Сетевое программирование в C# и .NET

Основы работы с сетями в C# и .NET

Введение в сети и протоколы

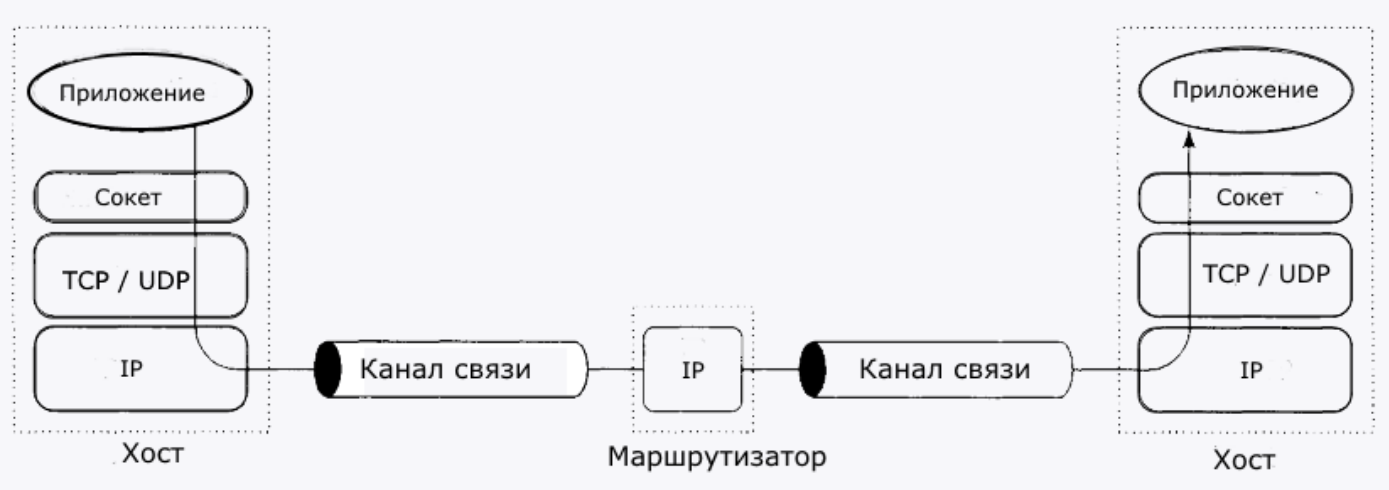
Сегодня миллионы компьютеров и устройств связаны в глобальную сеть интернет, либо в отдельные локальные подсети. В связи с этим возникает необходимость создания приложений, которые бы использовали все преимущества передачи данных по сети. Например, одним из распространенных приложений, которое использует передачу по сети, является веб-браузер. И платформа .NET и язык программирования C# предоставляют все необходимые возможности для создания приложений, которые могут взаимодействовать по сети и использовать различные сетевые протоколы.

Но прежде чем переходить непосредственно к созданию приложений, надо пару слов сказать, что вообще представляет собой коммуникация в сети.

Вся сеть состоит из отдельных элементов - хостов, которые представляют собой компьютеры и другие подключенные устройства. Между собой они соединены каналами связи (кабели Ethernet, Wi-Fi и т.д.) и маршрутизаторами. Маршрутизаторы объединяют компьютеры в подсети и контролируют передачу данных между ними.

Но компьютеры-хосты не взаимодействуют абы как между собой. Они применяют **протоколы**. Протокол представляет собой соглашения о том, как пакеты данных будут передаваться по каналам коммуникации. Таким образом, протокол упорядочивает взаимодействие.

Существует множество различных протоколов. Протоколы, которые используются для передачи данных по сети, составляют семейство протоколов TCP/IP. Основные из них: Internet Protocol (IP), Transmission Control Protocol (TCP) и User Datagram Protocol (UDP). Причем эти протоколы организованы в уровневую систему:



IP представляет сетевой уровень. Он использует нижележащие уровни, которые представляют физические каналы коммуникации - кабели Ethernet и т.д., для передачи пакетов с данными другому хосту.

Выше IP располагается транспортный уровень, который образуют протоколы TCP и UDP. Эти протоколы используют определенные порты для передачи данных. TCP позволяет отследить потерю

пакетов и их дублирование при передаче. UDP подобного не позволяет сделать и нацелен на простую передачу данных.

Однако приложение взаимодействует с уровнем TCP / UDP не напрямую, а через специальный API, который предоставляют **сокеты**. Сокеты - это не какой-либо протокол, это просто интерфейс для создания сетевых приложений, который опирается на встроенные возможности операционной системы.

В зависимости от используемого протокола различают два вида сокетов: потоковые сокеты (stream socket) и дейтаграммные сокеты (datagram socket). Потоковые сокеты используют протокол TCP, дейтаграммные - протокол UDP.

В итоге, когда приложение посылает сообщение приложению, запущенному на другом хосте, то приложение обращается к сокетам для передачи данных на уровень TCP / UDP. Далее с этого транспортного уровня данные передаются сетевому уровню - уровню протокола IP. И этот протокол передает данные далее физическим уровням, и после этого данные уходят по сети.

Чтобы уникально определять хосты в сети каждый хост имеет адрес. Существует несколько различных протоколов адресов. В настоящее время наиболее распространен протокол IPv4, который предполагает представление адреса в виде 32-битного числа, например, 37.120.16.63. Такой адрес содержит четыре числа, разделенных точками, и каждое число находится в диапазоне от 0 до 255. Однако также в последнее время набирает оборот использование адресов протокола IPv6, которые представляют собой 128-битное значение.

Однако такие адреса очень сложно запомнить, поэтому в реальности чаще оперируют доменами. Домены представляют специальные названия, используемые для интернет-адресов. Например, есть доменное имя "www.microsoft.com", ему соответствует адрес в формате IPv4 2.23.143.150. Но для протокола IP, через который идет взаимодействие, доменные адреса не существуют. Поэтому при отправке или передаче данных по доменному имени, компьютер еще обращается к службам Domain Name System (DNS), который выполняют сопоставление между интернет-адресами в формате IPv4 или IPv6 и доменными названиями.

Кроме адреса при сетевых взаимодействиях используются **порты**. Порт представляет 16-битное число в диапазоне от 1 до 65 535. Использование портов позволяет разграничить несколько запущенных приложений на одном хосте.

Собственно, это все базовые принципы взаимодействия по сети, которые надо знать. В реальности, как правило, при создании приложений не потребуется глубокого знания всех протоколов и нюансов их работы. Если в редких случаях возникнет необходимость более детального знания протоколов, то в этом случае можно обратиться к специализированной литературе, например, к книгам Олифера или Танненбаума.

Адреса в .NET

Прежде чем отправить запрос к какому-нибудь ресурсу, компьютер обращается к DNS-северу, чтобы по имени ресурса получить его ip-адрес. И затем уже обращается по этому ip-адресу.

Все ip-адреса представляют 32-битное (протокол IPv4) или 128-битное значение (протокол IPv6), например, `31.170.165.181`.

В системе классов .NET ip-адрес представлен классом **IPAddress**. Этот класс позволяет управлять адресами с помощью следующих свойств и методов:

- Метод `Parse()`: преобразует строковое представление адреса в IPAddress

  ```
  IPAddress ip = IPAddress.Parse("127.0.0.1"); // ip указывает на локальный адрес
  ```

- Статическое свойство `Loopback`: возвращает объект IPAddress для адреса 127.0.0.1. Аналогично вышеприведенному коду
- Статическое свойство `Any`: возвращает объект IPAddress для адреса 0.0.0.0
- Статическое свойство `Broadcast`: возвращает объект IPAddress для адреса 255.255.255.255

Также для получения адреса в сети используется класс **IPHostEntry**. Он содержит информацию об определенном компьютере-хосте.

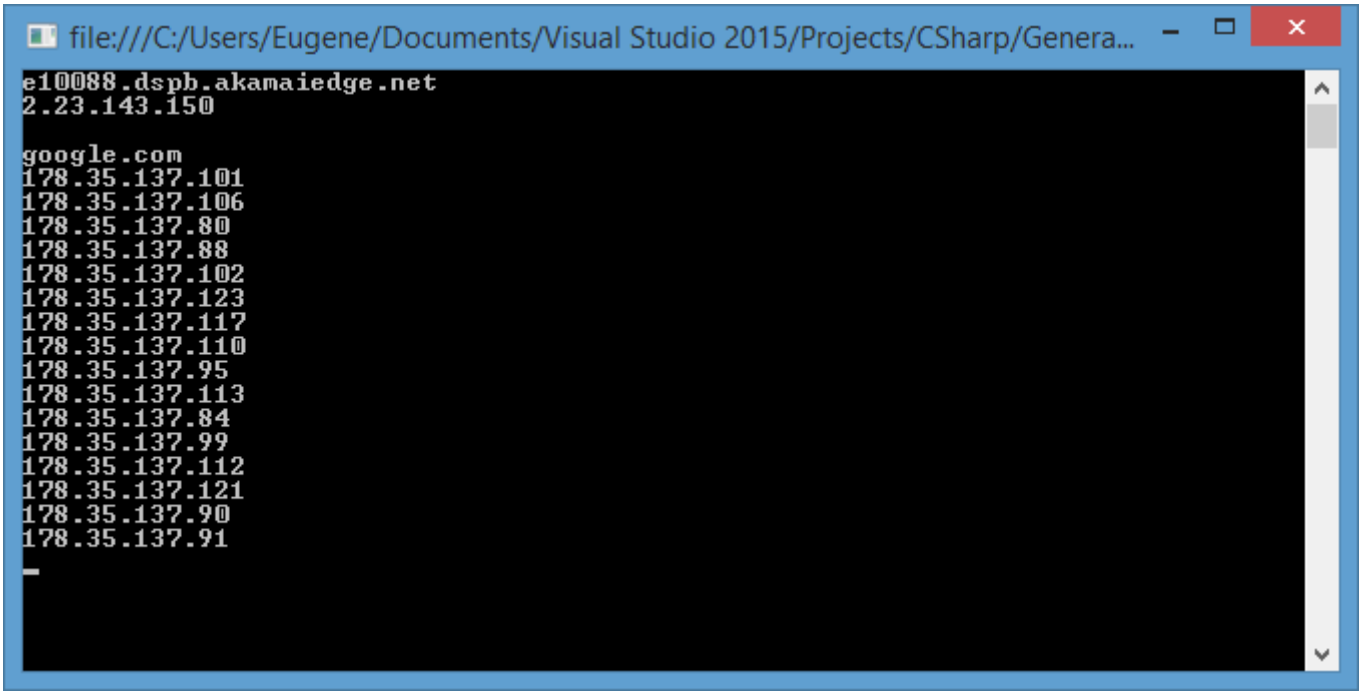
С помощью свойства **HostName** этот класс возвращает имя хоста, а с помощью свойства **AddressList** - все ip-адреса хоста, так как один компьютер может иметь в сети несколько ip-адресов.

Для взаимодействия с dns-сервером и получения ip-адреса применяется класс **Dns**. Для получения информации о хосте компьютера и его адресах у него имеется метод **GetHostEntry()**:

```
IPHostEntry host1 = Dns.GetHostEntry("www.microsoft.com");
Console.WriteLine(host1.HostName);
foreach (IPAddress ip in host1.AddressList)
    Console.WriteLine(ip.ToString());

Console.WriteLine();

IPHostEntry host2 = Dns.GetHostEntry("google.com");
Console.WriteLine(host2.HostName);
foreach (IPAddress ip in host2.AddressList)
    Console.WriteLine(ip.ToString());
```


file:///C:/Users/Eugene/Documents/Visual Studio 2015/Projects/CSharp/Genera...
e10088.dspb.akamaiedge.net
2.23.143.150
google.com
178.35.137.101
178.35.137.106
178.35.137.80
178.35.137.88
178.35.137.102
178.35.137.123
178.35.137.117
178.35.137.110
178.35.137.95
178.35.137.113
178.35.137.84
178.35.137.99
178.35.137.112
178.35.137.121
178.35.137.90
178.35.137.91

Отправка запросов

Класс WebClient

Для загрузки файлов с определенных ресурсов предназначен класс **WebClient**, который находится в пространстве имен System.Net.

Самый простой способ загрузки предоставляет метод **DownloadFile()**. Например, загрузим файл с какого-нибудь сайта:

```
WebClient client = new WebClient();
client.DownloadFile("http://somesite.com/book.pdf", "myBook.pdf");
Console.WriteLine("Файл загружен");
```

Метод DownloadFile() принимает два параметра. Первый параметр указывает на файл в интернете, который надо загрузить, а второй параметр определяет имя загруженного файла на локальном компьютере. После загрузки в папке с приложением появится файл myBook.pdf.

WebClient также позволяет получить файл в качестве потока и затем манипулировать этим потоком в процессе загрузки:

```
using System;
using System.Net;
using System.IO;

namespace NetConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            WebClient client = new WebClient();

                        using (Stream stream = client.OpenRead("http://somesite.com/sometext.txt"))
            {
                using (StreamReader reader = new StreamReader(stream))
                {
                    string line = "";
                    while ((line = reader.ReadLine()) != null)
                    {
                        Console.WriteLine(line);
                    }
                }
            }

              Console.WriteLine("Файл загружен");
          Console.Read();
          }
```

```
    }
}
```

Для открытия потока используется метод **OpenRead()**, в который передается адрес файла.

Для чтения потока применяется класс StreamReader. Затем прочитанные строки выводятся на консоль.

Загрузку файла можно осуществлять в асинхронном режиме:

```
using System;
using System.Net;
using System.Threading.Tasks;

namespace NetConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            DownloadFileAsync().GetAwaiter();

            Console.WriteLine("Файл загружен");
            Console.Read();
        }

        private static async Task DownloadFileAsync()
        {
            WebClient client = new WebClient();
            await client.DownloadFileTaskAsync(new Uri("http://somesite.com/myfile.txt"),
"mytxtFile.txt");
        }
    }
}
```

Классы WebRequest и WebResponse

Для отправки запроса предназначен класс **WebRequest**. Класс **WebResponse** позволяет получить ответ на запрос.

