Facultad de Ingeniería

2do Obligatorio Programación de Redes

21 de Mayo de 2024

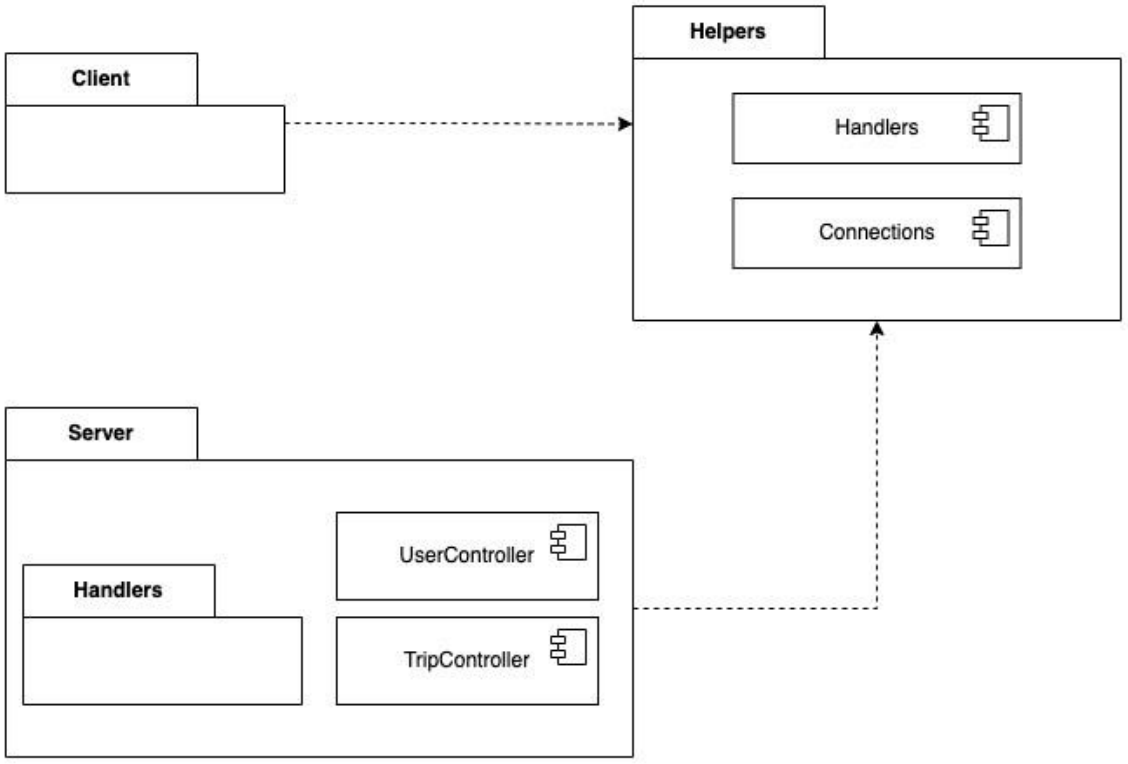
Índice.

Requisito Elegido:

b) “Agregar una opción al servidor que permita cerrar el servidor preguntando previamente si se quiere esperar a que los clientes actuales se desconecten o desconectarlos forzosamente. Se debe aprovechar las funcionalidades de cancelamiento de tareas que trae la clase Task”

Diagrama de Componentes:

Se adjunta un diagrama detallando la composición de los componentes del proyecto. En la documentación de la entrega anterior, se detalla las funcionalidades de cada uno pero no se había incluido una imagen con el diagrama de los mismos.



Comunicación

Protocolo:

Se aclara un poco acerca del protocolo, que no fue aclarado en la documentación anterior.

Para el protocolo de mensajes, el sender declara el largo del mensaje en un int, y envía ese int en formato de un array de 4 bytes en el header. El protocolo establece que el receptor debe leer los primeros 4 bytes e interpretarlos como el largo del mensaje, y luego, esperar a continuación a recibir los próximos n bytes del largo recibido.

Tasks Vs Threads

Para esta entrega y en pos de cumplir con la normativa, migramos nuestra estructura orientada en Threads a una nueva estructura basada en Task.

