

## Лабораторная работа 3

### «Одновременная обработка нескольких клиентов»

Целью данной лабораторной работы является доработка Эхо-сервера из предыдущей работа, а именно добавление возможности одновременного обслуживания нескольких клиентов.

Текущая версия сервера обладает серьезным недостатком - невозможность обработки нескольких клиентов, если второй клиент попытается подключиться к серверу во время работы с первым, то он будет стоять в очереди на обработку до тех пор, пока первый клиент не отключится от сервера. Это происходит из-за того что обмен данных между клиентом и сервером происходит в синхронном блокирующем режиме, когда сервер ожидает данные от подключенного клиента, он блокирует основной поток, не давая возможности исполнять никакой другой код. У решения этой проблемы есть несколько подходов:

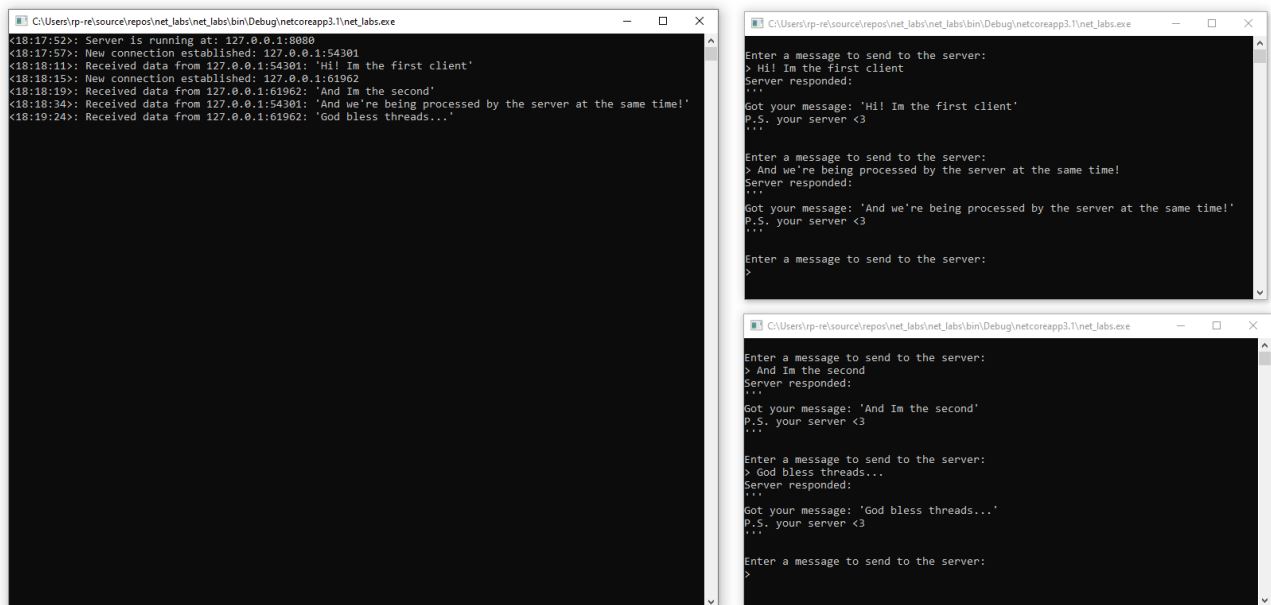
## Потоки

Самым простым решением будет вынести обработку клиентов в отдельный поток, в этом случае все блокирующие вызовы, возникающие при обработке очередного клиента, никак не будут отражаться на работе основного потока, принимающего подключения:

```
public void Listen() {  
    socket.Listen();  
    while (true) {  
        Socket client = socket.Accept();  
        new Thread( () => HandleConnection(client) ).Start();  
    }  
}
```