Выполним простейший запрос:

```
using System;
using System.Net;
using System.IO;
using System.Threading.Tasks;

namespace NetConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            WebRequest request = WebRequest.Create("http://somesite.com/myfile.txt");
            WebResponse response = request.GetResponse();
            using (Stream stream = response.GetResponseStream())
            {
                using (StreamReader reader = new StreamReader(stream))
                {
                    string line = "";
                    while ((line = reader.ReadLine()) != null)
                    {
                        Console.WriteLine(line);
                    }
                }
            }
                        response.Close();
            Console.WriteLine("Запрос выполнен");
            Console.Read();
        }
    }
}
```

Процесс отправки запроса разбивается на несколько этапов:

1. Создание объекта WebRequest с помощью метода `Create()`, в который передается адрес ресурса с виде строки или объекта Uri:

   ```
   WebRequest request = WebRequest.Create("http://somesite.com/myfile.txt");
   ```

2. Отправка запроса и получение ответа:

   ```
   WebResponse response = request.GetResponse();
   ```

3. Получение потока ответа и манипуляции с ним:

```
Stream stream = response.GetResponseStream();
```

Для получения информации, специфичной для протокола HTTP, используются два класса: **HttpWebRequest** и **HttpWebResponse**, которые наследуются соответственно от WebRequest и WebResponse.

Использование HttpWebRequest и HttpWebResponse во многом аналогично:

```
private static async Task RequestAsync()
{
    HttpWebRequest request =
(HttpWebRequest)WebRequest.Create("http://localhost:8080/mytxtFile.txt");
    HttpWebResponse response = (HttpWebResponse)await request.GetResponseAsync();
    using (Stream stream = response.GetResponseStream())
    {
        using (StreamReader reader = new StreamReader(stream))
        {
            Console.WriteLine(reader.ReadToEnd());
        }
    }
        response.Close();
}
```

Некоторые важные свойства класса HttpWebRequest:

- `TimeOut`: указывает на время в миллисекундах, допустимое для ожидания ответа от сервера (ожидание выполнения методов`GetResponse()` и `GetRequestStream()`). По умолчанию имеет значение 100 000 миллисекунд (100 секунд)
- `KeepAlive`: при значении `true` позволяет устанавливать постоянные подключения к серверу. В итоге для нескольких запросов можно будет использовать одно и то же подключение, что сэкономит время на открытие/закрытие нового подключения. По умолчанию имеет значение true.
- `AllowAutoRedirect`: указывает, должен ли запрос следовать ответам переадресации. При значении `true` запрос автоматически будет использовать переадресацию. Чтобы запретить переадресацию, надо установить значение `false`. По умолчанию имеет значение true.

  Также можно использовать свойство `MaximumAutomaticRedirections`, которое устанавливает максимальное количество переадресаций
- `Credentials`: представляет объект **NetworkCredential**, который устанавливает идентификацию пользователя (логин, пароль)

Некоторые важные свойства класса HttpWebResponse:

- `ContentLength`: значение заголовка Content-Length, возвращаемого с ответом, и возвращает длину содержимого в байтах в запросе (длина заголовков в этот объем не включается)
- `Cookies`: представляет объект CookieCollection и возвращает файлы куки, которые связаны с ответом
- `Headers`: Возвращает заголовки, связанные с данным ответом

- `LastModified`: возвращает дату и время последнего изменения содержимого ответа

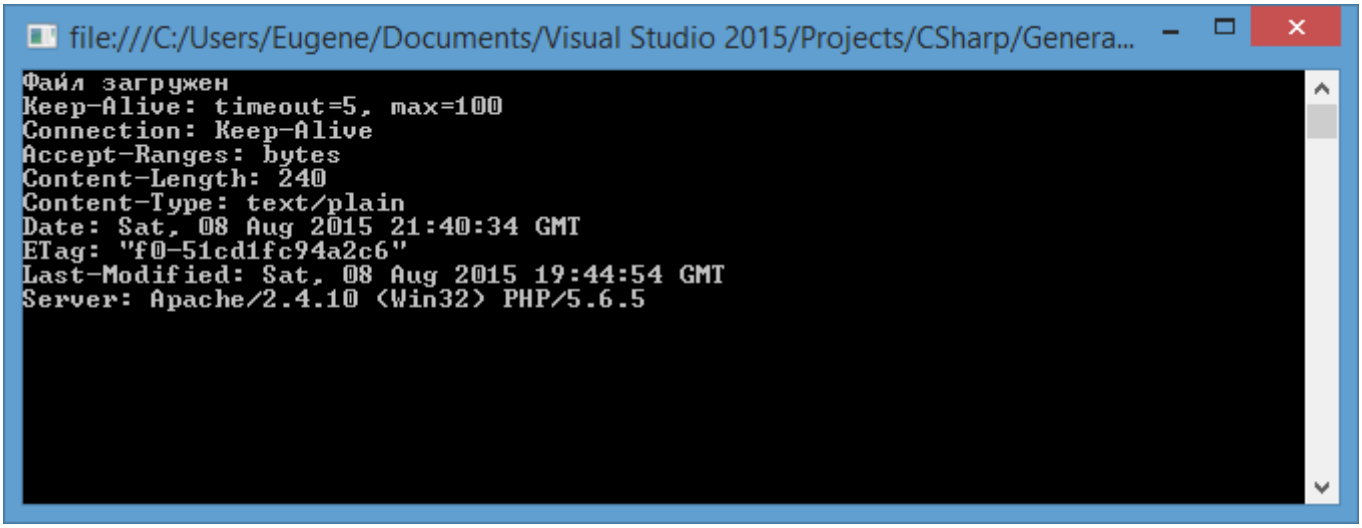
Установка логина и пароля в запросе:

```
HttpWebRequest request = (HttpWebRequest)WebRequest.Create("http://somesite.com/auth");
request.Credentials = new NetworkCredential("myLogin", "myPassword");
HttpWebResponse response = (HttpWebResponse)await request.GetResponseAsync();
```

В данном случае используется базовая HTTP-аутентификация.

Получим, к примеру, заголовки ответа:

```
HttpWebRequest request = (HttpWebRequest)WebRequest.Create("http://somesite.com/mytxtFile.txt");
HttpWebResponse response = (HttpWebResponse)await request.GetResponseAsync();
WebHeaderCollection headers = response.Headers;
for(int i=0; i< headers.Count; i++)
{
    Console.WriteLine("{0}: {1}", headers.GetKey(i), headers[i]);
}
response.Close();
```



Отправка данных в запросе

Если мы хотим отправить данные в запросе, то нам надо добавить их в поток запроса. Например, мы хотим обратиться к веб-приложению ASP.NET MVC, в котором определен контроллер Home и метод PostData, принимающий данные:

```
public class HomeController : Controller
{
    [HttpPost]
    public string PostData(string sName)
    {
        return "Параметр запроса: " + sName;
    }
}
```

В данном случае конкретная технология не важна. Это мог бы быть и скрипт PHP, который принимает данные:

```
<?php
$sName = "Не известно";
if(isset($_POST['sName'])) $sName = $_POST['sName'];
echo "Параметр запроса: $sName";
?>
```

Теперь передадим ресурсу данные:

```
private static async Task PostRequestAsync()
{
    WebRequest request = WebRequest.Create("http://localhost:5374/Home/PostData");
    request.Method = "POST"; // для отправки используется метод Post
    // данные для отправки
    string data = "sName=Hello world!";
    // преобразуем данные в массив байтов
    byte[] byteArray = System.Text.Encoding.UTF8.GetBytes(data);
    // устанавливаем тип содержимого - параметр ContentType
    request.ContentType = "application/x-www-form-urlencoded";
    // Устанавливаем заголовок Content-Length запроса - свойство ContentLength
    request.ContentLength = byteArray.Length;

    //записываем данные в поток запроса
    using (Stream dataStream = request.GetRequestStream())
    {
        dataStream.Write(byteArray, 0, byteArray.Length);
    }

    WebResponse response = await request.GetResponseAsync();
    using (Stream stream = response.GetResponseStream())
    {
```

```
        using (StreamReader reader = new StreamReader(stream))
        {
            Console.WriteLine(reader.ReadToEnd());
        }
    }
    response.Close();
    Console.WriteLine("Запрос выполнен...");
}
```

Тут надо отметить следующий момент. Мы точно знаем, что параметр, который получает веб-ресурс, должен называться "sName". Поэтому при запросе формируются данные в формате *название_параметра=данные*: `"sName=Hello world!"`

Еще также надо отметить, что для кодирования данных устанавливается тип содержимого "application/x-www-form-urlencoded".

После получения потока запроса ()`Stream dataStream = request.GetRequestStream()` данные добавляются в запрос.

А после обращения к веб-ресурсу консоль отобразит нам ответ:

```
Параметр запроса: Hello world!
Запрос выполнен...
```

Если веб-ресурс должен принять несколько параметров, например:

```
[HttpPost]
public string PostData(string sName, int age)
{
    return "Параметры запроса: " + sName + " " + age.ToString();
}
```

То при отправке параметры разделяются амперсандом:

```
WebRequest request = WebRequest.Create("http://localhost:5374/Home/PostData");
request.Method = "POST";
string sName = "sName=Иван Иванов&age=31";
byte[] byteArray = System.Text.Encoding.UTF8.GetBytes(sName);
request.ContentType = "application/x-www-form-urlencoded";
request.ContentLength = byteArray.Length;

using (Stream dataStream = request.GetRequestStream())
{
    dataStream.Write(byteArray, 0, byteArray.Length);
}
```

Ну и если нам надо отправить запрос GET, то все параметры можно передать в строке запроса:

```
WebRequest request = WebRequest.Create("http://localhost:5374/Home/PostData?sName=Иван
Иванов&age=31");
WebResponse response = await request.GetResponseAsync();
using (Stream stream = response.GetResponseStream())
{
    using (StreamReader reader = new StreamReader(stream))
   {
```

```
        Console.WriteLine(reader.ReadToEnd());
    }
}
response.Close();
```

Обработка ошибок при запросах

При работе с классами WebRequest и WebResponse могут возникать исключения. Непосредственно к работе с сетью относится класс исключений **WebException**. Его ключевым свойством является свойство **Status**, которое позволяет получит тип исключения.

Это свойство принимает одно из значений перечисления **WebExceptionStatus**:

- `ConnectFailure`: невозможно подключиться к ресурсу
- `ConnectionClosed`: подключение было преждевременно закрыто
- `KeepAliveFailure`: сервер закрыл подключение, для которого был установлен заголовок Keep-Alive
- `MessageLengthLimitExceeded`: запрос превышает допустимый размер
- `NameResolutionFailure`: служба DNS не может сопоставить имя хоста с ip-адресом
- `ProtocolError`: запрос был завершен, но возникла ошибка на уровне протокола, например, запрошенный ресурс не был найден
- `ReceiveFailure`: от удаленного сервера не был получен полный ответ
- `RequestCanceled`: запрос был отменен
- `SecureChannelFailure`: при установлении подключения с использованием SSL произошла ошибка
- `SendFailure`: полный запрос не был передан на удаленный сервер
- `ServerProtocolViolation`: ответ сервера не являлся допустимым ответом HTTP
- `Timeout`: ответ не был получен в течение определенного времени
- `TrustFailure`: сертификат сервера не может быть проверен
- `UnknownError`: возникло исключение неизвестного типа

Обработка исключения и возможное логгирование ошибок позволит определить природу неполадки, что в дальнейшем поможет найти и исправить ошибку. Например, обратимся к несуществующему ресурсу:

```
try
{
    WebRequest request = WebRequest.Create("http://localhost:5374/Home/PostData");
    using (WebResponse response = request.GetResponse())
    {
        using (Stream stream = response.GetResponseStream())
        {
            using (StreamReader reader = new StreamReader(stream))
           {
```

```
                Console.WriteLine(reader.ReadToEnd());
            }
        }
    }
}
catch(WebException ex)
{
    // получаем статус исключения
    WebExceptionStatus status = ex.Status;

    if (status == WebExceptionStatus.ProtocolError)
    {
        HttpWebResponse httpResponse = (HttpWebResponse)ex.Response;
        Console.WriteLine("Статусный код ошибки: {0} - {1}",
                                (int)httpResponse.StatusCode, httpResponse.StatusCode);
    }
}
```

И консоль выведет сообщение об ошибке:

```
Статусный код ошибки: 404 - Not Found
```

Сокеты

Класс Socket

В основе межсетевых взаимодействий по протоколам TCP и UDP лежат сокеты. В .NET сокеты представлены классом **System.NET.Sockets.Socket**, который предоставляет низкоуровневый интерфейс для приема и отправки сообщений по сети.

Рассмотрим основные свойства данного класса:

- `AddressFamily`: возвращает все адреса, используемые сокетом. Данное свойство представляет одно из значений, определенных в одноименном перечислении **AddressFamily**. Перечисление содержит 18 различных значений, наиболее используемые:
    - `InterNetwork`: адрес по протоколу IPv4
    - `InterNetworkV6`: адрес по протоколу IPv6
    - `Ipx`: адрес IPX или SPX
    - `NetBios`: адрес NetBios
- `Available`: возвращает объем данных, которые доступны для чтения
- `Connected`: возвращает true, если сокет подключен к удаленному хосту
- `LocalEndPoint`: возвращает локальную точку, по которой запущен сокет и по которой он принимает данные
- `ProtocolType`: возвращает одно из значений перечисления **ProtocolType**, представляющее используемый сокетом протокол. Есть следующие возможные значения:
    - `Ggp`
    - `Icmp`
    - `IcmpV6`
    - `Idp`
    - `Igmp`
    - `IP`
    - `IPSecAuthenticationHeader` (Заголовок IPv6 AH)
    - `IPSecEncapsulatingSecurityPayload` (Заголовок IPv6 ESP)
    - `IPv4`
    - `IPv6`
    - `IPv6DestinationOptions` (Заголовок IPv6 Destination Options)

- IPv6FragmentHeader (Заголовок IPv6 Fragment)
- IPv6HopByHopOptions (Заголовок IPv6 Hop by Hop Options)
- IPv6NoNextHeader (Заголовок IPv6 No next)
- IPv6RoutingHeader (Заголовок IPv6 Routing)
- Ipx
- ND
- Pup
- Raw
- Spx
- SpxII
- Tcp
- Udp
- Unknown (неизвестный протокол)
- Unspecified (неуказанный протокол)

Каждое значение представляет соответствующий протокол, но наиболее используемыми являются Tcp и Udp.

