Консольный TCP-чат

Последнее обновление: 10.11.2022

В прошлых темах tcp-клиенты получали и отправляли сообщения упорядочено: отправляли запрос - получали ответ и повторяли этот цикл. Однако нередко встречается ситуация, когда получение и отправка сообщений не связаны друг с другом. Банальный пример - чат, где человек может написать множество сообщений без относительно того, получит ли он на них какой-либо ответ. И, наоборот, получить много сообщений без отправки запросов. Для рассмотрения примера подобного взаимодействия напишем небольшую программу - консольный tcp-чат.

Определение сервера

Вначале создадим простейший консольный проект сервера. Определим в нем следующий код:

```
using System.Net;
using System.Net.Sockets;

ServerObject server = new ServerObject();// создаем сервер
await server.ListenAsync(); // запускаем сервер

class ServerObject
{
    TcpListener tcpListener = new TcpListener(IPAddress.Any, 8888); // сервер для прослушивания
    List<ClientObject> clients = new List<ClientObject>(); // все подключения
    protected internal void RemoveConnection(string id)
    {
        // получаем по id закрытое подключение
        ClientObject? client = clients.FirstOrDefault(c => c.Id == id);
        // и удаляем его из списка подключений
        if (client != null) clients.Remove(client);
        client?.Close();
    }
    // прослушивание входящих подключений
    protected internal async Task ListenAsync()
    {
        try
        {
            tcpListener.Start();
```

```
            Console.WriteLine("Сервер запущен. Ожидание подключений...");

            while (true)
            {
                TcpClient tcpClient = await tcpListener.AcceptTcpClientAsync();

                ClientObject clientObject = new ClientObject(tcpClient, this);
                clients.Add(clientObject);
                Task.Run(clientObject.ProcessAsync);
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            Disconnect();
        }
    }

    // трансляция сообщения подключенным клиентам
    protected internal async Task BroadcastMessageAsync(string message, string id)
    {
        foreach (var client in  clients)
        {
            if (client.Id != id) // если id клиента не равно id отправителя
            {
                await client.Writer.WriteLineAsync(message); //передача данных
                await client.Writer.FlushAsync();
            }
        }
    }
    // отключение всех клиентов
    protected internal void Disconnect()
    {
        foreach (var client in clients)
        {
            client.Close(); //отключение клиента
        }
        tcpListener.Stop(); //остановка сервера
    }
}
class ClientObject
{
    protected internal string Id { get;} = Guid.NewGuid().ToString();
    protected internal StreamWriter Writer { get;}
    protected internal StreamReader Reader { get;}

    TcpClient client;
    ServerObject server; // объект сервера

    public ClientObject(TcpClient tcpClient, ServerObject serverObject)
    {
        client = tcpClient;
        server = serverObject;
        // получаем NetworkStream для взаимодействия с сервером
        var stream = client.GetStream();
        // создаем StreamReader для чтения данных
        Reader = new StreamReader(stream);
        // создаем StreamWriter для отправки данных
```

```
        Writer = new StreamWriter(stream);
    }

    public async Task ProcessAsync()
    {
        try
        {
            // получаем имя пользователя
            string? userName = await Reader.ReadLineAsync();
            string? message = $"{userName} вошел в чат";
            // посылаем сообщение о входе в чат всем подключенным пользователям
            await server.BroadcastMessageAsync(message, Id);
            Console.WriteLine(message);
            // в бесконечном цикле получаем сообщения от клиента
            while (true)
            {
                try
                {
                    message = await Reader.ReadLineAsync();
                    if (message == null) continue;
                    message = $"{userName}: {message}";
                    Console.WriteLine(message);
                    await server.BroadcastMessageAsync(message, Id);
                }
                catch
                {
                    message = $"{userName} покинул чат";
                    Console.WriteLine(message);
                    await server.BroadcastMessageAsync(message, Id);
                    break;
                }
            }
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            // в случае выхода из цикла закрываем ресурсы
            server.RemoveConnection(Id);
        }
    }
    // закрытие подключения
    protected internal void Close()
    {
        Writer.Close();
        Reader.Close();
        client.Close();
    }
}
```

Весь код программы фактически разбивается на два класса: класс ServerObject представляет сервер, а класс ClientObject представляет подключение - отдельного клиента. Сначала рассмотрим код ClientObject.