Изменение одной этой строчки кода уже даст желаемый результат - теперь сервер может обрабатывать несколько клиентов, однако у данного подхода есть и ряд недостатков. В рамках обработки запроса от клиента, создание потока это дорогая операция. У каждого потока своя память, свой стек, свое отдельное место с планировщике задач, затраты на поддержку работающего потока будут куда больше, чем на непосредственную обработку клиента. Плюс к этому, при одновременной обработке сильно большего кол-ва клиентов, чем количества ядер на машине, наш сервер, вероятно зависнет, из-за неспособности поддерживать такое большое число

активных потоков. Решить часть этих проблем можно с помощью асинхронной обработки клиентов в одном потоке.



```
C:\Users\ip-re\source\repos\net_lab3\net_lab3\bin\Debug\netcoreapp3.1\net_lab3.exe
<18:17:52>: Server is running at: 127.0.0.1:8080
<18:17:57>: New connection established: 127.0.0.1:54301
<18:18:11>: Received data from 127.0.0.1:54301: 'Hi! Im the first client'
<18:18:15>: New connection established: 127.0.0.1:61962
<18:18:19>: Received data from 127.0.0.1:61962: 'And Im the second'
<18:18:30>: Received data from 127.0.0.1:54301: 'And we're being processed by the server at the same time!'
<18:19:24>: Received data from 127.0.0.1:61962: 'God bless threads...'

C:\Users\ip-re\source\repos\net_lab3\net_lab3\bin\Debug\netcoreapp3.1\net_lab3.exe
Enter a message to send to the server:
> Hi! Im the first client
Server responded:
'''
Got your message: 'Hi! Im the first client'
P.S. your server <3
'''

Enter a message to send to the server:
> And we're being processed by the server at the same time!
Server responded:
'''
Got your message: 'And we're being processed by the server at the same time!'
P.S. your server <3
'''

Enter a message to send to the server:
>

C:\Users\ip-re\source\repos\net_lab3\net_lab3\bin\Debug\netcoreapp3.1\net_lab3.exe
Enter a message to send to the server:
> And Im the second
Server responded:
'''
Got your message: 'And Im the second'
P.S. your server <3
'''

Enter a message to send to the server:
> God bless threads...
Server responded:
'''
Got your message: 'God bless threads...'
P.S. your server <3
'''

Enter a message to send to the server:
>
```

## Асинхронная обработка

Обработка запроса через асинхронные функции, позволяет “поставить на паузу” обрабатывающую функцию в месте, где она блокирует основной поток и “возобновить” её исполнение, когда блокировка будет окончена. В момент пока асинхронная функция “стоит на паузе” основной поток может продолжать исполнять другой код.

В нашем случае функция, которую мы хотим иметь возможность ставить на паузу это функция обработки запросов `HandleRequest`, а другой код, который будет исполняться в это время - это цикл принятия новых подключений.

В языке C# асинхронность реализована следующим образом: чтобы сделать функцию асинхронной, необходимо пометить её ключевым словом `async`, внутри такой функции, вызывая блокирующую функцию (она также должна быть асинхронной) можно применить к ней оператор `await`, в этом месте исполнение функции будет поставлено на паузу, пока ожидаемая оператором `await` функция не завершится.

Очевидно, что для того чтобы воспользоваться преимуществами асинхронной работы, необходимо, чтобы сами блокирующие операции (в нашем случае это `socket.Receive`) были асинхронными, к счастью класс `Socket` предоставляет асинхронные аналоги всех методом сетевого взаимодействия (`AcceptAsync`, `ReceiveAsync`, `SendAsync` ...).

Тогда перевод нашего синхронного кода в асинхронный можно начать с того, чтобы сделать асинхронным метод `Listen` у нашего сервера - это позволит запускать в одном потоке сразу несколько серверов:

```
public async Task Listen() {
    socket.Listen();
    while (true) {
        Socket client = await socket.AcceptAsync();
        HandleConnection(client);
    }
}
```

Теперь исполнение функции Listen будет ставится на паузу до тех пор, пока к серверу не подключается новый клиент. Хотя теперь мы и можем запускать несколько серверов одновременно, но т.к. обработка клиента всё ещё синхронный процесс, вся программа в один момент времени может обслуживать только одного пользователя (сколько бы серверов не было запущено).