- RemoteEndPoint: возвращает адрес удаленного хоста, к которому подключен сокет
- SocketType: возвращает тип сокета. Представляет одно из значений из перечисления **SocketType**:
  - Dgram: сокет будет получать и отправлять дейтаграммы по протоколу Udp. Данный тип сокета работает в связке с типом протокола - Udp и значениемAddressFamily.InterNetwork
  - Raw: сокет имеет доступ к нижележащему протоколу транспортного уровня и может использовать для передачи сообщений такие протоколы, как ICMP и IGMP
  - Rdm: сокет может взаимодействовать с удаленными хостами без установки постоянного подключения. В случае, если отправленные сокетом сообщения невозможно доставить, то сокет получит об этом уведомление
  - Seqpacket: обеспечивает надежную двустороннюю передачу данных с установкой постоянного подключения
  - Stream: обеспечивает надежную двустороннюю передачу данных с установкой постоянного подключения. Для связи используется протокол TCP, поэтому этот тип сокета используется в паре с типом протокола Tcp и значением AddressFamily.InterNetwork
  - Unknown: адрес NetBios

Для создания объекта сокета можно использовать один из его конструкторов. Например, сокет, использующий протокол Tcp:

```
Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
```

Или сокет, использующий протокол Udp:

```
Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
```

Таким образом, при создании сокета мы можем указывать разные комбинации протоколов, типов сокета, значений из перечисления AddressFamily. Однако в то же время не все комбинации являются корректными. Так, для работы через протокол Tcp, нам надо обязательно указать параметры: AddressFamily.InterNetwork, SocketType.Stream и ProtocolType.Tcp. Для Udp набор параметров будет другим: AddressFamily.InterNetwork, SocketType.Dgram и ProtocolType.Udp. Для других протоколов набор значений будет отличаться. Поэтому использование сокетов может потребовать некоторого знания принципов работы отдельных протоколов. Хотя в отношении Tcp и Udp все относительно просто.

Общий принцип работы сокетов

При работе с сокетами вне зависимости от выбранных протоколов мы будем опираться на методы класса Socket:

- `Accept()`: создает новый объект Socket для обработки входящего подключения
- `Bind()`: связывает объект Socket с локальной конечной точкой
- `Close()`: закрывает сокет
- `Connect()`: устанавливает соединение с удаленным хостом
- `Listen()`: начинает прослушивание входящих запросов
- `Poll()`: определяет состояние сокета
- `Receive()`: получает данные
- `Send()`: отправляет данные
- `Shutdown()`: блокирует на сокете прием и отправку данных

В зависимости от применяемого протокола (TCP, UDP и т.д.) общий принцип работы с сокетами будет немного различаться.

При применении протокола, который требует установление соединения, например, TCP, сервер должен вызвать метод Bind для установки точки для прослушивания входящих подключений и затем запустить прослушивание подключений с помощью метода Listen. Далее с помощью метода Accept можно получить входящие запросы на подключение в виде объекта Socket, который используется для взаимодействия с удаленным узла. У полученного объекта Socket вызываются методы Send и Receive соответственно для отправки и получения данных. Если необходимо подключиться к серверу, то вызывается метод Connect. Для обмена данными с сервером также применяются методы Send или Receive.

Если применяется протокол, для которого не требуется установление соединения, например, UDP, то после вызова метода Bind не надо вызывать метод Listen. И в этом случае для приема данных используется метод ReceiveFrom, а для отправки данных - метод SendTo.

Клиент-серверное приложение на сокетах TCP

Рассмотрим, как создать сервер, работающий по протоколу TCP, с помощью сокетов. Общая схема работы серверного сокета TCP будет следующей:

Код программы сервера будет таким:

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

namespace SocketTcpServer
{
    class Program
    {
        static int port = 8005; // порт для приема входящих запросов
        static void Main(string[] args)
        {
            // получаем адреса для запуска сокета
            IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), port);

            // создаем сокет
            Socket listenSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
```

```
ProtocolType.Tcp);
            try
            {
                // связываем сокет с локальной точкой, по которой будем принимать данные
                listenSocket.Bind(ipPoint);

                // начинаем прослушивание
                listenSocket.Listen(10);

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

                while (true)
                {
                    Socket handler = listenSocket.Accept();
                    // получаем сообщение
                    StringBuilder builder = new StringBuilder();
                    int bytes = 0; // количество полученных байтов
                    byte[] data = new byte[256]; // буфер для получаемых данных

                    do
                    {
                        bytes = handler.Receive(data);
                        builder.Append(Encoding.Unicode.GetString(data, 0, bytes));
                    }
                    while (handler.Available>0);

                    Console.WriteLine(DateTime.Now.ToShortTimeString() + ": " +
builder.ToString());

                    // отправляем ответ
                    string message = "ваше сообщение доставлено";
                    data = Encoding.Unicode.GetBytes(message);
                    handler.Send(data);
                    // закрываем сокет
                    handler.Shutdown(SocketShutdown.Both);
                    handler.Close();
                }
            }
            catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

Вначале после создания сокета связываем его с локальной точкой методом Bind:

```
listenSocket.Bind(ipPoint);
```

Сокет будет прослушивать подключения по 8005 порту на локальном адресе 127.0.0.1. То есть клиент должен будет подключаться к локальному адресу и порту 8005.

Далее начинаем прослушивание:

```
listenSocket.Listen(10);
```

Метод Listen вызывается только после метода Bind. В качестве параметра он принимает количество входящих подключений, которые могут быть поставлены в очередь сокета.

После вызова метода Listen начинается прослушивание входящих подключений, и если подключения приходят на сокет, то их можно получить с помощью метода Accept:

```
Socket handler = listenSocket.Accept();
```

Метод Accept извлекает из очереди ожидающих запрос первый запрос и создает для его обработки объект Socket. Если очередь запросов пуста, то метод Accept блокирует вызывающий поток до появления нового подключения.

Для обработки запроса сначала в цикле do..while получаем данные с помощью метода Receive:

```
do
{
    bytes = handler.Receive(data);
    builder.Append(Encoding.Unicode.GetString(data, 0, bytes));
}
while (handler.Available > 0);
```

Метод Receive в качестве параметра принимает массив байтов, в который считываются полученные данные, и возвращает количество полученных байтов.

Если отсутствуют данные, доступные для чтения, метод Receive блокирует вызывающий поток до тех пор, пока не станут доступны данные, если не было установлено значение тайм-аута путем использования объекта Socket.ReceiveTimeout. Если значение тайм-аута было превышено, объект Receive выдаст исключение SocketException. Чтобы отследить наличие данных в потоке, используется свойство **Available**.

После получения данных клиенту посылается ответное сообщение с помощью метода Send, который в качестве параметра принимает массив байтов:

```
handler.Send(data);
```

В конце обработки запроса надо закрыть связанный с ним сокет:

```
handler.Shutdown(SocketShutdown.Both);
handler.Close();
```

Вызов метода **Shutdown()** перед закрытием сокета гарантирует, что не останется никаких необработанных данных. Этот метод в качестве параметра принимает значение из перечисления **SocketShutdown**:

- `Both`: остановка как отправки, так и получения данных сокетом
- `Receive`: остановка получения данных
- `Send`: остановка отправки данных

Клиент

Теперь добавим проект для клиента. Общая схема работы клиента на сокетах будет немного отличаться:

Полный код клиента:

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

namespace SocketTcpClient
{
    class Program
    {
        // адрес и порт сервера, к которому будем подключаться
        static int port = 8005; // порт сервера
        static string address = "127.0.0.1"; // адрес сервера
        static void Main(string[] args)
        {
            try
            {
                IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse(address), port);

                Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
                // подключаемся к удаленному хосту
                socket.Connect(ipPoint);
                Console.Write("Введите сообщение:");
                string message = Console.ReadLine();
                byte[] data = Encoding.Unicode.GetBytes(message);
                socket.Send(data);

                // получаем ответ
               data = new byte[256]; // буфер для ответа
              StringBuilder builder = new StringBuilder();
              int bytes = 0; // количество полученных байт

               do
```

```
                {
                    bytes = socket.Receive(data, data.Length, 0);
                    builder.Append(Encoding.Unicode.GetString(data, 0, bytes));
                }
                while (socket.Available > 0);
                Console.WriteLine("ответ сервера: " + builder.ToString());

                // закрываем сокет
                socket.Shutdown(SocketShutdown.Both);
                socket.Close();
            }
            catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
                        Console.Read();
        }
    }
}
```

Для клиента характерно все то же самое, только теперь после создания сокета вызывается метод **Connect()**, в который передается адрес сервера:

```
IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse(address), port);
socket.Connect(ipPoint);
```

Теперь запустим программы сервера и клиента. Консоль клиента:

```
Введите сообщение: привет мир
ответ сервера: ваше сообщение
доставлено
```

Консоль сервера:

```
Сервер запущен. Ожидание
подключений...
22:34: привет мир
```

Использование сокетов для работы с UDP

Протокол UDP не требует установки постоянного подключения, и, возможно, многим покажется легче работать с UDP, чем с TCP. Большинство принципов при работе с UDP те же, что и с TCP.

Вначале создается сокет:

```
Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
```

Если сокет должен получать сообщения, то надо привязать его к локальному адресу и одному из портов с помощью метода Bind:

```
IPEndPoint localIP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 5555);
socket.Bind(localIP);
```

После этого можно отправлять и получать сообщения. Для получения сообщений используется метод **ReceiveFrom()**:

```
byte[] data = new byte[256]; // буфер для получаемых данных

//адрес, с которого пришли данные
EndPoint remoteIp = new IPEndPoint(IPAddress.Any, 0);
int bytes = socket.ReceiveFrom(data, ref remoteIp);
```

В качестве параметра в метод передается массив байтов, в который надо считать данные, и удаленная точка, с которой приходят эти данные. Метод возвращает количество считанных байтов.

Для отправки данных используется метод **SendTo()**:

```
string message = Console.ReadLine();
byte[] data = Encoding.Unicode.GetBytes(message);
EndPoint remotePoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), remotePort);
listeningSocket.SendTo(data, remotePoint);
```

В метод передается массив отправляемых данных, а также адрес, по которому эти данные надо отправить.

Создадим программу UDP-клиента:

```
using System;
using System.Text;
using System.Threading.Tasks;
using System.Net;
using System.Net.Sockets;

namespace SocketUdpClient
{
    class Program
    {
        static int localPort; // порт приема сообщений
```

```
        static int remotePort; // порт для отправки сообщений
        static Socket listeningSocket;

        static void Main(string[] args)
        {
            Console.Write("Введите порт для приема сообщений: ");
            localPort = Int32.Parse(Console.ReadLine());
            Console.Write("Введите порт для отправки сообщений: ");
            remotePort = Int32.Parse(Console.ReadLine());
            Console.WriteLine("Для отправки сообщений введите сообщение и нажмите Enter");
            Console.WriteLine();

            try
            {
                listeningSocket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
ProtocolType.Udp);
                Task listeningTask = new Task(Listen);
                listeningTask.Start();

                // отправка сообщений на разные порты
                while (true)
                {
                    string message = Console.ReadLine();

                    byte[] data = Encoding.Unicode.GetBytes(message);
                    EndPoint remotePoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"),
remotePort);
                    listeningSocket.SendTo(data, remotePoint);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                Close();
            }
        }

        // поток для приема подключений
        private static void Listen()
        {
            try
            {
                //Прослушиваем по адресу
                IPEndPoint localIP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), localPort);
                listeningSocket.Bind(localIP);

                while (true)
               {
               // получаем сообщение
```

```
                    StringBuilder builder = new StringBuilder();
                    int bytes = 0; // количество полученных байтов
                    byte[] data = new byte[256]; // буфер для получаемых данных

                    //адрес, с которого пришли данные
                    EndPoint remoteIp = new IPEndPoint(IPAddress.Any, 0);

                    do
                    {
                        bytes = listeningSocket.ReceiveFrom(data, ref remoteIp);
                        builder.Append(Encoding.Unicode.GetString(data, 0, bytes));
                    }
                    while (listeningSocket.Available > 0);
                    // получаем данные о подключении
                    IPEndPoint remoteFullIp = remoteIp as IPEndPoint;

                    // выводим сообщение
                    Console.WriteLine("{0}:{1} - {2}", remoteFullIp.Address.ToString(),

remoteFullIp.Port, builder.ToString());
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                Close();
            }
        }
                // закрытие сокета
        private static void Close()
        {
            if (listeningSocket != null)
            {
                listeningSocket.Shutdown(SocketShutdown.Both);
                listeningSocket.Close();
                listeningSocket = null;
            }
        }
    }
}
```

Вначале пользователь вводит порты для приема данных и для отправки. Предполагается, что два приложения клиента, которые будут между собой взаимодействовать, запущены на одной локальной машине. Если адреса клиентов различаются, то можно предусмотреть и ввода адреса для отправки данных.

После ввода портов запускается задача на прослушивание входящих сообщений. В отличие от tcp-сервера здесь не надо вызывать методы Listen и Accept. В бесконечном цикле мы напрямую можем

получить получение данные с помощью метода `ReceiveFrom()`, который блокирует вызывающий поток, пока не придет очередная порция данных.

Этот метод возвращает через ref-параметр удаленную точку, с которой получены данные:

```
IPEndPoint remoteFullIp = remoteIp as IPEndPoint;
```

То есть, не смотря на то, что в данном случае прием и отправка сообщений разграничены и текущий клиент отправляет данные только на введенный вначале порт, но мы вполне можем добавить возможность ответа на сообщения, используя данные полученной удаленной точки (адрес и порт).

В главном потоке происходит отправка сообщений с помощью метода **SendTo()**

Таким образом, приложение сразу осуществляет функции и сервера, и клиента.

Теперь запустим две копии приложения и введем разные данные для портов. Первый клиент:

```
Введите порт для приема сообщений: 4004
Введите порт для отправки сообщений:
4005
Для отправки сообщений введите
сообщение и нажмите Enter

127.0.0.1:4005 - привет порт 4004
добрый день, порт 4005
чудная погодка
```

Второй клиент:

```
Введите порт для приема сообщений: 4005
Введите порт для отправки сообщений:
4004
Для отправки сообщений введите
сообщение и нажмите Enter

привет порт 4004
127.0.0.1:4004 - добрый день, порт 4005
127.0.0.1:4004 - чудная погодка
```

Протокол TCP

Огромное количество трафика и взаимодействия в сети сейчас происходит по протоколу TCP (Transmission Control Protocol). Этот протокол гарантирует доставку сообщений и широко используется в различных существующих на сегодняшний день программах.

Для работы с протоколом TCP в .NET предназначены классы **TcpClient** и **TcpListener**. Эти классы строятся поверх класса **System.Net.Sockets.Socket**.

TCP-клиент. Класс TcpClient

Для создания клиентской программы, работающей по протоколу TCP, предназначен класс **TcpClient**.

Для подключения к серверу TCP, в этом классе определен метод `Connect()`, которому передается название хоста и порт:

```
TcpClient tcpClient = new TcpClient ();
tcpClient.Connect ("www.microsoft.com", 80);
```

Чтобы взаимодействовать с сервером TcpClient определяет метод `GetStream()`, который возвращает объект **NetworkStream**. Через данный объект можно передавать сообщения серверу или, наоборот, получать данные с сервера.