Для создания объекта ClientObject вызывается конструктор, в котором устанавливаются поля и свойства класса:

```
protected internal string Id { get;} = Guid.NewGuid().ToString();
protected internal StreamWriter Writer { get;}
protected internal StreamReader Reader { get;}

```

```
TcpClient client;
ServerObject server; // объект сервера

public ClientObject(TcpClient tcpClient, ServerObject serverObject)
{
    client = tcpClient;
    server = serverObject;
    var stream = client.GetStream();
    Reader = new StreamReader(stream);
    Writer = new StreamWriter(stream);
}
```

Каждое подключение будет уникальным образом идентифицировано с помощью свойства Id, которое хранит значение Guid в строчном виде. Через конструктор получаем объект TcpClient для взаимодействия с подключенным клиентом и родительский объект ServerObject. Для отправки и получения сообщений для простоты применяются свойства Writer и Reader, которые представляют соответственно классы StreamWriter и StreamReader.

Основные действия происходят в методе `Process()`, в котором реализован простейший протокол для обмена сообщениями с клиентом. Так, в начале получаем имя подключенного пользователя, а затем в цикле получаем все остальные сообщения. Для трансляции этих сообщений всем остальным клиентам будет использоваться метод `BroadcastMessageAsync()` класса ServerObject.

Класс ServerObject представляет сервер. Он определяет две переменных: переменная tcpListener хранит объект TcpListener для прослушивания входящих подключений, а переменная clients представляет список, в который добавляются все подключенные клиенты.

```
TcpListener tcpListener = new TcpListener(IPAddress.Any, 8888); // сервер для прослушивания
List<ClientObject> clients = new List<ClientObject>(); // все подключения
```

Основной метод - `ListenAsync()`, в котором будет осуществляться прослушивание всех входящих подключений:

```
protected internal async Task ListenAsync()
{
    try
    {
        tcpListener.Start();
        Console.WriteLine("Сервер запущен. Ожидание подключений...");

        while (true)
        {
            TcpClient tcpClient = await tcpListener.AcceptTcpClientAsync();

            ClientObject clientObject = new ClientObject(tcpClient, this);
            clients.Add(clientObject);
            Task.Run(clientObject.ProcessAsync);
```

При получении подключения создаем для него объект ClientObject, добавляем его в список clients и запускаем новую задачу, в которой будет выполняться метод Process объекта ClientObject.

Для передачи сообщений всем клиентам, кроме отправившего, предназначен метод `BroadcastMessageAsync()`:

```
protected internal async Task BroadcastMessageAsync(string message, string id)
{
    foreach (var client in  clients)
    {
        if (client.Id != id) // если id клиента не равно id отправителя
        {
            await client.Writer.WriteLineAsync(message); //передача данных
            await client.Writer.FlushAsync();
        }
```

```
    }
}
```

Таким образом разделяются сущность подключенного клиента и сущность сервера.

И после определения классов ServerObject и ClientObject надо запустить прослушивание:

```
ServerObject server = new ServerObject();// создаем сервер
await server.ListenAsync(); // запускаем сервер
```

Здесь просто запускается новый поток, который обращается к методу `ListenAsync()` объекта ServerObject.

Это не идеальный проект сервера, но достаточный для базовой демонстрации организации кода и взаимодействия с клиентом.