Исправим это, сделав саму функцию обработки асинхронной:

```
public virtual async void HandleConnection(Socket client) {
    while (true) {
        string request = await ReceiveAsync(client);
        ...
        SendAsync(response, client);
    }
    ...
}
```

Сделав описанные изменения, мы также получили желаемый результат, теперь сервер может обрабатывать несколько клиентов:

The image shows three screenshots of a terminal window running a .NET Core application. The terminal displays the following log output:

```
<19:00:08>: Server is running at: 127.0.0.1:8080
<19:00:11>: New connection established: 127.0.0.1:51264
<19:00:43>: Received data from 127.0.0.1:51264: 'Hi! I'm the first client'
<19:00:53>: New connection established: 127.0.0.1:51266
<19:00:57>: Received data from 127.0.0.1:51266: 'And I'm the second!'
<19:01:16>: Received data from 127.0.0.1:51264: 'And we're being processed by the server at the same time!'
<19:01:22>: Received data from 127.0.0.1:51266: 'God bless threads'
<19:01:51>: Received data from 127.0.0.1:51264: 'Or asynchrony?.....'
```

The client interactions show the server responding to multiple clients simultaneously:

```
Enter a message to send to the server:
> Hi! I'm the first client
Server responded:
...
Got your message: 'Hi! I'm the first client'
P.S. your server <3
...

Enter a message to send to the server:
> And we're being processed by the server at the same time!
Server responded:
...
Got your message: 'And we're being processed by the server at the same time!'
P.S. your server <3
...

Enter a message to send to the server:
> Or asynchrony?.....
Server responded:
...
Got your message: 'Or asynchrony?.....'
P.S. your server <3
...

Enter a message to send to the server:
>
...

Enter a message to send to the server:
> And I'm the second!
Server responded:
...
Got your message: 'And I'm the second!'
P.S. your server <3
...

Enter a message to send to the server:
> God bless threads
Server responded:
...
Got your message: 'God bless threads'
P.S. your server <3
...

Enter a message to send to the server:
>
...
```

# Потоки против асинхронности

Преимущество асинхронности перед потоками заключается в том, что исполнение всей программы происходит в одном основном потоке и не нужно тратить на создание, поддержание и переключение потоков. Такой подход приводит к выигрышу при выполнении большого числа “параллельных” задач, которые сами по себе с вычислительной точки зрения не подразумевают никакой сложности, например линейная обработка нескольких клиентов на сервере.

Поскольку у нас есть сервер, который может работать как в многопоточном, так и в асинхронном режиме - сравним эти два подхода с точки зрения производительности. На сервер одновременно подключится 50 клиентов, каждый пошлет 100 запросов с интервалами в 100 миллисекунд и затем отключится, замерим время ожидания ответа от сервера на каждый запрос каждого клиента в миллисекундах и проанализируем полученные данные:

	Async	Threads
count	5000.000000	5000.0000
mean	2.569800	3.9392
std	2.625815	2.0336
min	0.000000	1.0000
25%	2.000000	3.0000
50%	2.000000	4.0000
75%	3.000000	5.0000
max	44.000000	44.0000

(единицы измерения в таблице - время ответа сервера на один запрос в миллисекундах. Имитация 50 клиентов происходила на изолированной от сервера машине, чтобы не иметь влияния на результаты)

Как видно, асинхронный подход выиграл в сравнении с подходом “поток на клиента” (например медианное время ответа на запрос в два раза ниже на асинхронном сервере).

В итоговом приложении используется асинхронный подход, однако имеется реализация и многопоточного режима, которую можно включить вручную.

## Приложения

1. Код программы на GitHub: <https://github.com/proxodilka/web-labs>