```
NetworkStream netStream = tcpClient.GetStream();
```

После окончания работы с TcpClient его надо закрыть методом `Close()`.

```
tcpClient.Close();
```

Для отправки данных у потока используется метод `Write()`:

```
NetworkStream stream = tcpClient.GetStream();
string response = "Hello world!"
byte[] data = System.Text.Encoding.UTF8.GetBytes(response);
stream.Write(data, 0, data.Length);
```

Для чтения используется метод `Read()` класса NetworkStream:

```
NetworkStream stream = tcpClient.GetStream();
byte[] data = new byte[256];
int bytes = stream.Read(data, 0, data.Length); // получаем количество считанных байтов
string message = Encoding.UTF8.GetString(data, 0, bytes)
```

При этом в обоих случаях нам надо учитывать тип кодировки: он должен совпадать с кодировкой, в которой сервер отправляет или получает сообщения. В данном случае используется кодировка UTF8.

Полноценный код клиента, выполняющий чтение из потока, мог бы выглядеть так:

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

namespace TcpClientApp
{
    class Program
    {
        private const int port = 8888;
        private const string server = "127.0.0.1";

        static void Main(string[] args)
        {
            try
            {
                TcpClient client = new TcpClient();
                client.Connect(server, port);

                byte[] data = new byte[256];
                StringBuilder response = new StringBuilder();
                NetworkStream stream = client.GetStream();

                do
                {
                    int bytes = stream.Read(data, 0, data.Length);
                    response.Append(Encoding.UTF8.GetString(data, 0, bytes));
                }
                while (stream.DataAvailable); // пока данные есть в потоке

                Console.WriteLine(response.ToString());

                // Закрываем потоки
                stream.Close();
                client.Close();
            }
            catch (SocketException e)
            {
                Console.WriteLine("SocketException: {0}", e);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception: {0}", e.Message);
            }

            Console.WriteLine("Запрос завершен...");
            Console.Read();
        }
    }
}
```

В данном случае предполагается, что по локальному адресу 127.0.0.1 на порту 8888 запущен tcp-сервер.

Мы ожидаем, что сервер будет передавать строку в кодировке UTF8, и для создания строки определяется объект StringBuilder.

Для считывания данных создаем массив байтов: `byte[] data = new byte[256]`. Однако поскольку данные могут иметь больший размер, то считываем их из потока в цикле do..while. При этом проверяется значение `stream.DataAvailable` на наличие данных в потоке.

TCP-сервер. Класс TcpListener

Класс TcpListener прослушивает входящие подключения по определенному порту.

Для создания объекта слушателя надо передать ему в конструктор объект IPAddress и порт, по которому будет вестись прослушивание входящих подключений:

```
IPAddress localAddr = IPAddress.Parse("127.0.0.1");
int port = 8888;
TcpListener server = new TcpListener(localAddr, port);
server.Start();
```

Для запуска сервера вызывается метод **Start()**.

Когда к серверу обращается клиент, то мы можем использовать один из двух методов **AcceptSocket** или **AcceptTcpClient** для получения соответственно объекта **Socket** или **TcpClient**, которые будут использоваться для взаимодействия с подключенным клиентом.

```
TcpClient client = server.AcceptTcpClient();
// или
Socket socket = server.AcceptSocket();
```

Методы AcceptSocket и AcceptTcpClient блокируют выполняющий поток, пока сервер не обслужит подключенного клиента. Затем через методы, определенные в классах TcpClient и Socket, можно взаимодействовать с подключенным клиентом: получать от него данные или, наоборот, отправлять ему.

В конце по завершению работы сервера надо вызвать метод **Stop()**.

В прошлой теме был создан TCP-клиент. Теперь создадим примитивный TCP-сервер, к которому будет подключаться клиент:

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

namespace TcpListenerApp
{
    class Program
    {
        const int port = 8888; // порт для прослушивания подключений
        static void Main(string[] args)
        {
            TcpListener server=null;
            try
            {
                IPAddress localAddr = IPAddress.Parse("127.0.0.1");
```

```
                server = new TcpListener(localAddr, port);

                // запуск слушателя
                server.Start();

                while (true)
                {
                    Console.WriteLine("Ожидание подключений... ");

                    // получаем входящее подключение
                    TcpClient client = server.AcceptTcpClient();
                    Console.WriteLine("Подключен клиент. Выполнение запроса...");

                    // получаем сетевой поток для чтения и записи
                    NetworkStream stream = client.GetStream();

                    // сообщение для отправки клиенту
                    string response = "Привет мир";
                                        // преобразуем сообщение в массив байтов
                    byte[] data = Encoding.UTF8.GetBytes(response);

                    // отправка сообщения
                    stream.Write(data, 0, data.Length);
                    Console.WriteLine("Отправлено сообщение: {0}", response);
                    // закрываем подключение
                    client.Close();
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (server != null)
                    server.Stop();
            }
        }
    }
}
```

В данном случае сервер прослушивает все подключения по порту 8888. В бесконечном цикле он получает входящие подключения.

У входящих объектов TcpClient получает сетевой поток через метод `client.GetStream()` и затем его использует для отправки сообщения.

И мы можем совместить клиента из прошлой темы с этим сервером. Тогда сервер будет отправлять клиентам данные, а клиенты - получать их.

Многопоточное клиент-серверное приложение TCP

Рассмотрим, как создать многопоточное клиент-серверное приложение. Фактически оно будет отличаться от однопоточного только тем, что обработка запроса клиента будет вынесена в отдельный поток.

Вначале создадим проект для клиента. Назовем проект ConsoleClient и в классе Program определим следующий код:

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

namespace ConsoleClient
{
    class Program
    {
        const int port = 8888;
        const string address = "127.0.0.1";
        static void Main(string[] args)
        {
            Console.Write("Введите свое имя:");
            string userName = Console.ReadLine();
            TcpClient client = null;
            try
            {
                client = new TcpClient(address, port);
                NetworkStream stream = client.GetStream();

                while (true)
                {
                    Console.Write(userName + ": ");
                    // ввод сообщения
                    string message = Console.ReadLine();
                    message = String.Format("{0}: {1}", userName, message);
                    // преобразуем сообщение в массив байтов
                    byte[] data = Encoding.Unicode.GetBytes(message);
                    // отправка сообщения
                    stream.Write(data, 0, data.Length);

                    // получаем ответ
                    data = new byte[64]; // буфер для получаемых данных
                    StringBuilder builder = new StringBuilder();
                    int bytes = 0;
                    do
                    {
                        bytes = stream.Read(data, 0, data.Length);
```

```
                    builder.Append(Encoding.Unicode.GetString(data, 0, bytes));
                }
                while (stream.DataAvailable);

                message = builder.ToString();
                Console.WriteLine("Сервер: {0}", message);
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            client.Close();
        }
    }
}
```

В программе клиента пользователь будет вначале вводить свое имя, а затем сообщение для отправки. Причем сообщение будет уходить в формате `Имя: сообщение`.

После отправки сообщения клиент получает сообщение с сервера.

Теперь создадим проект сервера, который назовем ConsoleServer. Вначале в проект сервера добавим новый класс **ClientObject**, который будет представлять отдельное подключение:

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

namespace ConsoleServer
{
    public class ClientObject
    {
        public TcpClient client;
        public ClientObject(TcpClient tcpClient)
        {
            client = tcpClient;
        }

        public void Process()
        {
            NetworkStream stream = null;
            try
            {
                stream = client.GetStream();
                byte[] data = new byte[64]; // буфер для получаемых данных
                while (true)
                {
                    // получаем сообщение
                    StringBuilder builder = new StringBuilder();
```

```
                    int bytes = 0;
                    do
                    {
                        bytes = stream.Read(data, 0, data.Length);
                        builder.Append(Encoding.Unicode.GetString(data, 0, bytes));
                    }
                    while (stream.DataAvailable);

                    string message = builder.ToString();

                    Console.WriteLine(message);
                    // отправляем обратно сообщение в верхнем регистре
                    message = message.Substring(message.IndexOf(':') + 1).Trim().ToUpper();
                    data = Encoding.Unicode.GetBytes(message);
                    stream.Write(data, 0, data.Length);
                }
            }
            catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                if (stream != null)
                    stream.Close();
                if (client != null)
                    client.Close();
            }
        }
    }
}
```

В этом классе, наоборот, сначала получаем сообщение в цикле do..while и потом немного его изменяем (отрезаем по двоеточию и переводим в верхний регистр) и отправляем обратно клиенту. То есть класс ClientObject заключает в себе все действия по работе с отдельным подключением.

В главном классе проекта сервера определим следующий код:

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

namespace ConsoleServer
{
    class Program
    {
        const int port = 8888;
        static TcpListener listener;
        static void Main(string[] args)
        {
             try
              {
```

```
                listener = new TcpListener(IPAddress.Parse("127.0.0.1"), port);
                listener.Start();
                Console.WriteLine("Ожидание подключений...");

                while(true)
                {
                    TcpClient client = listener.AcceptTcpClient();
                    ClientObject clientObject = new ClientObject(client);

                    // создаем новый поток для обслуживания нового клиента
                    Thread clientThread = new Thread(new ThreadStart(clientObject.Process));
                    clientThread.Start();
                }
            }
            catch(Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                if(listener!=null)
                    listener.Stop();
            }
        }
    }
}
```

Сразу после подключения нового клиента:

```
TcpClient client = listener.AcceptTcpClient()
```

Создается объект ClientObject и новый поток, который запускает метод Process объекта ClientObject, где собственно и происходит получение и отправка сообщений. Таким образом, сервер сможет одновременно обрабатывать сразу несколько запросов.

Результаты работы программы. Один из клиентов:

```
Введите свое имя: Евгений
Евгений: привет мир
Сервер: ПРИВЕТ МИР
Евгений: пока мир Сервер:
ПОКА МИР Евгений: _
```

Сервер:

```
Ожидание подключений...
Евгений: привет мир
Евгений: пока мир
Том: привет чат
```

Консольный TCP-чат

Напишем более менее осмысленную программу - консольный tcp-чат.