Создание клиента

Теперь создадим новый консольный проект для клиента, который будет подключать к выше определенному серверу. Определим для клиента следующий код:

```
using System.Net.Sockets;

string host = "127.0.0.1";
int port = 8888;
using TcpClient client = new TcpClient();
Console.Write("Введите свое имя: ");
string? userName = Console.ReadLine();
Console.WriteLine($"Добро пожаловать, {userName}");
StreamReader? Reader = null;
StreamWriter? Writer = null;

try
{
    client.Connect(host, port); //подключение клиента
    Reader = new StreamReader(client.GetStream());
    Writer = new StreamWriter(client.GetStream());
    if (Writer is null || Reader is null) return;
    // запускаем новый поток для получения данных
    Task.Run(()=>ReceiveMessageAsync(Reader));
    // запускаем ввод сообщений
    await SendMessageAsync(Writer);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
Writer?.Close();
Reader?.Close();

// отправка сообщений
async Task SendMessageAsync(StreamWriter writer)
{
    // сначала отправляем имя
    await writer.WriteLineAsync(userName);
    await writer.FlushAsync();
    Console.WriteLine("Для отправки сообщений введите сообщение и нажмите Enter");

    while (true)
    {
        string? message = Console.ReadLine();
        await writer.WriteLineAsync(message);
```

```
        await writer.FlushAsync();
    }
}
// получение сообщений
async Task ReceiveMessageAsync(StreamReader reader)
{
    while (true)
    {
        try
        {
            // считываем ответ в виде строки
            string? message = await reader.ReadLineAsync();
            // если пустой ответ, ничего не выводим на консоль
            if (string.IsNullOrEmpty(message)) continue;
            Print(message);//вывод сообщения
        }
        catch
        {
            break;
        }
    }
}
// чтобы полученное сообщение не накладывалось на ввод нового сообщения
void Print(string message)
{
    if (OperatingSystem.IsWindows())    // если ОС Windows
    {
        var position = Console.GetCursorPosition(); // получаем текущую позицию курсора
        int left = position.Left;   // смещение в символах относительно левого края
        int top = position.Top;     // смещение в строках относительно верха
        // копируем ранее введенные символы в строке на следующую строку
        Console.MoveBufferArea(0, top, left, 1, 0, top + 1);
        // устанавливаем курсор в начало текущей строки
        Console.SetCursorPosition(0, top);
        // в текущей строке выводит полученное сообщение
        Console.WriteLine(message);
        // переносим курсор на следующую строку
        // и пользователь продолжает ввод уже на следующей строке
        Console.SetCursorPosition(left, top + 1);
    }
    else Console.WriteLine(message);
}
```

Код клиента также фактически состоит из двух функциональных частей: метод SendMessageAsync для отправки данных и метод ReceiveMessageAsync для получения данных.

Метод SendMessageAsync в качестве параметра получает объект StreamWriter, который, используя NetworkStream клиента, будет отправлять строку на сервер. А метод ReceiveMessageAsync получает объект StreamReader, через который будет считывать из NetworkStream присланное сообщение в виде строки.

