWrapper a Socket

Para realizar el cambio de TcpListener/Client a Sockets fue necesario cambiar código en ambos: ClientHandler y ServerHandler. La mayoría de los cambios no eran de gran impacto, el layout general del código ya estaba provisto y la lógica no cambiaba mucho de si era un Socket o un Tcp Wrapper.

En la mayoría de los casos bastó con cambiar la operación del Wrapper a la equivalente del Socket como se ve en el siguiente cambio visto en nuestro SCM GitHub:

```
_tcpClient.Close();
_clientSocket.Shutdown(SocketShutdown.Both);
_clientSocket.Close();
```

A lo sumo bastaba con un cambio de línea por otro, o un cambio por dos, pero la lógica general del código se mantuvo igual. Otro ejemplo:

```
_tcpServerListener = new TcpListener(_serverIpEndPoint);
_serverSocket = new Socket(AddressFamily.InterNetwork,
                           SocketType.Stream,
                           ProtocolType.Tcp);

_serverSocket.Bind(_serverIpEndPoint);
```

Donde tenemos que explícitamente realizar lo que el TcpListener realiza por sí solo: el bind al endpoint del servidor definido previamente.

La desconexión del servidor a los clientes se mantuvo igual, creando un socket falso para terminar el loop de conexión, y en los rasgos más grandes se mantuvo la lógica del código.

Los cambios más grandes al código son resultantes del envío y recepción de los datos. Previamente, hacemos uso de una clase que hacía de wrapper para el NetworkStream del TcpClient:

```
public class NetworkStreamHandler : IStreamHandler
{
    3 references
    private readonly NetworkStream _stream;
    0 references
    public NetworkStreamHandler(NetworkStream stream)
    {
        _stream = stream;
    }
```

El cual se encargaba de manejar la lógica de leer o escribir al socket del otro lado. Obviamente, con el cambio a Sockets, el NetworkStream ya no corre, por lo que decidimos extender la interfaz de IStreamHandler a una versión que si maneje el stream de un socket:

```
public class SocketStreamHandler : IStreamHandler
{
    3 references
    private readonly Socket _socket;
    2 references
    public SocketStreamHandler(Socket socket)
    {
        _socket = socket;
    }
```

El cual implementa la misma interfaz que el anterior y maneja las escrituras y lecturas de manera adecuada para el socket.

Esto hace uso de la extensibilidad de la interfaz y solo basta con cambiar la instanciación a la del socket para que funcione el código.

Con esto, el cambio de TcpWrapper a Sockets se completó de manera satisfactoria.

Async/Await

Desde un primer momento fuimos conscientes que el cambio a Async/Await se esparciría por distintas partes del código, tanto Client como Server, y por ello decidimos comenzar desde lo más básico, esto siendo los Stream Handlers.

Estos cambios fueron sencillos, se cambiaron los métodos Read y Write a métodos asincrónicos que retornan un task y se los renombró de forma apropiada. Luego simplemente utilizamos los propios métodos asincrónicos de tanto la clase FileStream de System.IO como la clase NetworkStream dentro de System.Net.Sockets. Estos métodos corren en un await ya que de correrse más de un Write a la vez o de leer algo donde en simultáneo se está escribiendo se sobreescribirá la información y generaría errores o información corrupta.

Una vez modificados los Stream Handlers a async debimos adaptar los llamados a estos métodos a forma de que sean justamente de manera asincrónica. Es importante recordar por que esos read y write era necesario ejecutarlos con un await y no permitir que el código corra mientras estos ocurren ya que la misma lógica se cumplirá a la hora de aplicar estos cambios al VaporProtocol. Estas lecturas y escrituras deben ser esperadas ya que de seguir corriendo el código mientras aún se están ejecutando puede dar excepciones o incluso corromper archivos. Por esto debimos cambiar los métodos de Envío y Recibo de información dentro del protocolo para asegurarnos que cada llamada de estos métodos sea dentro de un await.

Es importante mencionar que se decidió no hacer awaits a la hora de bloquear la base de datos, debido a que no se ajusta al alcance del obligatorio, y por lo tanto la lógica de negocio no debería ser afectada de ninguna manera.

Una vez prontos los cambios en el paquete de Common, empezamos a investigar la modificación del Server al modelo de async/await.

El método de los cambios consistió en cambiar de sincrónico a el modelo asincrónico a los métodos en el código que:

- Podría tomar tiempo para realizarse.
- Implicara paralelismo con threads.
- Implicara uso de operaciones de red. Este último es consecuencia del primero.

Toda referencia a threads en el código fue reemplazada por Tasks, y todo inicio de threads nuevos fueron cambiados por inicios de task:

```
private void StartClientThread(Socket acceptedClientSocket)
{
    var clientThread = new Thread(() => HandleClient(acceptedClientSocket));
    clientThread.Start();
}
```

```
private async Task StartClientTask(Socket acceptedClientSocket)
{
    await Task.Run(async() => await HandleClient(acceptedClientSocket).ConfigureAwait(false));
}
```

Todo código de conexión/desconexión o de envío/recepción de información fueron cambiados a su formato Async, y a su vez los métodos que los contenían:

```
var foundClient = _serverSocket.Accept();
var foundClient = await _serverSocket.AcceptAsync();
```

```
vp.SendCommand(ReqResHeader.RES, response.Command, response.Response);
await vp.SendCommandAsync(ReqResHeader.RES, response.Command, response.Response);
```

Luego, a pesar de no hacer uso de Tasks explícitamente, modificamos de manera similar al cliente al modelo async/await. Por más que el cliente pudiera correr perfectamente de manera síncrona, decidimos realizar estos cambios por una cuestión de prolijidad y de cohesión con el servidor.

El approach fue el mismo y se cambiaron aquellos métodos que realizan envíos explícitos de información a través de la red o que implican conexión/desconexión de los sockets.

```
_vaporProtocol.ReceiveCover(path);
```

```
await _vaporProtocol.ReceiveCoverAsync(path);
```

Por último, como buena práctica de codificación, decidimos nombrar todos los métodos que fueran asincrónicos como el nombre del método con un Async al final de su nombre, para que los futuros codificadores del proyecto puedan saber rápidamente que la función es asincrónica.