Вначале создадим консольный проект сервера, который назовем ChatServer. В этот проект добавим два новых класса **ClientObject** и **ServerObject**. Класс ClientObject будет выглядеть так:

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

namespace ChatServer
{
    public class ClientObject
    {
        protected internal string Id { get; private set; }
        protected internal NetworkStream Stream {get; private set;}
        string userName;
        TcpClient client;
        ServerObject server; // объект сервера

        public ClientObject(TcpClient tcpClient, ServerObject serverObject)
        {
            Id = Guid.NewGuid().ToString();
            client = tcpClient;
            server = serverObject;
            serverObject.AddConnection(this);
        }

        public void Process()
        {
            try
            {
                Stream = client.GetStream();
                // получаем имя пользователя
                string message = GetMessage();
                userName = message;

                message = userName + " вошел в чат";
                // посылаем сообщение о входе в чат всем подключенным пользователям
                server.BroadcastMessage(message, this.Id);
                Console.WriteLine(message);
                // в бесконечном цикле получаем сообщения от клиента
                while (true)
                {
                    try
                    {
```

```
                    message = GetMessage();
                    message = String.Format("{0}: {1}", userName, message);
                    Console.WriteLine(message);
                    server.BroadcastMessage(message, this.Id);
                }
                catch
                {
                    message = String.Format("{0}: покинул чат", userName);
                    Console.WriteLine(message);
                    server.BroadcastMessage(message, this.Id);
                    break;
                }
            }
        }
        catch(Exception e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            // в случае выхода из цикла закрываем ресурсы
            server.RemoveConnection(this.Id);
            Close();
        }
    }

    // чтение входящего сообщения и преобразование в строку
    private string GetMessage()
    {
        byte[] data = new byte[64]; // буфер для получаемых данных
        StringBuilder builder = new StringBuilder();
        int bytes = 0;
        do
        {
            bytes = Stream.Read(data, 0, data.Length);
            builder.Append(Encoding.Unicode.GetString(data, 0, bytes));
        }
        while (Stream.DataAvailable);

        return builder.ToString();
    }

    // закрытие подключения
    protected internal void Close()
    {
        if (Stream != null)
            Stream.Close();
        if (client != null)
            client.Close();
    }
  }
}
```

У объекта ClientObject будет устанавливаться свойство Id, которое будет уникально его идентифицировать, и свойство Stream, хранящее поток для взаимодействия с клиентом. При создании нового объекта в конструкторе будет происходить его добавление в коллекцию подключений класса ServerObject, который мы далее создадим:

```
serverObject.AddConnection(this);
```

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 `BroadcastMessage()` класса ServerObject.

Класс ServerObject будет выглядеть таким образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Sockets;
using System.Net;
using System.Text;
using System.Threading;

namespace ChatServer
{
    public class ServerObject
    {
        static TcpListener tcpListener; // сервер для прослушивания
        List<ClientObject> clients = new List<ClientObject>(); // все подключения

        protected internal void AddConnection(ClientObject clientObject)
        {
            clients.Add(clientObject);
        }
        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);
        }
                // прослушивание входящих подключений
        protected internal void Listen()
        {
            try
            {
                tcpListener = new TcpListener(IPAddress.Any, 8888);
                tcpListener.Start();
                Console.WriteLine("Сервер запущен. Ожидание подключений...");

                while (true)
               {
                      TcpClient tcpClient = tcpListener.AcceptTcpClient();
```

```
                ClientObject clientObject = new ClientObject(tcpClient, this);
                Thread clientThread = new Thread(new ThreadStart(clientObject.Process));
                clientThread.Start();
            }
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex.Message);
            Disconnect();
        }
    }

    // трансляция сообщения подключенным клиентам
    protected internal void BroadcastMessage(string message, string id)
    {
        byte[] data = Encoding.Unicode.GetBytes(message);
        for (int i = 0; i < clients.Count; i++)
        {
            if (clients[i].Id!= id) // если id клиента не равно id отправляющего
            {
                clients[i].Stream.Write(data, 0, data.Length); //передача данных
            }
        }
    }
    // отключение всех клиентов
    protected internal void Disconnect()
    {
        tcpListener.Stop(); //остановка сервера

        for (int i = 0; i < clients.Count; i++)
        {
            clients[i].Close(); //отключение клиента
        }
        Environment.Exit(0); //завершение процесса
    }
  }
}
```

Все подключенные клиенты будут храниться в коллекции clients. С помощью методов AddConnection и RemoveConnection мы можем управлять добавлением / удалением объектов из этой коллекции.

Основной метод - `Listen()`, в котором будет осуществляться прослушивание всех входящих подключений. При получении подключения будет запускаться новый поток, в котором будет выполняться метод Process объекта ClientObject.

Для передачи сообщений всем клиентам, кроме отправившего, предназначен метод `BroadcastMessage()`.

Таким образом разделяются сущность подключенного клиента и сущность сервера.

Теперь надо запустить прослушивание в основном классе программы. Для этого изменим класс Program:

```
using System;
using System.Threading;

namespace ChatServer
{
    class Program
    {
        static ServerObject server; // сервер
        static Thread listenThread; // потока для прослушивания
        static void Main(string[] args)
        {
            try
            {
                server = new ServerObject();
                listenThread = new Thread(new ThreadStart(server.Listen));
                listenThread.Start(); //старт потока
            }
            catch (Exception ex)
            {
                server.Disconnect();
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

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

Теперь создадим новый консольный проект для клиента, который назовем ChatClient. Изменим его стандартный класс Program следующим образом:

```
using System;
using System.Threading;
using System.Net.Sockets;
using System.Text;

namespace ChatClient
{
    class Program
    {
        static string userName;
        private const string host = "127.0.0.1";
        private const int port = 8888;
        static TcpClient client;
        static NetworkStream stream;

        static void Main(string[] args)
        {
            Console.Write("Введите свое имя: ");
            userName = Console.ReadLine();
            client = new TcpClient();
            try
```

```
    {
        client.Connect(host, port); //подключение клиента
        stream = client.GetStream(); // получаем поток

        string message = userName;
        byte[] data = Encoding.Unicode.GetBytes(message);
        stream.Write(data, 0, data.Length);

        // запускаем новый поток для получения данных
        Thread receiveThread = new Thread(new ThreadStart(ReceiveMessage));
        receiveThread.Start(); //старт потока
        Console.WriteLine("Добро пожаловать, {0}", userName);
        SendMessage();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        Disconnect();
    }
}
        // отправка сообщений
static void SendMessage()
{
    Console.WriteLine("Введите сообщение: ");

    while (true)
    {
        string message = Console.ReadLine();
        byte[] data = Encoding.Unicode.GetBytes(message);
        stream.Write(data, 0, data.Length);
    }
}
        // получение сообщений
static void ReceiveMessage()
{
    while (true)
    {
        try
        {
            byte[] data = new byte[64]; // буфер для получаемых данных
            StringBuilder builder = new StringBuilder();
            int bytes = 0;
            do
            {
                bytes = stream.Read(data, 0, data.Length);
                builder.Append(Encoding.Unicode.GetString(data, 0, bytes));
            }
            while (stream.DataAvailable);
```

```
                string message = builder.ToString();
                Console.WriteLine(message);//вывод сообщения
            }
            catch
            {
                Console.WriteLine("Подключение прервано!"); //соединение было прервано
                Console.ReadLine();
                Disconnect();
            }
        }
    }

    static void Disconnect()
    {
        if(stream!=null)
            stream.Close();//отключение потока
        if(client!=null)
            client.Close();//отключение клиента
        Environment.Exit(0); //завершение процесса
    }
  }
}
```

Чтобы не блокировать ввод сообщений в главном потоке, для получения сообщений создается новый поток, который обращается к методу ReceiveMessage.

Работа одного из клиентов:

```
Введите свое имя: Евгений
Добро пожаловать, Евгений
Введите сообщение:
Олег вошел в чат
привет мир
Олег: привет чат
```

Работа сервера:

```
Сервер запущен. Ожидание
подключений...
Евгений вошел в чат
Олег вошел в чат
Евгений: привет мир
Олег: привет чат
```

Протокол UDP

UDP (User Datagram Protocol) представляет сетевой протокол, который позволяет доставить данные на удаленный узел. Для этого передачи сообщений по протоколу UDP нет надобности использовать сервер, данные напрямую передаются от одного узла к другому. Снижаются накладные расходы при передаче, по сравнению с TCP, сами данные передаются быстрее. Все посылаемые сообщения по протоколу UDP называются дейтаграммами. Также через UDP можно передавать широковещательные сообщения для для набора адресов в подсети.

В то же время UDP имеет недостатки по сравнению с TCP. В частности, UDP не гарантирует доставку дейтаграммы конечному адресу. Поэтому данный протокол подходит больше для передачи изображений, мультимедийных файлов, потокового аудио, видео и т.п., когда недостаток небольшого количества потерянных при передаче дейтаграмм не сильно скажется на качестве переданных данных.

UdpClient

В .NET за работу с протоколом UDP отвечает класс **UdpClient**, который построен на основе класса Socket.

Для создания подключения достаточно создать объект UdpClient и вызывать у него метод `Connect()`:

```
UdpClient client = new UdpClient();
client.Connect("www.microsoft.com", 8888);
```

В метод Connect передается адрес и порт для подключения, то есть настройки внешнего адреса, к которому мы хотим подключиться. В случае с локальным адресом можно указывать один порт. При этом Connect не устанавливает постоянного соединения с удаленным хостом. Он только устанавливает параметры подключения, которые также можно задать с помощью конструктора:

```
UdpClient client = new UdpClient("www.microsoft.com", 8888);
```

Фактически образование соединения происходит при передаче данных на удаленный хост или, наоборот, при получении с него данных. Для передачи данных применяется метод **Send()**, который в качестве параметра принимает массив отправляемых байтов и количество байтов, которые надо отправить:

```
UdpClient client = new UdpClient(8001);
string message = "Hello world!";
byte[] data = Encoding.UTF8.GetBytes(message);
int numberOfSentBytes = client.Send(data, data.Length);
client.Close();
```

Метод `Send()` возвращает количество реально отправленных байтов, благодаря чему мы можем сравнить, сколько из отправляемых байтов в реальности было отправлено.

После завершения работы объекта UdpClient его надо закрыть с помощью метода `Close()`.

Метод `Send()` имеет ряд перегруженных версий, благодаря чему мы можем указать адрес хоста и порт прямо при отправке:

```
UdpClient client = new UdpClient();
string message = "Hello world!";
byte[] data = Encoding.UTF8.GetBytes(message);
client.Send(data, data.Length, "127.0.0.1", 8001);
client.Close();
```

Для получения данных применяется метод **Receive()**. Этот метод принимает один параметр типа **System.Net.IPEndPoint** с модификатором **ref**:

```
UdpClient client = new UdpClient(8001);
IPEndPoint ip = null;
byte[] data = client.Receive(ref ip);
string message = Encoding.UTF8.GetString(data);
client.Close();
```

Объект IPEndPoint представляет удаленную точку, с которой поступили данные. Полученные данные возвращает метод Receive в виде массива байтов.

Как правило, рекомендуется выносить вызов метода Receive в отдельный поток, поскольку данный метод блокирует вызывающий поток пока данные не будут получены.

При этом если мы указали информацию о подключении (название хоста, номер порта) в конструкторе или в методе Connect, то метод Receive будет получать данные только с указанной удаленной точки, остальные подключения будут игнорироваться. Если же мы использовали пустой конструктор и не вызывали метод Connect, то UdpClient будет принимать данные от всех подключений.

Создадим примитивный консольный udp-чат:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;

namespace UdpClientApp
{
    class Program
    {
        static string remoteAddress; // хост для отправки данных
        static int remotePort; // порт для отправки данных
        static int localPort; // локальный порт для прослушивания входящих подключений

        static void Main(string[] args)
        {
            try
            {
                Console.Write("Введите порт для прослушивания: "); // локальный порт
                localPort = Int32.Parse(Console.ReadLine());
                Console.Write("Введите удаленный адрес для подключения: ");
                remoteAddress = Console.ReadLine(); // адрес, к которому мы подключаемся
                Console.Write("Введите порт для подключения: ");
                remotePort = Int32.Parse(Console.ReadLine()); // порт, к которому мы подключаемся
```

```
        Thread receiveThread = new Thread(new ThreadStart(ReceiveMessage));
        receiveThread.Start();
        SendMessage(); // отправляем сообщение
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
private static void SendMessage()
{
    UdpClient sender = new UdpClient(); // создаем UdpClient для отправки сообщений
    try
    {
        while(true)
        {
            string message = Console.ReadLine(); // сообщение для отправки
            byte[] data = Encoding.Unicode.GetBytes(message);
            sender.Send(data, data.Length, remoteAddress, remotePort); // отправка
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        sender.Close();
    }
}

private static void ReceiveMessage()
{
    UdpClient receiver = new UdpClient(localPort); // UdpClient для получения данных
    IPEndPoint remoteIp = null; // адрес входящего подключения
    try
    {
        while(true)
        {
            byte[] data = receiver.Receive(ref remoteIp); // получаем данные
            string message = Encoding.Unicode.GetString(data);
            Console.WriteLine("Собеседник: {0}", message);
        }
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        receiver.Close();
    }
```

```
        }
    }
}
```

Программа определяет три переменных, значения для которых вводятся в самом начале: localPort (порт, по которому UdpClient будет прослушивать входящие подключения), remoteAddress (внешний адрес, к которому подключаемся и на который будем отправлять сообщения), remotePort (внешний порт, на который будут отправляться сообщения).

После ввода этих данных запускается новый поток, в котором запускается метод ReceiveMessage и в котором будет идти прослушивание локального порта и получение от входящих подключений сообщений.

Параллельно в главном потоке будет срабатывать метод **SendMessage()**, в котором будет идти отправка введенных сообщений на внешний удаленный адрес и порт.

Построим проект и запустим сразу две копии приложения. В каждой копии для внешнего адреса введем адрес "127.0.0.1", так как мы будем подключаться к локальному адресу. Но в первой копии для портов введем 8001 и 8002, то есть будет идти прослушивание порта 8001, а данные будут отправляться на порт 8002. А во второй копии, наоборот, введем 8002 и 8001, то есть будет идти прослушивание порта 8002, а данные будут отправляться на порт 8001 первой копии приложения.

Работа первой копии:

```
Введите порт для прослушивания: 8001
Введите удаленный адрес для
подключения: 127.0.0.1
Введите порт для подключения: 8002
Собеседник: Привет мир
Я не мир
```

Работа второй копии:

```
Введите порт для прослушивания: 8002
Введите удаленный адрес для
подключения: 127.0.0.1
Введите порт для подключения: 8001
Привет мир
Собеседник: Я не мир
```

Широковещательная рассылка

Протокол UDP позволяет рассылать сообщения с помощью широковещательной групповой рассылки. При такой рассылке клиенту достаточно отправить одно сообщение, и его получат все остальные клиенты, которые подключены к группе. И нет надобности отправлять одно сообщение каждому отдельному клиенту.

При использовании широковещательной передачи надо учитывать, что передача не идет дальше локальных сетей, так как маршрутизаторы подобные рассылки не пропускают.

Ключевым в данном случае является метод **JoinMulticastGroup()**, который позволяет присоединиться клиенту к группе в локальной сети. Благодаря этому объект UdpClient сможет получать дейтаграммы, которые предназначаются всей группы объектов UdpClient.

Для присоединения к группе в метод JoinMulticastGroup передается адрес группы. В качестве адреса может использоваться один из адресов в диапазоне от 224.0.0.0 до 239.255.255.255.