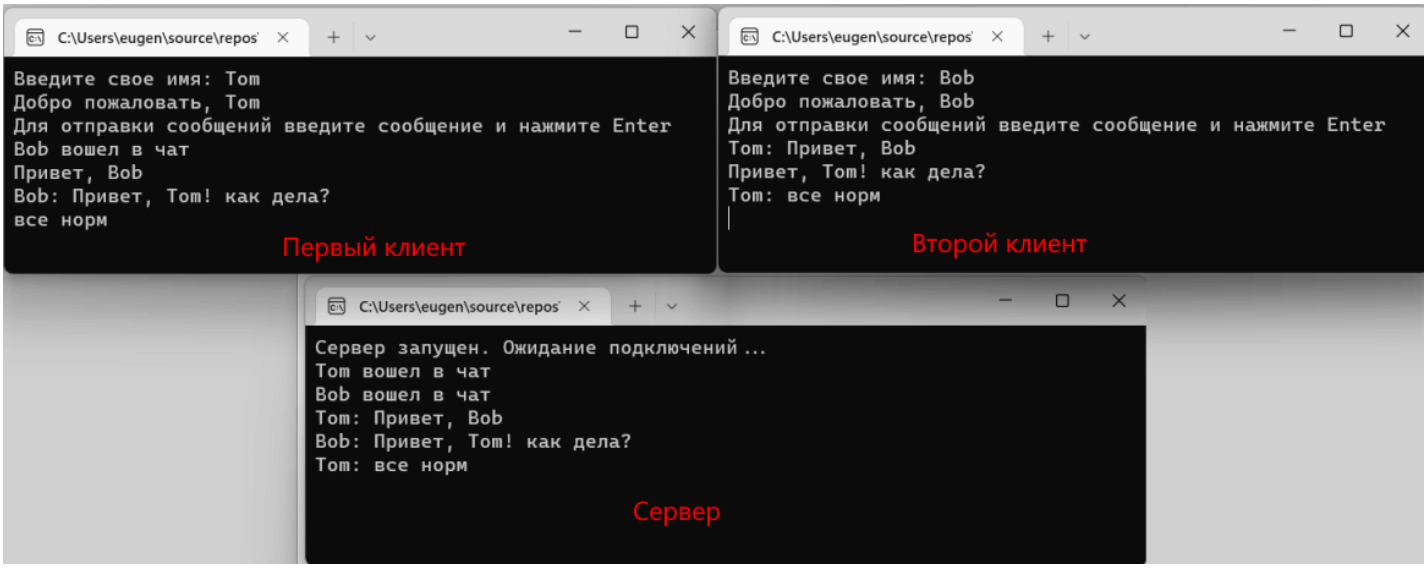
В качестве бонусного костыля приведен метод Print, который позволяет избежать вывода полученного сообщения в той же строке консоли, где пользователь вводит свое сообщение - в этом случае ввод просто переносится на следующую строку. Правда, это доступно на данный момент только на Windows.

Чтобы не блокировать ввод сообщений в главном потоке, для получения сообщений создается новая задача, которая обращается к методу ReceiveMessageAsync:

```
client.Connect(host, port); //подключение клиента
Reader = new StreamReader(client.GetStream());
Writer = new StreamWriter(client.GetStream());
```

```
4  if (Writer is null || Reader is null) return;
5
6  // запускаем новый поток для получения данных
7  Task.Run(()=>ReceiveMessageAsync(Reader));
8  await SendMessageAsync(Writer);
```

В конце запустим сервер и пару копий приложений клиента и протестируем их:



Назад **Содержание** **Вперед**

ТАКЖЕ НА METANIT.COM

Параметры маршрутов
4 месяца назад · 1 коммент...
Параметры маршрутов в приложении Blazor на C#, получение параметров ...

Все операции с БД в графическом ...
3 месяца назад · 2 коммент...
Выборка, сохранение, обновление и удаление в базу данных MS SQL ...

Встроенные компоненты ввода
4 месяца назад · 1 коммент...
Встроенные компоненты ввода Blazor из пространства имен ...

Использование NoSQL-хранилища ...
3 месяца назад · 1 коммент...
Использование документо-ориентированной базы ...

Взаимодейс кодом Pytho
4 месяца назад
Взаимодейств Python в прогр языке Си, уста

57 Комментариев Войти ▾

Присоединиться к обсуждению...

ВОЙТИ С ПОМОЩЬЮ ИЛИ ЧЕРЕЗ DISQUS ?

Имя

8 Поделиться Лучшие Новые Старые

bas-tion.ru
7 лет назад edited

Добавил клиенту в метод ReceiveMessage() биппер. Также добавил сообщение об уже введённых данных, если клиент получает сообщение не успев отправить своё:

```
string message = builder.ToString();
Console.Beep();
if (Console.CursorLeft > 0)
{
```

Помощь сайту

YooMoney:
410011174743222

Qiwi:

qiwi.com/n/METANIT

Перевод на карту
Номер карты:
4048415020898850

Вконтакте | Телеграм | Twitter | Помощь сайту

Контакты для связи: metanit22@mail.ru