A continuación se presentan una serie de ventajas de usar Tasks en lugar de simplemente threads.

- **Modelo de Programación Simplificado:** Las tareas proporcionan una abstracción de mayor nivel para la concurrencia, lo que facilita escribir, leer y mantener el código asincrónico en comparación con trabajar directamente con hilos.
- **Valores de Retorno:** Las tareas pueden devolver valores a través del tipo *Task<TResult>*, lo que permite manejar de manera más sencilla los resultados de operaciones asincrónicas.
- **Cancelación:** Las tareas soportan la cancelación mediante el *CancellationToken*, permitiendo la cancelación cooperativa de operaciones asincrónicas. Esto es más complejo de implementar con hilos puros.
- **Continuación y Composición:** Las tareas soportan continuaciones, permitiendo especificar acciones que deben realizarse cuando una tarea se completa. Esto se facilita mediante métodos como *ContinueWith* y *await*. Las tareas también soportan la composición, permitiendo combinar múltiples tareas con métodos como *Task.WhenAll* y *Task.WhenAny*.
- **Utilización del Pool de Hilos:** Las tareas normalmente utilizan el pool de hilos, que es gestionado por el runtime de .NET. Esto permite una mejor gestión de recursos y optimización, ya que el runtime puede gestionar eficientemente el número de hilos y distribuir la carga de trabajo.
- **Integración con Async/Await:** Las tareas son la base de las palabras clave async y await en C#, que proporcionan una forma más natural e intuitiva de escribir código asincrónico.
- **Biblioteca de Paralelismo de Tareas (TPL):** La TPL proporciona potentes características para la programación paralela, incluyendo paralelismo de datos con *Parallel.For* y *Parallel.ForEach*, así como PLINQ (Parallel LINQ), que permite el procesamiento paralelo de consultas LINQ.

Usar una estructura basada en Task y tcpListener nos facilitó el requisito agregado en esta entrega. En especial poder hacer **await Task.WhenAll** para esperar a que todas las conexiones hayan finalizado.

Librerías TCP Client / TCP Listener

Para esta entrega, cambiamos el tipo de dato Socket y lo reemplazamos por los tipos TcpClient y TcpListener que son proporcionados por librerías.

Las bibliotecas TcpListener y TcpClient en .NET proporcionan abstracciones de más alto nivel para la comunicación de red basada en TCP en comparación con los sockets sin

procesar (Socket). A continuación algunas ventajas de usar TcpListener y TcpClient en lugar de trabajar directamente con Socket:

Modelo de Programación Simplificado: TcpListener y TcpClient proporcionan una interfaz más sencilla y fácil de usar para configurar y gestionar conexiones TCP, en comparación con la complejidad de trabajar directamente con los sockets.

Abstracción y Encapsulamiento: Estas clases encapsulan muchos detalles de bajo nivel, como la creación de sockets, la vinculación a direcciones IP y puertos, y la gestión de conexiones, lo que reduce la cantidad de código que necesitamos escribir y entender.

Facilidad de Uso: TcpClient y TcpListener proporcionan métodos y propiedades más intuitivas, lo que facilita tareas comunes como conectar a un servidor, aceptar conexiones entrantes y leer/escribir datos.

Integración con Streams: TcpClient proporciona acceso a NetworkStream, lo que permite leer y escribir datos usando las clases de flujo de datos (Stream) de .NET. Esto facilita la integración con otras partes del framework que trabajan con streams, como los lectores y escritores de streams.

Configuración y Personalización Simplificadas: Aunque proporcionan una abstracción de alto nivel, TcpClient y TcpListener aún permiten acceder a la configuración subyacente del socket si es necesario, proporcionando un buen equilibrio entre simplicidad y flexibilidad.

Compatibilidad con Asincronía: Ambas clases soportan métodos asincrónicos, lo que facilita la implementación de operaciones de red no bloqueantes, mejorando el rendimiento y la capacidad de respuesta de las aplicaciones.

Menor Probabilidad de Errores: Al reducir la cantidad de código de bajo nivel que se debe escribir, TcpClient y TcpListener disminuyen la probabilidad de errores comunes en la programación de sockets, como la gestión incorrecta de buffers y la sincronización de hilos.