Создадим приложение, которое будет использовать широковещательную рассылку. Если отталкиваться от проекта из прошлой темы, то для поддержки групповой рассылки достаточно добавить в потоке получения данных вызов метода `JoinMulticastGroup()`:

```
private static void ReceiveMessage()
{
    UdpClient receiver = new UdpClient(localPort); // UdpClient для получения данных
    receiver.JoinMulticastGroup(remoteAddress, 50);
    IPEndPoint remoteIp = null;
        //  остальное содержимое метода
}
```

Метод `JoinMulticastGroup()` принимает два параметра: адрес групповой рассылки (должен быть одним и тем же для всей группы клиентов) и число проходов через маршрутизаторы. То есть пока сообщение достигнет получателя, оно может пройти через некоторое число маршрутизаторов, которые направят его по нужному пути. В данном случае число 50 означает, что чтобы пройти от компьютера-отправителя к компьютеру-получателю, дейтаграмма может миновать 50 маршрутизаторов.

На основе Udp-чата из прошлой темы сделаем консольный чат с применением широковещательной рассылки:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;

namespace MulticastApp
{
    class Program
   {
```

```
static IPAddress remoteAddress; // хост для отправки данных
const int remotePort = 8001; // порт для отправки данных
const int localPort = 8001; // локальный порт для прослушивания входящих подключений
static string username;
static void Main(string[] args)
{
    try
    {
        Console.Write("Введите свое имя:");
        username = Console.ReadLine();
        remoteAddress = IPAddress.Parse("235.5.5.11");
        Thread receiveThread = new Thread(new ThreadStart(ReceiveMessage));
        receiveThread.Start();
        SendMessage(); // отправляем сообщение
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}
private static void SendMessage()
{
    UdpClient sender = new UdpClient(); // создаем UdpClient для отправки
    IPEndPoint endPoint = new IPEndPoint(remoteAddress, remotePort);
    try
    {
        while (true)
        {
            string message = Console.ReadLine(); // сообщение для отправки
            message = String.Format("{0}: {1}", username, message);
            byte[] data = Encoding.Unicode.GetBytes(message);
            sender.Send(data, data.Length, endPoint); // отправка
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        sender.Close();
    }
}
private static void ReceiveMessage()
{
  UdpClient receiver = new UdpClient(localPort); // UdpClient для получения
  данных receiver.JoinMulticastGroup(remoteAddress, 20);
  IPEndPoint remoteIp = null;
  string localAddress = LocalIPAddress();
   try
   {
         while (true)
```

```
                {
                    byte[] data = receiver.Receive(ref remoteIp); // получаем данные
                    if (remoteIp.Address.ToString().Equals(localAddress))
                        continue;
                    string message = Encoding.Unicode.GetString(data);
                    Console.WriteLine(message);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                receiver.Close();
            }
        }
        private static string LocalIPAddress()
        {
            string localIP = "";
            IPHostEntry  host = Dns.GetHostEntry(Dns.GetHostName());
            foreach (IPAddress ip in host.AddressList)
            {
                if (ip.AddressFamily == AddressFamily.InterNetwork)
                {
                    localIP = ip.ToString();
                    break;
                }
            }
            return localIP;
        }
    }
}
```

Теперь для групповой рассылки в локальной сети будет использоваться адрес "235.5.5.11", в качестве порта - 8001. Причем поскольку отправленное пользователем сообщение будет транслироваться всем в группе, в том числе и ему самому, то чтобы сообщение не дублировать в консоли, то в данном случае проверяется адрес отправителя с помощью вспомогательного метода LocalIPAddress(). Если адрес отправителя и получателя совпадает, то ему сообщение не выводится.

После построения проекта мы можем запустить скомпилированное приложение на разных машинах в локальной сети.

Первый клиент:

```
Введите свое имя:Евгений
Олег: привет чат
привет Олег
Олег: тест
```

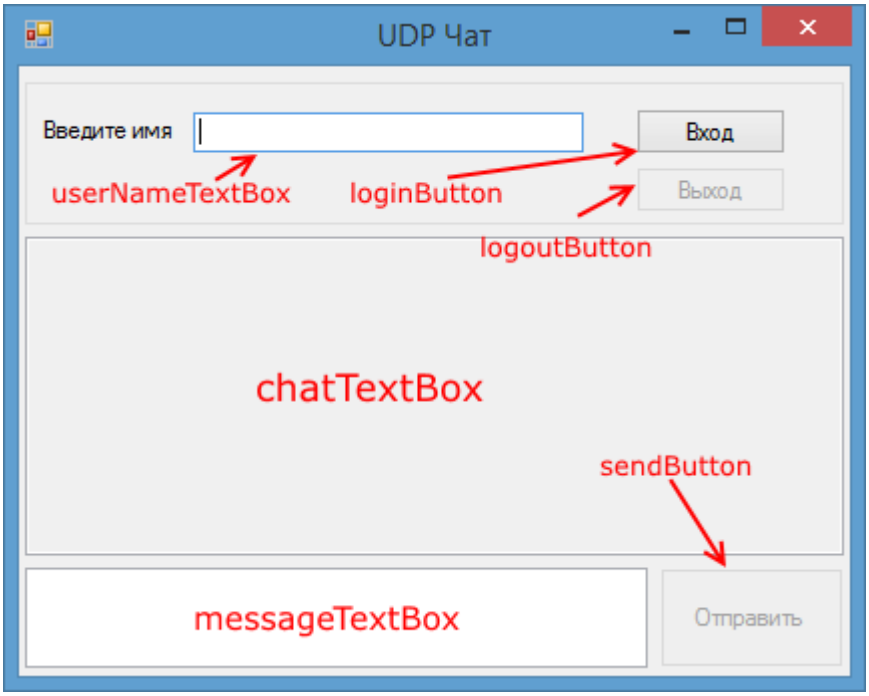
Второй клиент:

```
Введите свое имя:Олег
привет чат
Евгений: привет Олег
тест
```

Чат с широковещательной рассылкой на Windows Forms

Использовать консольные чаты и подобные приложения для работы в сети не очень удобно. Гораздо проще использовать графические программы, поэтому создадим новый проект на Windows Forms, который также будет использовать широковещательную рассылку и который будет называться UdpChat.

Главная и единственная форма приложения будет выглядеть следующим образом:



На форме есть две кнопки loginButton и logoutButton соответственно для входа и выхода из чата, а также кнопка sendButton для отправки сообщений. Для ввода имени, которое будет использоваться в чате, есть текстовое поле userNameTextBox. Для ввода сообщения внизу имеется текстовое поле messageTextBox, которое является многострочным, то есть у которого свойство MultiLine имеет значение true.

И в самом центре размещено многострочное текстовое поле chatTextBox, в которое будут выводиться полученные сообщения.

Весь код формы:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Windows.Forms;
using System.Threading.Tasks;

namespace UdpChat
{
	public partial class Form1 : Form
```

```
{
    bool alive = false; // будет ли работать поток для приема
    UdpClient client;
    const int LOCALPORT = 8001; // порт для приема сообщений
    const int REMOTEPORT = 8001; // порт для отправки сообщений
    const int TTL = 20;
    const string HOST = "235.5.5.1"; // хост для групповой рассылки
    IPAddress groupAddress; // адрес для групповой рассылки

    string userName; // имя пользователя в чате
    public Form1()
    {
        InitializeComponent();

        loginButton.Enabled = true; // кнопка входа
        logoutButton.Enabled = false; // кнопка выхода
        sendButton.Enabled = false; // кнопка отправки
        chatTextBox.ReadOnly = true; // поле для сообщений

        groupAddress = IPAddress.Parse(HOST);
    }
            // обработчик нажатия кнопки loginButton
    private void loginButton_Click(object sender, EventArgs e)
    {
        userName = userNameTextBox.Text;
        userNameTextBox.ReadOnly = true;

        try
        {
            client = new UdpClient(LOCALPORT);
            // присоединяемся к групповой рассылке
            client.JoinMulticastGroup(groupAddress, TTL);

            // запускаем задачу на прием сообщений
            Task receiveTask = new Task(ReceiveMessages);
            receiveTask.Start();

            // отправляем первое сообщение о входе нового пользователя
            string message = userName + " вошел в чат";
            byte[] data = Encoding.Unicode.GetBytes(message);
            client.Send(data, data.Length, HOST, REMOTEPORT);

            loginButton.Enabled = false;
            logoutButton.Enabled = true;
            sendButton.Enabled = true;
        }
        catch(Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
    // метод приема сообщений
```

```
private void ReceiveMessages()
{
    alive = true;
    try
    {
        while (alive)
        {
            IPEndPoint remoteIp = null;
            byte[] data = client.Receive(ref remoteIp);
            string message = Encoding.Unicode.GetString(data);

            // добавляем полученное сообщение в текстовое поле
            this.Invoke(new MethodInvoker(() =>
            {
                string time = DateTime.Now.ToShortTimeString();
                chatTextBox.Text = time + " " + message + "\r\n" + chatTextBox.Text;
            }));
        }
    }
    catch (ObjectDisposedException)
    {
        if (!alive)
            return;
        throw;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
        // обработчик нажатия кнопки sendButton
private void sendButton_Click(object sender, EventArgs e)
{
    try
    {
        string message = String.Format("{0}: {1}", userName, messageTextBox.Text);
        byte[] data = Encoding.Unicode.GetBytes(message);
        client.Send(data, data.Length, HOST, REMOTEPORT);
        messageTextBox.Clear();
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
        // обработчик нажатия кнопки logoutButton
private void logoutButton_Click(object sender, EventArgs e)
{
    ExitChat();
}
// выход из чата
 private void ExitChat()
```

```
        {
            string message = userName + " покидает чат";
            byte[] data = Encoding.Unicode.GetBytes(message);
            client.Send(data, data.Length, HOST, REMOTEPORT);
            client.DropMulticastGroup(groupAddress);

            alive = false;
            client.Close();

            loginButton.Enabled = true;
            logoutButton.Enabled = false;
            sendButton.Enabled = false;
        }
                // обработчик события закрытия формы
        private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            if (alive)
                ExitChat();
        }
    }
}
```

Все клиенты будут подключаться к общему адресу "235.5.5.1". По нажатию на кнопку входа будет происходить отправка сообщения о входе нового пользователя и запускаться новая задача на прием входящих сообщений.

Для приема сообщений будет использоваться метод `ReceiveMessages()`. Так как в этой задаче мы обращаемся к элементам, созданным в главном потоке, то для добавления сообщения надо использовать делегат MethodInvoker:

```
this.Invoke(new MethodInvoker(() =>
{
    string time = DateTime.Now.ToShortTimeString();
    chatTextBox.Text = time + " " + message + "\r\n" + chatTextBox.Text;
}));
```

В качестве параметра он принимает либо ссылку на метод, либо лямбда-выражение.

По нажатию на кнопку sendButton происходит отправка сообщения. И по нажатию на кнопку выхода или по закрытию формы срабатывает метод `ExitChat()`, которые прекращает поток и осуществляет выход из чата.

При выходе мы можем столкнуться с проблемой. При прослушивании входящих сообщений задача останавливается на строке

```
byte[] data = client.Receive(ref remoteIp);
```

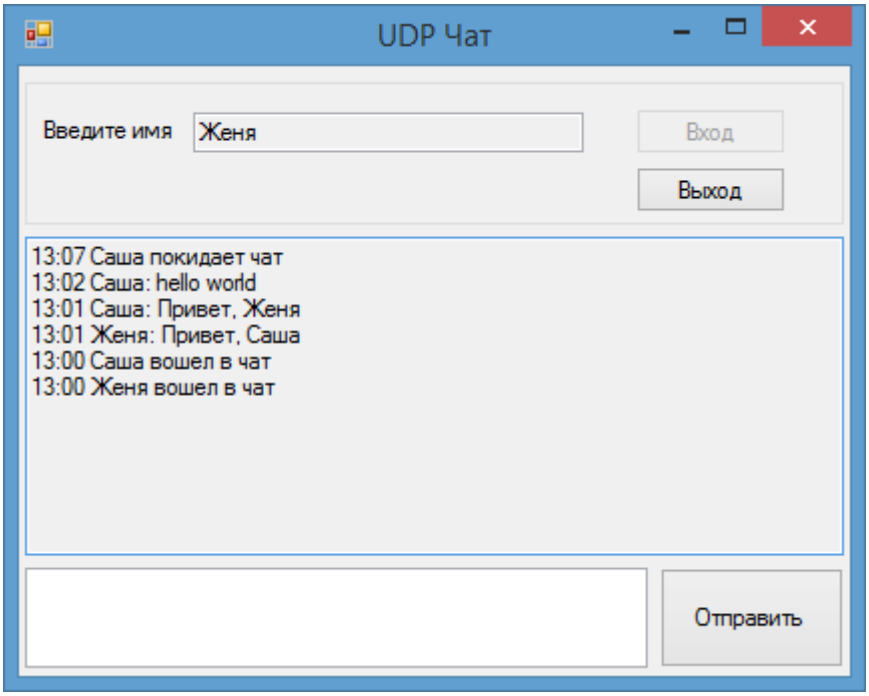
В ожидании приема сообщений. Когда происходит выход из чата по закрытию формы, то всем, в том числе и этому ожидающему клиенту посылается сообщение о выходе. Однако при этом идет закрытие и уничтожение формы, в связи с чем мы можем столкнуться с исключением ObjectDisposedException. Поэтому при приеме сообщений включается дополнительный блок обработки исключений:

```
catch (ObjectDisposedException)
{
```

```
    if (!alive)
        return;
    throw;
}
```

Поэтому если исключение было вызвано при закрытии потока, то мы его не обрабатываем. Иначе выбрасываем ошибку.

Запустим чат на нескольких машинах в рамках одной подсети и мы сможем взаимодействовать со всеми подключенными клиентами:



Потоки

NetworkStream и текстовые потоки

Для получения и отправки данных с помощью сокетов в .NET используется класс потоков **NetworkStream** из пространства имен System.Net.Sockets. Он наследуется от базового класса Stream. В то же время он отличается от других классов потоков тем, что он не является буферизованным и не поддерживает перемещение в произвольную позицию с помощью метода Seek.

Также при записи в поток не надо использовать метод Flush для сброса в поток всех данных.

Основные методы и свойства NetworkStream:

- Свойство `DataAvailable`: возвращает значение true, если в потоке есть данные. Если их нет, возвращается false.
- Метод `Read(byte[] buffer, int offset, int length);`: считывает данные из потока в массив buffer, начиная со смещения offset. Количество считываемых из потока данных указывается в параметре length
- Метод `Write(byte[] buffer, int offset, int length)`: отправляет данные из массива buffer, начиная со смещения offset, в поток. Количество отправляемых из массива данных указывается в параметре length
- Метод `Close()`: закрывает поток

При работе с NetworkStream надо учитывать, что его методы Write и Read блокируют вызывающий поток до тех пор, пока не завершится операция записи и чтения соответственно. Чтобы отправить данные, нам надо преобразовать их в бинарную форму:

```
string response = "Привет мир";
byte[] data = Encoding.UTF8.GetBytes(response);
stream.Write(data, 0, data.Length);
```

При получении данных метод Read считывает данные из потока также в массив байтов:

```
StringBuilder response = new StringBuilder();
byte[] data = new byte[256];
do
{
    int bytes = stream.Read(data, 0, data.Length);
    response.Append(Encoding.UTF8.GetString(data, 0, bytes));
}
while (stream.DataAvailable);
```

Поскольку удаленный хост может послать данные гораздо большего размера, чем размер буфера data, то для считывания всех данных, как правило используется цикл. Здесь в цикле do..while проверяем наличие данных с помощью свойства `stream.DataAvailable`

Но подобные способы работы не всегда удобны. Создание цикла, получение данных и их перекодирование в строку, добавление в объект StringBuilder - все это можно упростить. И для упрощения работы с этим потоком мы можем обертывать его в другие потоки.

Например, если мы получаем и отправляем только текстовые данные - обычные строки, то вполне логично было бы использовать классы текстовых потоков StreamWriter и StreamReader.

Например, определим класс сервера:

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

namespace TxtServer
{
    class Program
    {
        const int PORT = 5006; // порт для прослушивания подключений
        static TcpListener listener;
        static void Main(string[] args)
        {
            try
            {
                listener = new TcpListener(IPAddress.Parse("127.0.0.1"), PORT);
                listener.Start();
                Console.WriteLine("Ожидание подключений...");

                while (true)
                {
                    TcpClient client = listener.AcceptTcpClient();
                    NetworkStream stream = client.GetStream();

                    StreamReader reader = new StreamReader(stream);
                    // считываем строку из потока
                    string message = reader.ReadLine();
                    Console.WriteLine("Получено: " + message);

                    // отправляем ответ
                    StreamWriter writer = new StreamWriter(stream);
                    message = message.ToUpper();
                    Console.WriteLine("Отправлено: " + message);
                    writer.WriteLine(message);

                    writer.Close();
                    reader.Close();
                    stream.Close();
                    client.Close();
                }
            }
            catch (Exception ex)
            {
             Console.WriteLine(ex.Message);
```

```
            }
            finally
            {
                if (listener != null)
                    listener.Stop();
            }
        }
    }
}
```

И класс клиента:

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

namespace TxtClient
{
    class Program
    {
        const int PORT = 5006;
        const string ADDRESS = "127.0.0.1";
        static void Main(string[] args)
        {
            TcpClient client = null;
            try
            {
                Console.Write("Введите сообщение: ");
                string message = Console.ReadLine();
                client = new TcpClient(ADDRESS, PORT);
                NetworkStream stream = client.GetStream();

                // отправляем сообщение
                StreamWriter writer = new StreamWriter(stream);
                writer.WriteLine(message);
                writer.Flush();

                // BinaryReader reader = new BinaryReader(new BufferedStream(stream));
                StreamReader reader = new StreamReader(stream);
                message = reader.ReadLine();
                Console.WriteLine("Получен ответ: " + message);

                reader.Close();
                writer.Close();
                stream.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
```

```
                if(client!=null)
                    client.Close();
            }
        }
    }
}
```

В классах сервера и клиента поток NetworkStream обертывается в классы StreamReader и StreamWriter. Для чтения данных достаточно применить метод **ReadLine()** класса StreamReader, а для отправки данных - метод **WriteLine()** класса StreamWriter. Код значительно упрощается. При этом надо учитывать, что если мы на сервере получаем данные с помощью ReadLine, то соответственно они должны быть посланы через WriteLine.

В качестве альтернативы потокам StreamReader и StreamWriter можно использовать потоки бинарных данных BinaryReader и BinaryWriter, используя те их методы, которые связаны с работой со строками. Некоторые специалисты, в частности, Албахари, указывают, что для отправки текстовой информации бинарные потоки данных даже предпочтительнее, чем текстовые. Потому что класс StreamReader использует буфер упреждающего чтения, что может вылиться в считывание большего количества байтов, чем доступно в текущий момент. Что в результате может привести к блокировке потока.

Теперь применим бинарные потоки для чтения и отправки текста, изменив код сервера:

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

namespace TxtServer
{
    class Program
    {
        const int PORT = 5006; // порт для прослушивания подключений
        static TcpListener listener;
        static void Main(string[] args)
        {
            try
            {
                listener = new TcpListener(IPAddress.Parse("127.0.0.1"), PORT);
                listener.Start();
                Console.WriteLine("Ожидание подключений...");

                while (true)
                {
                    TcpClient client = listener.AcceptTcpClient();
                    NetworkStream stream = client.GetStream();

                   BinaryReader reader = new BinaryReader(stream);
                   // считываем строку из потока
                   string message = reader.ReadString();
                   Console.WriteLine("Получено: " + message);

                   // отправляем ответ
```

```
                BinaryWriter writer = new BinaryWriter(stream);
                message = message.ToUpper();
                Console.WriteLine("Отправлено: " + message);
                writer.Write(message);
                writer.Flush();

                writer.Close();
                reader.Close();
                stream.Close();
                client.Close();
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            if (listener != null)
                listener.Stop();
        }
    }
}
```

И соответственно изменим код клиента:

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

namespace TxtClient
{
    class Program
    {
        const int PORT = 5006;
        const string ADDRESS = "127.0.0.1";
        static void Main(string[] args)
        {
            TcpClient client = null;
            try
            {
                client = new TcpClient(ADDRESS, PORT);
                Console.Write("Введите сообщение: ");
                string message = Console.ReadLine();
                NetworkStream stream = client.GetStream();

                // отправляем сообщение
                BinaryWriter writer = new BinaryWriter(stream);
                writer.Write(message);
                writer.Flush();
```

```
            BinaryReader reader = new BinaryReader(stream);
            message = reader.ReadString();
            Console.WriteLine("Получен ответ: " + message);

            reader.Close();
            writer.Close();
            stream.Close();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            if(client!=null)
                client.Close();
        }
    }
}
```

Потоки бинарных данных

Создадим небольшое финансовое клиент-серверное приложение для демонстрации того, как мы можем отправлять и получать данные примитивных типов. Для работы с примитивными типами в .NET используются классы BinaryReader и BinaryWriter. Подобно тому, как с их помощью можно писать данные в файл, также можно добавлять данные в поток NetworkStream.

Наше приложение будет регистрировать вклады от клиентов. Клиент направляет запрос на открытие вклада, а сервер создает вклад и посылает ответ.

Вначале создадим проект сервера. И первым делом добавим в него класс Account, который будет представлять модель вклада:

```
public class Account
{
    public Account(string username, decimal sum, int period)
    {
        this.Id = Guid.NewGuid().ToString(); // генерируем номер счета
        this.Name = username;
        this.Sum = sum;
        this.Procent = period > 6 ? 10 : 1; // если период вклада больше 6 месяцев, то 10%
    }
    public string Id { get; private set; } // id - номер счета
    public string Name { get; private set; } // имя владельца
    public decimal Sum { get; private set; } // сумма
    public int Procent { get; private set; } // процент вклада
}
```

Затем добавим класс ClientObject, который будет выполнять обработку запроса от клиента:

```
public class ClientObject
{
    public TcpClient client;
    public ClientObject(TcpClient tcpClient)
        {
        client = tcpClient;
    }

    public void Process()
    {
        NetworkStream stream = null;
        try
        {
            stream = client.GetStream();
            BinaryReader reader = new BinaryReader(stream);
            // считываем данные из потока
            string name = reader.ReadString();
            decimal sum = reader.ReadDecimal();
```

```
            int period = reader.ReadInt32();
            // создаем по полученным от клиента данным объект счета
                        Account account = new Account(name, sum, period);

                        Console.WriteLine("{0} зарегистрировал счет на сумму {1}", account.Name,
account.Sum);

            // отправляем ответ в виде номера счета
            BinaryWriter writer = new BinaryWriter(stream);
            writer.Write(account.Id);
            writer.Flush();

            writer.Close();
            reader.Close();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            if (stream != null)
                stream.Close();
            if (client != null)
                client.Close();
        }
    }
}
```

Для считывания данных обертываем объект NetworkStream в BinaryReader. Затем с помощью методов считываем отдельные данные примитивных типов из потока.

После считывания создаем объект счета Account. При необходимости его можно сохранить в базу данных, выполнить с ним какие-либо действия, но в данном случае просто выводим сообщение на консоль.

И далее с помощью BinaryWriter пишем в ответ клиенту номер нового счета.

И теперь используем этот класс в главном классе программы:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Threading.Tasks;
using System.IO;

namespace FinanceServer
{
    class Program
    {
        const int PORT = 5006; // порт для прослушивания подключений
        static TcpListener listener;
        static void Main(string[] args)
```

```
        {
            try
            {
                listener = new TcpListener(IPAddress.Parse("127.0.0.1"), PORT);
                listener.Start();
                Console.WriteLine("Ожидание подключений...");

                while (true)
                {
                    TcpClient client = listener.AcceptTcpClient();
                    ClientObject clientObject = new ClientObject(client);

                    // создаем новый поток для обслуживания нового клиента
                    Task clientTask = new Task(clientObject.Process);
                    clientTask.Start();
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                if (listener != null)
                    listener.Stop();
            }
        }
    }
}
```

И также создадим проект клиента:

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

namespace FinanceClient
{
    class Program
    {
        const int PORT = 5006;
        const string ADDRESS = "127.0.0.1";

        static void Main(string[] args)
        {
            TcpClient client = null;
            try
            {
                Console.WriteLine("Для регистрации вклада введите данные!");
                Console.Write("Укажите ваше имя: ");
                string userName = Console.ReadLine();
```

```
                Console.Write("Укажите сумму вклада: ");
                decimal sum  = Decimal.Parse(Console.ReadLine());

                Console.Write("Укажите период вклада в месяцах: ");
                int period = Int32.Parse(Console.ReadLine());

                client = new TcpClient(ADDRESS, PORT);
                NetworkStream stream = client.GetStream();

                BinaryWriter writer = new BinaryWriter(stream);
                writer.Write(userName);
                writer.Write(sum);
                writer.Write(period);
                writer.Flush();

                BinaryReader reader = new BinaryReader(stream);
                string accountNumber = reader.ReadString();
                Console.WriteLine("Номер вашего счета " + accountNumber);

                reader.Close();
                writer.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                client.Close();
            }
            Console.Read();
        }
    }
}
```

Принципы работы клиента будут те же самые, что и сервера.

Запустим программы сервера и клиента. Клиент:

```
Для регистрации вклада введите данные!
Укажите ваше имя: Евгений
Укажите сумму вклада: 300
Укажите период вклада в месяцах: 7 Номер
вашего счета 874364398dc-23dc-
8841-6043829843489
```

Сервер:

```
Ожидание подключений...
Евгений зарегистрировал счет на сумму
300
```

Протокол HTTP

HttpListener

Для прослушивания подключений по протоколу HTTP и ответа на HTTP-запросы предназначен класс **HttpListener**. Данный класс построен на базе библиотеки HTTP.sys, которая является слушателем режима ядра, обрабатывающим весь трафик HTTP для Windows.

Для работы с HttpListener нам надо вначале задать адреса, которые будут использоваться для обращения к приложению. Адреса задаются через свойство **Prefixes** класса HttpListener. При этом адрес должен оканчиваться на слеш, например: `http://somesite.com:8888/connection/`

Чтобы начать прослушивать входящие подключения, надо вызвать метод `Start()` и затем метод `GetContext()` класса HttpListener.

Напишем небольшое приложение:

```
using System;
using System.Net;
using System.IO;

namespace NetConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            HttpListener listener = new HttpListener();
            // установка адресов прослушки
            listener.Prefixes.Add("http://localhost:8888/connection/");
            listener.Start();
            Console.WriteLine("Ожидание подключений...");
            // метод GetContext блокирует текущий поток, ожидая получение запроса
            HttpListenerContext context = listener.GetContext();
            HttpListenerRequest request = context.Request;
            // получаем объект ответа
            HttpListenerResponse response = context.Response;
            // создаем ответ в виде кода html
            string responseStr = "<html><head><meta charset='utf8'></head><body>Привет мир!</body>
</html>";
            byte[] buffer = System.Text.Encoding.UTF8.GetBytes(responseStr);
            // получаем поток ответа и пишем в него ответ
            response.ContentLength64 = buffer.Length;
            Stream output = response.OutputStream;
            output.Write(buffer, 0, buffer.Length);
            // закрываем поток
            output.Close();
            // останавливаем прослушивание подключений
```

```
            listener.Stop();
            Console.WriteLine("Обработка подключений завершена");
            Console.Read();
        }
    }
}
```

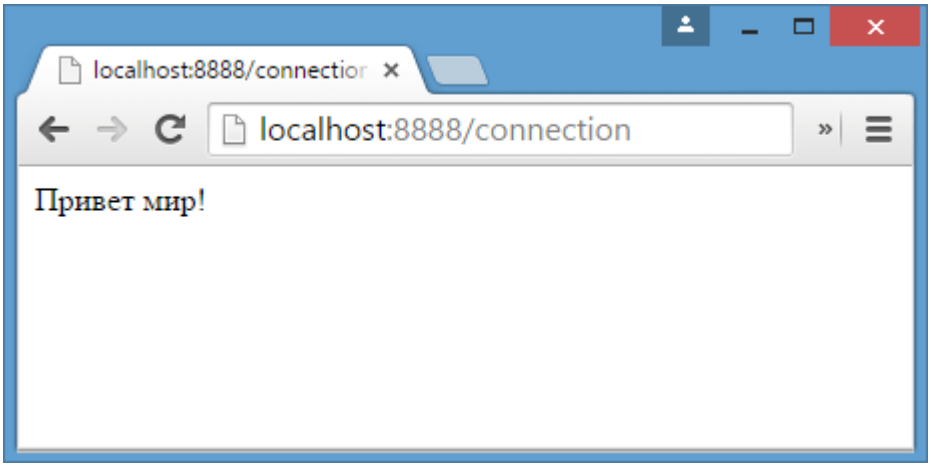
В данном случае наша программа будет прослушивать все обращения по локальному адресу *http://localhost:8888/connection/*

При вызове метода `listener.GetContext()` текущий поток блокируется, и слушатель начинает ожидать входящие подключения. Этот метод возвращает объект **HttpListenerContext**, с помощью которого можно получить доступ к объектам запроса и ответа.

Для получения запроса используется свойство `context.Request`, которое возвращает объект **HttpListenerRequest**, а для получения объекта ответа - свойство `context.Response`. После получения ответа получаем выходной поток и пишем в него массив байтов:

```
Stream output = response.OutputStream;
output.Write(buffer, 0, buffer.Length);
output.Close();
```

В итоге, если мы запустим приложение и обратимся в каком-нибудь браузере по адресу *http://localhost:8888/connection/*, то нам отобразится сформированный в приложении код html:



Также мы использовать асинхронную версию метода `GetContext()` - `GetContextAsync()`:

```
private static async Task Listen()
{
    HttpListener listener = new HttpListener();
    listener.Prefixes.Add("http://localhost:8888/");
    listener.Start();
    Console.WriteLine("Ожидание подключений...");
    HttpListenerContext context = await listener.GetContextAsync();
    // остальное содержимое
    listener.Stop();
}
```

И также чтобы постоянно прослушивать всходящие подключения, можно заключить код слушателя в бесконечный цикл while:

```
private static async Task Listen()
{
    HttpListener listener = new HttpListener();
    listener.Prefixes.Add("http://localhost:8888/");
    listener.Start();
    Console.WriteLine("Ожидание подключений...");

    while(true)
    {
        HttpListenerContext context = await listener.GetContextAsync();
        HttpListenerRequest request = context.Request;
        HttpListenerResponse response = context.Response;

                string responseString = "<html><head><meta charset='utf8'></head><body>Привет мир!
</body></html>";
        byte[] buffer = System.Text.Encoding.UTF8.GetBytes(responseString);
        response.ContentLength64 = buffer.Length;
        Stream output = response.OutputStream;
        output.Write(buffer, 0, buffer.Length);
        output.Close();
    }
}
```

Работа с электронной почтой

Отправка почты. SmtpClient

Для отправки почты в среде интернет используется протокол SMTP (Simple Mail Transfer Protocol). Данный протокол указывает, как почтовые сервера взаимодействуют при передаче электронной почты.

Для работы с протоколом SMTP и отправки электронной почты в .NET предназначен класс **SmtpClient** из пространства имен **System.Net.Mail**.

Этот класс определяет ряд свойств, которые позволяют настроить отправку:

- `Host`: smtp-сервер, с которого производится отправление почты. Например, smtp.yandex.ru
- `Port`: порт, используемый smp-сервером. Если не указан, то по умолчанию используется 25 порт.
- `Credentials`: аутентификационные данные отправителя
- `EnableSsl`: указывает, будет ли использоваться протокол SSL при отправке

Еще одним ключевым классом, который используется при отправке, является **MailMessage**. Данный класс представляет собой отправляемое сообщение. Среди его свойств можно выделить следующие:

- `Attachments`: содержит все прикрепления к письму
- `Body`: непосредственно текст письма
- `From`: адрес отправителя. Представляет объект MailAddress
- `To`: адрес получателя. Также представляет объект MailAddress
- `Subject`: определяет тему письма
- `IsBodyHtml`: указывает, представляет ли письмо содержимое с кодом html

Используем эти классы и выполним отправку письма:

```
using System;
using System.Net;
using System.IO;
using System.Threading.Tasks;
using System.Net.Mail;

namespace NetConsoleApp
{
    class Program
   {
         static void Main(string[] args)
```

```
        {

            // отправитель - устанавливаем адрес и отображаемое в письме имя
            MailAddress from = new MailAddress("somemail@gmail.com", "Tom");
            // кому отправляем
            MailAddress to = new MailAddress("somemail@yandex.ru");
            // создаем объект сообщения
            MailMessage m = new MailMessage(from, to);
            // тема письма
            m.Subject = "Тест";
            // текст письма
            m.Body = "<h2>Письмо-тест работы smtp-клиента</h2>";
                        // письмо представляет код html
            m.IsBodyHtml = true;
            // адрес smtp-сервера и порт, с которого будем отправлять письмо
            SmtpClient smtp = new SmtpClient("smtp.gmail.com", 587);
            // логин и пароль
            smtp.Credentials = new NetworkCredential("somemail@gmail.com", "mypassword");
            smtp.EnableSsl = true;
            smtp.Send(m);
            Console.Read();
        }
    }
}
```

Для отправки применяется метод `Send()`, в который передается объект MailMessage.

Также мы можем использовать асинхронную версию отправки с помощью метода `SendMailAsync`:

```
using System;
using System.Net;
using System.IO;
using System.Threading.Tasks;
using System.Net.Mail;

namespace NetConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {

            SendEmailAsync().GetAwaiter();
            Console.Read();
        }

        private static async Task SendEmailAsync()
        {
            MailAddress from = new MailAddress("somemail@gmail.com", "Tom");
            MailAddress to = new MailAddress("somemail@yandex.ru");
            MailMessage m = new MailMessage(from, to);
            m.Subject = "Тест";
            m.Body = "Письмо-тест 2 работы smtp-клиента";
```

```
            SmtpClient smtp = new SmtpClient("smtp.gmail.com", 587);
            smtp.Credentials = new NetworkCredential("somemail@gmail.com", "mypassword");
            smtp.EnableSsl = true;
            await smtp.SendMailAsync(m);
            Console.WriteLine("Письмо отправлено");
        }
    }
}
```

Добавление вложений

К письму мы можем прикрепить вложения с помощью свойства Attachments. Каждое вложение представляет объект **System.Net.Mail.Attachment**:

```
MailAddress from = new MailAddress("somemail@gmail.com", "Tom");
MailAddress to = new MailAddress("somemail2@yandex.ru");
MailMessage m = new MailMessage(from, to);
m.Attachments.Add(new Attachment("D://temlog.txt"));
```

Протокол FTP

FtpWebRequest и FtpWebResponse

Протокол FTP (File Transfer Protocol) предназначен для передачи файлов по сети. Он работает поверх протокола TCP и использует 21 порт. Однако так как это довольно используемый протокол, и чтобы разработчикам не приходилось с нуля создавать весь функционал, используя TCP-сокеты, в библиотеке классов .NET уже есть готовые решения. Эти решения представляют классы **FtpWebRequest** и **FtpWebResponse**. Эти классы являются производными от WebRequest и WebResponse и позволяют отправлять запрос к FTP-серверу и получать от него ответ.

FtpWebRequest позволяет отправить запрос к серверу. Для настройки запроса мы можем использовать следующие его свойства:

- `Credentials`: задает или возвращает аутентификационные данные пользователя
- `EnableSsl`: указывает, надо ли использовать ssl-соединение
- `Method`: задает команду протокола FTP, которая будет использоваться в запросе
- `UsePassive`: при значении true устанавливает пассивный режим запроса к серверу
- `UseBinary`: указывает тип данных, которые будут использоваться в запросе. Значение true указывает, что передаваемые данные являются двоичными, а значение false - что данные будут представлять текст. Значение по умолчанию — true

Например, загрузим текстовый файл с фтп-сервера:

```
using System;
using System.IO;
using System.Net;
using System.Text;

namespace FtpConsoleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            // Создаем объект FtpWebRequest
            FtpWebRequest request = (FtpWebRequest)WebRequest.Create("ftp://127.0.0.1/test.txt");
            // устанавливаем метод на загрузку файлов
            request.Method = WebRequestMethods.Ftp.DownloadFile;

            // если требуется логин и пароль, устанавливаем их
           //request.Credentials = new NetworkCredential("login",
          "password");  //request.EnableSsl = true; // если используется ssl

            // получаем ответ от сервера в виде объекта FtpWebResponse
```

```
            FtpWebResponse response = (FtpWebResponse)request.GetResponse();

            // получаем поток ответа
            Stream responseStream = response.GetResponseStream();
            // сохраняем файл в дисковой системе
            // создаем поток для сохранения файла
            FileStream fs = new FileStream("newTest.txt",FileMode.Create);

            //Буфер для считываемых данных
            byte[] buffer = new byte[64];
            int size = 0;

            while ((size = responseStream.Read(buffer, 0, buffer.Length)) > 0)
            {
                fs.Write(buffer, 0, size);

            }
            fs.Close();
            response.Close();

            Console.WriteLine("Загрузка и сохранение файла завершены");
            Console.Read();
        }
    }
}
```

В данном случае идет обращение к фтп-серверу "ftp://127.0.0.1", который я поднял на локальном компьютере, но это может быть любой адрес рабочего фтп-сервера. Если нам надо просто загрузить файл, то мы можем использовать метод

```
request.Method = WebRequestMethods.Ftp.DownloadFile
```

При отправке запроса нам надо указать соответствующий метод.

Если фтп-сервер требует установки логина и пароля, то применяется свойство Credentials:

```
request.Credentials = new NetworkCredential("login", "password");
```

Если сервер использует ssl-соединение, то надо его установить в запросе: `request.EnableSsl = true`

После отправки запроса мы можем получить ответ в виде FtpWebResponse:

```
FtpWebResponse response = (FtpWebResponse)request.GetResponse();
Stream responseStream = response.GetResponseStream();
```

Получив поток ответа, мы можем им манипулировать. В данном случае с помощью FileStream сохраняем файл в папку программы под именем newTest.txt.

Команды протокола FTP

Для выполнения запросов в протоколе FTP используются команды. Например, команда `LIST` предназначена для получения списка файлов каталога сервера, команда `STOR` применяется для сохранения файла и так далее. В .NET для отправки нужной команды нам не надо запихивать ее в тело запроса, так как можно использовать свойство `Method` класса FtpWebRequest.

Этот метод в качестве значения принимает строки, определенные в классе **WebRequestMethods.Ftp**:

- `AppendFile`: добавляет в запрос команду APPE, которая используется для присоединения файла к существующему файлу на FTP-сервере
- `DeleteFile`: добавляет в запрос команду DELE, которая используется для удаления файла на FTP-сервере
- `DownloadFile`: добавляет команду RETR, которая используется для загрузки файла
- `GetDateTimestamp`: представляет команду MDTM, которая применяется для получения даты и времени из файла
- `GetFileSize`: команда SIZE, получение размера файла
- `ListDirectory`: команда NLIST, возвращает краткий список файлов на сервере
- `ListDirectoryDetails`: команда LIST, возвращает подробный список файлов на FTP-сервере
- `MakeDirectory`: команда MKD, создает каталог на FTP-сервере
- `PrintWorkingDirectory`: команда PWD, отображает имя текущего рабочего каталога
- `RemoveDirectory`: команда RMD, удаляет каталог
- `Rename`: команда RENAME, переименовывает каталог
- `UploadFile`: команда STOR, загружает файл на FTP-сервер
- `UploadFileWithUniqueName`: команда STOU, загружает файл с уникальным именем на FTP-сервер

Однако в реальности при работе с конкретными ftp-серверами нам не все эти команды будут доступны в силу ограничений на чтение/запись на стороне сервера, которые может установить администратор.

Например, загрузим текстовый файл на фтп-сервер:

```
using System;
using System.IO;
using System.Net;
using System.Text;

namespace FtpConsoleClient
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            // Создаем объект FtpWebRequest - он указывает на файл, который будет создан
            FtpWebRequest request = (FtpWebRequest)WebRequest.Create("ftp://127.0.0.1/hellow.txt");
            // устанавливаем метод на загрузку файлов
            request.Method = WebRequestMethods.Ftp.UploadFile;

            // создаем поток для загрузки файла
            FileStream fs = new FileStream("D://test.txt", FileMode.Open);
            byte[] fileContents = new byte[fs.Length];
            fs.Read(fileContents, 0, fileContents.Length);
            fs.Close();
            request.ContentLength = fileContents.Length;

            // пишем считанный в массив байтов файл в выходной поток
            Stream requestStream = request.GetRequestStream();
            requestStream.Write(fileContents, 0, fileContents.Length);
            requestStream.Close();

            // получаем ответ от сервера в виде объекта FtpWebResponse
            FtpWebResponse response = (FtpWebResponse)request.GetResponse();

            Console.WriteLine("Загрузка файлов завершена. Статус: {0}",
response.StatusDescription);

            response.Close();
            Console.Read();
        }
    }
}
```

Для загрузки на сервер у нас, естественно, должны быть права на запись. Возможно, потребуется установить логин и пароль для доступа к серверу.

С помощью другой команды получим содержимое сервера:

```
FtpWebRequest request = (FtpWebRequest)WebRequest.Create("ftp://127.0.0.1/");

request.Method = WebRequestMethods.Ftp.ListDirectoryDetails;

FtpWebResponse response = (FtpWebResponse)request.GetResponse();
Console.WriteLine("Содержимое сервера:");
Console.WriteLine();

Stream responseStream = response.GetResponseStream();
StreamReader reader = new StreamReader(responseStream);
Console.WriteLine(reader.ReadToEnd());

reader.Close();
responseStream.Close();
```

```
response.Close();
Console.Read();
```

Поскольку в данном случае сервер возвращает текстовую информацию о файлах и каталогах, то с помощью объекта StreamReader мы можем считать ее и вывести на консоль: