

**Кафедра высшей математики и информационных технологий**

**ПРОГРАММИРОВАНИЕ СЕТЕВЫХ  
ПРИЛОЖЕНИЙ**

**Лабораторный практикум**

# **СОДЕРЖАНИЕ**

## **ЛАБОРАТОРНАЯ РАБОТА № 1**

1. Сокет
2. Типы сокетов
  - 2.1 Поточковые сокет (stream socket)
  - 2.2 Дейтаграммные сокет (datagram socket)
  - 2.3 Сырые сокет (raw socket)
3. Порты
4. Работа с сокетами в .NET
5. Класс Socket
6. Клиент-серверное приложение на сокетах TCP
  - 6.1 Сервер
  - 6.2 Клиент
7. Использование сокетов для работы с UDP
  - 7.1 Программа UDP-клиент
8. Задание

## **ЛАБОРАТОРНАЯ РАБОТА № 2**

1. Протокол TCP
2. TCP-клиент. Класс TcpClient
3. Класс TcpListener
4. Задание

## **Лабораторная работа № 3**

1. Многопоточное клиент-серверное приложение TCP
2. Задание

## **Лабораторная работа № 4**

1. Консольный TCP-чат
  - 1.1 Класс ClientObject
  - 1.2 Класс ServerObject
  - 1.3 Класс Program
  - 1.4 Изменение стандартного класса Program
2. Задание

## **Лабораторная работа № 5**

1. Протокол UDP
2. UdpClient
3. Широковещательная рассылка
4. Чат с широковещательной рассылкой на Windows Forms
5. Задание

## **Лабораторная работа № 6**

1. Протокол HTTP
  - 1.1 HTTP-заголовки
    - 1.1.2 Заголовки ответов
    - 1.1.3 Общие заголовки
2. Классы HttpRequest и HttpResponse
- 2.1 Работа с HTTP-заголовками
3. Класс WebClient
  - 3.1 Загрузка файлов
4. Аутентификация и разрешения
5. Web-прокси
6. Класс HttpListener
7. Задание

## **Лабораторная работа № 7**

1. Протокол FTP
  - 1.2 Создание FTP-сайта
  - 1.2 Классы FtpWebRequest и FtpWebResponse
2. Команды протокола FTP
3. Задание

## **Лабораторная работа № 8**

1. Протокол SMTP (SmtpClient, MailMessage)
2. Задание

## Лабораторная работа №1. Сокеты

**Цель работы.** Приобрести практические навыки создания клиент-серверных приложений на потоковых и дейтаграмных сокетах опираясь на протоколы TCP и UDP.

### Задачи:

1. Изучить, что такое сокеты и типы сокетов.
2. Рассмотреть классы в пространстве имен System.Net.Sockets обеспечивающих поддержку сокетов в .NET
3. Изучить пример серверного приложения на сокетах TCP.
4. Разработать приложение клиент-сервер согласно заданию.

### Сокет

Сокет – это один конец двустороннего канала связи между двумя программами, работающими в сети. Соединяя вместе два сокета, можно передавать данные между разными процессами (локальными или удаленными). Реализация сокетов обеспечивает инкапсуляцию протоколов сетевого и транспортного уровней.

Первоначально сокеты были разработаны для UNIX в Калифорнийском университете в Беркли. В UNIX обеспечивающий связь метод ввода-вывода следует алгоритму open/read/write/close. Прежде чем ресурс использовать, его нужно открыть, задав соответствующие разрешения и другие параметры. Как только ресурс открыт, из него можно считывать или в него записывать данные. После использования ресурса пользователь должен вызывать метод Close(), чтобы подать сигнал операционной системе о завершении его работы с этим ресурсом.

Когда в операционную систему UNIX были добавлены средства *межпроцессного взаимодействия (Inter-Process Communication, IPC)* и сетевого обмена, был заимствован привычный шаблон ввода-вывода. Все ресурсы, открытые для связи, в UNIX и Windows идентифицируются дескрипторами. Эти дескрипторы, или *описатели (handles)*, могут указывать на файл, память или какой-либо другой канал связи, а фактически указывают на внутреннюю структуру данных, используемую операционной системой. Сокет, будучи таким же ресурсом, тоже представляется дескриптором. Следовательно, для сокетов жизнь дескриптора можно разделить на три фазы: открыть (создать) сокет, получить из сокета или отправить сокету и в конце концов закрыть сокет.

Интерфейс IPC для взаимодействия между разными процессами построен поверх методов ввода-вывода. Они облегчают для сокетов отправку и получение данных. Каждый целевой объект задается адресом сокета, следовательно, этот адрес можно указать в клиенте, чтобы установить соединение с целью.

## Типы сокетов

Существуют два основных типа сокетов – потоковые сокет и дейтаграммные.

### Потоковые сокет (stream socket)

Потоковый сокет – это сокет с установленным соединением, состоящий из потока байтов, который может быть двунаправленным, т. е. через эту конечную точку приложение может и передавать, и получать данные.

Потоковый сокет гарантирует исправление ошибок, обрабатывает доставку и сохраняет последовательность данных. На него можно положиться в доставке упорядоченных, сдублированных данных. Потоковый сокет также подходит для передачи больших объемов данных, поскольку накладные расходы, связанные с установлением отдельного соединения для каждого отправляемого сообщения, может оказаться неприемлемым для небольших объемов данных. Потоковые сокет достигают этого уровня качества за счет использования протокола *Transmission Control Protocol (TCP)*. TCP обеспечивает поступление данных на другую сторону в нужной последовательности и без ошибок.

Для этого типа сокетов путь формируется до начала передачи сообщений. Тем самым гарантируется, что обе участвующие во взаимодействии стороны принимают и отвечают.

Если приложение отправляет получателю два сообщения, то гарантируется, что эти сообщения будут получены в той же последовательности.

Однако отдельные сообщения могут дробиться на пакеты, и способа определить границы записей не существует. При использовании TCP этот протокол берет на себя разбиение передаваемых данных на пакеты соответствующего размера, отправку их в сеть и сборку их на другой стороне. Приложение знает только, что оно отправляет на уровень TCP определенное число байтов и другая сторона получает эти байты. В свою очередь TCP эффективно разбивает эти данные на пакеты подходящего размера, получает эти пакеты на другой стороне, выделяет из них данные и объединяет их вместе.

Потоки базируются на явных соединениях: сокет А запрашивает соединение с сокетом В, а сокет В либо соглашается с запросом на установление соединения, либо отвергает его.

Если данные должны гарантированно доставляться другой стороне или размер их велик, потоковые сокет предпочтительнее дейтаграммных. Следовательно, если надежность связи между двумя приложениями имеет первостепенное значение, выбирайте потоковые сокет. Сервер электронной почты представляет пример приложения, которое должно доставлять содержание в правильном порядке, без дублирования и пропусков. Потоковый

сокет рассчитывает, что TCP обеспечит доставку сообщений по их назначениям.

### **Дейтаграммные сокеты (datagram socket)**

Дейтаграммные сокеты иногда называют сокетами без организации соединений, т. е. никакого явного соединения между ними не устанавливается – сообщение отправляется указанному сокету и, соответственно, может получаться от указанного сокета.

Потоковые сокеты по сравнению с дейтаграммными действительно дают более надежный метод, но для некоторых приложений накладные расходы, связанные с установкой явного соединения, неприемлемы (например, сервер времени суток, обеспечивающий синхронизацию времени для своих клиентов). В конце концов на установление надежного соединения с сервером требуется время, которое просто вносит задержки в обслуживание, и задача серверного приложения не выполняется. Для сокращения накладных расходов нужно использовать дейтаграммные сокеты.

Использование дейтаграммных сокетов требует, чтобы передачей данных от клиента к серверу занимался *User Datagram Protocol (UDP)*. В этом протоколе на размер сообщений налагаются некоторые ограничения, и в отличие от потоковых сокетов, умеющих надежно отправлять сообщения серверу-адресату, дейтаграммные сокеты надежность не обеспечивают. Если данные затерялись где-то в сети, сервер не сообщит об ошибках.

Кроме двух рассмотренных типов существует также обобщенная форма сокетов, которую называют необрабатываемыми или сырыми.

### **Сырые сокеты (raw socket)**

Главная цель использования сырых сокетов состоит в обходе механизма, с помощью которого компьютер обрабатывает TCP/IP. Это достигается обеспечением специальной реализации стека TCP/IP, замещающей механизм, предоставленный стеком TCP/IP в ядре – пакет непосредственно передается приложению и, следовательно, обрабатывается гораздо эффективнее, чем при проходе через главный стек протоколов клиента.

По определению, *сырой сокет* – это сокет, который принимает пакеты, обходит уровни TCP и UDP в стеке TCP/IP и отправляет их непосредственно приложению.

При использовании таких сокетов пакет не проходит через фильтр TCP/IP, т.е. никак не обрабатывается, и предстает в своей сырой форме. В таком случае обязанность правильно обработать все данные и выполнить такие действия, как удаление заголовков и разбор полей, ложится на получающее приложение – все равно, что включить в приложение небольшой стек TCP/IP.

Однако нечасто может потребоваться программа, работающая с сырыми сокетами. Если вы не пишете системное программное обеспечение или программу, аналогичную анализатору пакетов, вникать в такие детали не придется. Сырые сокеты главным образом используются при разработке специализированных низкоуровневых протокольных приложений. Например, такие разнообразные утилиты TCP/IP, как `trace route`, `ping` или `arp`, используют сырые сокеты.

Работа с сырыми сокетами требует солидного знания базовых протоколов TCP/UDP/IP.

## Порты

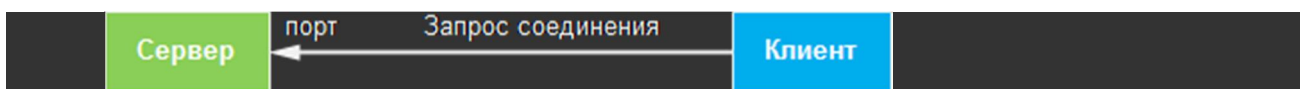
Порт определен, чтобы разрешить задачу одновременного взаимодействия с несколькими приложениями. По существу с его помощью расширяется понятие IP-адреса. Компьютер, на котором в одно время выполняется несколько приложений, получая пакет из сети, может идентифицировать целевой процесс, пользуясь уникальным номером порта, определенным при установлении соединения.

Сокет состоит из IP-адреса машины и номера порта, используемого приложением TCP. Поскольку IP-адрес уникален в Интернете, а номера портов уникальны на отдельной машине, номера сокетов также уникальны во всем Интернете. Эта характеристика позволяет процессу общаться через сеть с другим процессом исключительно на основании номера сокета.

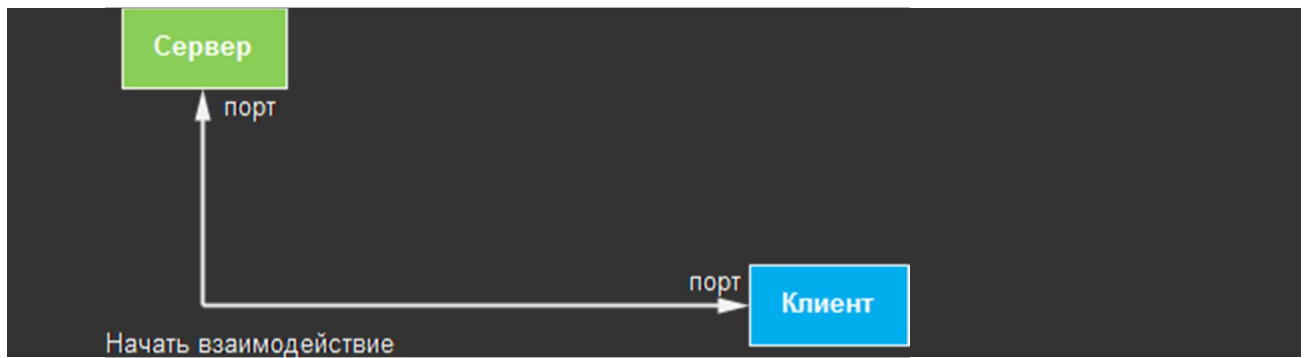
За определенными службами номера портов зарезервированы – это широко известные номера портов, например порт 21, использующийся в FTP. Ваше приложение может пользоваться любым номером порта, который не был зарезервирован и пока не занят. Агентство *Internet Assigned Numbers Authority (IANA)* ведет перечень широко известных номеров портов.

Обычно приложение клиент-сервер, использующее сокеты, состоит из двух разных приложений - клиента, иницирующего соединение с целью (сервером), и сервера, ожидающего соединения от клиента.

Например, на стороне клиента, приложение должно знать адрес цели и номер порта. Отправляя запрос на соединение, клиент пытается установить соединение с сервером:



Если события развиваются удачно, при условии что сервер запущен прежде, чем клиент попытался с ним соединиться, сервер соглашается на соединение. Дав согласие, серверное приложение создает новый сокет для взаимодействия именно с установившим соединение клиентом:



Теперь клиент и сервер могут взаимодействовать между собой, считывая сообщения каждый из своего сокета и, соответственно, записывая сообщения.

## Работа с сокетами в .NET

Поддержку сокетов в .NET обеспечивают классы в пространстве имен `System.Net.Sockets` – начнем с их краткого описания.

Класс	Описание
<i>MulticastOption</i>	Класс <i>MulticastOption</i> устанавливает значение IP-адреса для присоединения к IP-группе или для выхода из нее.
<i>NetworkStream</i>	Класс <i>NetworkStream</i> реализует базовый класс потока, из которого данные отправляются и в котором они получаются. Это абстракция высокого уровня, представляющая соединение с каналом связи TCP/IP.
<i>TcpClient</i>	Класс <i>TcpClient</i> строится на классе <code>Socket</code> , чтобы обеспечить TCP-обслуживание на более высоком уровне. <i>TcpClient</i> предоставляет несколько методов для отправки и получения данных через сеть.
<i>TcpListener</i>	Этот класс также построен на низкоуровневом классе <code>Socket</code> . Его основное назначение – серверные приложения. Он ожидает входящие запросы на соединения от клиентов и уведомляет приложение о любых соединениях.
<i>UdpClient</i>	UDP – это протокол, не организующий соединение, следовательно, для реализации UDP-обслуживания в .NET требуется другая функциональность. Класс <i>UdpClient</i> предназначен для реализации UDP-обслуживания.



<i>SocketException</i>	Это исключение порождается, когда в сокете возникает ошибка.
<i>Socket</i>	Последний класс в пространстве имен System.Net.Sockets – это сам класс Socket. Он обеспечивает базовую функциональность приложения сокета.

## Класс Socket

Класс Socket играет важную роль в сетевом программировании, обеспечивая функционирование как клиента, так и сервера. Главным образом, вызовы методов этого класса выполняют необходимые проверки, связанные с безопасностью, в том числе проверяют разрешения системы безопасности, после чего они переправляются к аналогам этих методов в Windows Sockets API.

Прежде чем обращаться к примеру использования класса Socket, рассмотрим некоторые важные свойства и методы этого класса:

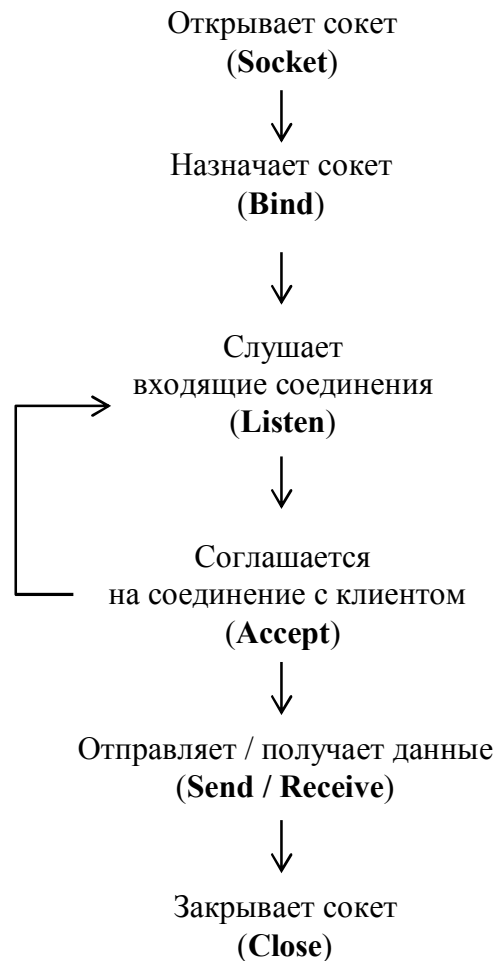
Свойство	Описание
AddressFamily	Дает семейство адресов сокета – значение из перечисления Socket.AddressFamily.
Available	Возвращает объем доступных для чтения данных.
Blocking	Дает или устанавливает значение, показывающее, находится ли сокет в блокирующем режиме.
Connected	Возвращает значение, информирующее, соединен ли сокет с удаленным хостом.
LocalEndPoint	Дает локальную конечную точку.
ProtocolType	Дает тип протокола сокета.
RemoteEndPoint	Дает удаленную конечную точку сокета.

SocketType	Дает тип сокета.
Accept()	Создает новый сокет для обработки входящего запроса на соединение.
Bind()	Связывает сокет с локальной конечной точкой для ожидания входящих запросов на соединение.
Close()	Заставляет сокет закрыться.
Connect()	Устанавливает соединение с удаленным хостом.
GetSocketOption()	Возвращает значение SocketOption.
IOControl()	Устанавливает для сокета низкоуровневые режимы работы. Этот метод обеспечивает низкоуровневый доступ к лежащему в основе классу Socket.
Listen()	Помещает сокет в режим прослушивания (ожидания). Этот метод предназначен только для серверных приложений.
Receive()	Получает данные от соединенного сокета.
Poll()	Определяет статус сокета.
Select()	Проверяет статус одного или нескольких сокетов.
Send()	Отправляет данные соединенному сокету.
SetSocketOption()	Устанавливает опцию сокета.
Shutdown()	Запрещает операции отправки и получения данных на сокете.

## Клиент-серверное приложение на сокетах TCP

### Сервер

Рассмотрим пример, как создать сервер, работающий по протоколу TCP, с помощью сокетов. Структура сервера показано на следующей функциональной диаграмме:



Листинг 1.

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

namespace SocketTcpServer
{
    class Program {
        // Резервируем порт для приема входящих запросов
        static int port = 8001;

        static void Main(string[] args)
        {
            // Создаем сокет (данный сокет используется только для установки соединения)
            Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
```

```
ProtocolType.Tcp);
```

```
// Устанавливаем для сокета локальную конечную точку
```

```
IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), port);
```

```
try
```

```
{
```

```
// Связываем сокет с локальной точкой, по которой будем принимать данные  
socket.Bind(ipPoint);
```

```
// Начинаем слушать
```

```
// Метод Listen() ожидает подключения со стороны клиентов
```

```
socket.Listen(5);
```

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

```
while (true)
```

```
{
```

```
// Программа приостанавливается, ожидая входящее СОЕДИНЕНИЕ
```

```
// Метод Accept() выдает другой объект типа Socket,
```

```
// который используется для работы с новым соединением
```

```
// (т.е. для приема и передачи данных при помощи методов Receive и Send )
```

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

```
// Буфер для приема входящего сообщения
```

```
byte[] bytes = new byte[1024];
```

```
// Метод Receive() считывает данные в буфер и
```

```
// возвращает число успешно прочитанных байтов
```

```
int bytesRec = handler.Receive(bytes);
```

```
// Преобразуем данные в строку
```

```
string data = null;
```

```
data += Encoding.UTF8.GetString(bytes, 0, bytesRec);
```

```
// Отображаем полученное сообщение
```

```
Console.Write("Полученный текст: " + data + "\n\n");
```

```
// закрываем сокет созданный методом Accept()
```

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

```
handler.Close();
```

```
}
```

```
}
```

```
catch (Exception ex)
```

```
{
```

```
Console.WriteLine(ex.ToString());
```

```
}
```

```
finally
```

```
{
```

```
Console.ReadLine();
```

```
}
```

```
}
```

```
}  
}
```

Рассмотрим структуру данной программы.

Первый шаг заключается в установлении для сокета локальной конечной точки. Прежде чем открывать сокет для ожидания соединений, нужно подготовить для него адрес локальной конечной точки. Уникальный адрес для обслуживания TCP/IP определяется комбинацией IP-адреса хоста с номером порта обслуживания, которая (комбинация) создает конечную точку для обслуживания (сокета).

Класс **Dns** предоставляет методы, возвращающие информацию о сетевых адресах, поддерживаемых устройством в локальной сети. Если у устройства локальной сети имеется более одного сетевого адреса, класс Dns возвращает информацию обо всех сетевых адресах, и приложение должно выбрать из массива подходящий адрес для обслуживания.

Создадим **EndPoint** для сервера, комбинируя *первый IP-адрес* хоста, полученный от метода *Dns.Resolve()*, с *номером порта*:

```
IPHostEntry ipHost = Dns.GetHostEntry("localhost");  
IPAddress ipAddr = ipHost.AddressList[0];  
EndPoint ipEndPoint = new EndPoint(ipAddr, 11000);
```

Здесь класс *EndPoint* представляет *localhost* на порте 11000.

Далее новым экземпляром класса *Socket* создаем потоковый сокет:

```
Socket sListener = new Socket(ipAddr.AddressFamily, SocketType.Stream,  
ProtocolType.Tcp);
```

Перечисление **AddressFamily** указывает схемы адресации, которые экземпляр класса *Socket* может использовать для разрешения адреса.

В параметре **SocketType** различаются сокеты TCP и UDP. В нем можно определить в том числе следующие значения:

<i>Dgram</i>	Поддерживает дейтаграммы. Значение Dgram требует указать Udp для типа протокола и InterNetwork в параметре семейства адресов.
<i>Raw</i>	Поддерживает доступ к базовому транспортному протоколу.
<i>Stream</i>	Поддерживает потоковые сокеты. Значение Stream требует указать Tcp для типа протокола.

Третий и последний параметр определяет тип протокола, требуемый для сокета. В параметре **ProtocolType** можно указать следующие наиболее важные значения – Tcp, Udp, Ip, Raw.

Следующим шагом должно быть назначение сокета с помощью метода **Bind()**. Когда сокет открывается конструктором, ему не назначается имя, а только резервируется дескриптор. Для назначения имени сокету сервера

вызывается метод `Bind()`. Чтобы сокет клиента мог идентифицировать потоковый сокет TCP, серверная программа должна дать имя своему сокету:

```
sListener.Bind(ipEndPoint);
```

Метод `Bind()` связывает сокет с локальной конечной точкой. Вызывать метод `Bind()` надо до любых попыток обращения к методам `Listen()` и `Accept()`.

Теперь, создав сокет и связав с ним имя, можно слушать входящие сообщения, воспользовавшись методом **`Listen()`**. В состоянии прослушивания сокет будет ожидать входящие попытки соединения:

```
sListener.Listen(10);
```

В параметре определяется *задел* (*backlog*), указывающий максимальное число соединений, ожидающих обработки в очереди. В приведенном коде значение параметра допускает накопление в очереди до десяти соединений.

В состоянии прослушивания надо быть готовым дать согласие на соединение с клиентом, для чего используется метод **`Accept()`**. С помощью этого метода получается соединение клиента и завершается установление связи имен клиента и сервера. Метод `Accept()` блокирует поток вызывающей программы до поступления соединения.

Метод `Accept()` извлекает из очереди ожидающих запросов первый запрос на соединение и создает для его обработки новый сокет. Хотя новый сокет создан, первоначальный сокет продолжает слушать и может использоваться с многопоточной обработкой для приема нескольких запросов на соединение от клиентов. Никакое серверное приложение не должно закрывать слушающий сокет. Он должен продолжать работать наряду с сокетами, созданными методом `Accept` для обработки входящих запросов клиентов.

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

Как только клиент и сервер установили между собой соединение, можно отправлять и получать сообщения, используя методы **`Send()`** и **`Receive()`** класса `Socket`.

Метод `Send()` записывает исходящие данные сокету, с которым установлено соединение. Метод `Receive()` считывает входящие данные в потоковый сокет. При использовании системы, основанной на TCP, перед выполнением методов `Send()` и `Receive()` между сокетами должно быть установлено соединение. Точный протокол между двумя взаимодействующими сущностями должен быть определен заблаговременно, чтобы клиентское и серверное приложения не блокировали друг друга, не зная, кто должен отправить свои данные первым.

Когда обмен данными между сервером и клиентом завершается, нужно закрыть соединение используя методы **`Shutdown()`** и **`Close()`**:

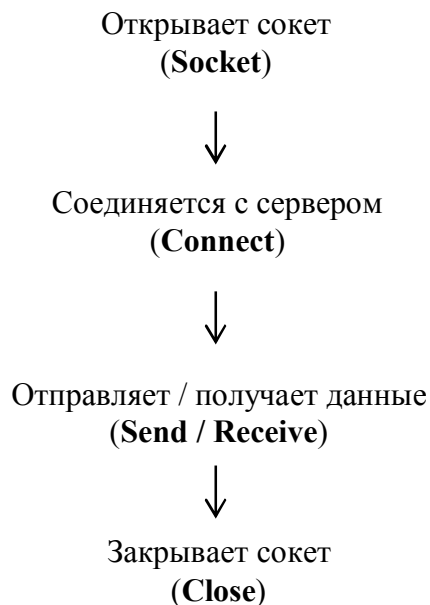
```
handler.Shutdown(SocketShutdown.Both);  
handler.Close();
```

`SocketShutdown` – это перечисление, содержащее три значения для остановки: *Both* – останавливает отправку и получение данных сокетом, *Receive* – останавливает получение данных сокетом и *Send* – останавливает отправку данных сокетом.

Сокет закрывается при вызове метода `Close()`, который также устанавливает в свойстве `Connected` сокета значение `false`.

## Клиент

Функции, которые используются для создания приложения-клиента, более или менее напоминают серверное приложение. Как и для сервера, используются те же методы для определения конечной точки, создания экземпляра сокета, отправки и получения данных и закрытия сокета. Структура клиента показано на следующей функциональной диаграмме:



Листинг 2.

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

namespace SocketTcpClient
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
                ProtocolType.Tcp);
```

```

IPEndPoint ipPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8001);

// Подключаемся к серверу, используя метод Connect() и конечную удаленную точку
socket.Connect(ipPoint);

// Формируем и отправляем сообщение
byte[] data = Encoding.UTF8.GetBytes("Запрос клиента");
socket.Send(data);

// Закрываем сокет
socket.Shutdown(SocketShutdown.Both);
socket.Close();
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
finally
{
    //Console.ReadLine();
}
}
}
}

```

Единственный новый метод — метод **Connect()**, используется для соединения с удаленным сервером.

## 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. Вызов метода **Socket()** выглядит следующим образом:

```

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

```

Так как мы создаем сокет дейтаграмм, использующий протокол UDP, вторым аргументом, задающим тип создаваемого сокета, должен быть **SocketType.Dgram**.

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

```

IPEndPoint localEP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1111);
socket.Bind(localEP);

```



После создания сокета в приложении-получателе и привязки его к определенному адресу и порту можно принимать данные от приложения-отправителя. 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 первого параметра в метод **ReceiveFrom()** передается массив байтов, в который надо считать данные. Метод **ReceiveFrom()** считывает данные в массив, возвращает значение, указывающее число успешно прочитанных данных, и фиксирует ту конечную точку удаленного узла, с которой были посланы данные (по этому адресу можно послать ответ).

Если размер буфера, переданный в метод **ReceiveFrom()**, слишком мал для приема всей дейтаграммы целиком, буфер заполняется теми данными, которые в него помещаются, а избыточные данные теряются.

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

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

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

Листинг 3.

```
using System;  
using System.Text;  
using System.Net;  
using System.Net.Sockets;  
  
namespace SocketSender  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Socket socket = new Socket(AddressFamily.InterNetwork,  
                                         SocketType.Dgram, ProtocolType.Udp);  
  
            // формируем сообщение  
            Console.Write("Введите сообщение: ");  
            string message = Console.ReadLine();  
  
            byte[] data = Encoding.Unicode.GetBytes(message);  
            EndPoint remoteEP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1111);  
            // отправляем сообщение  
            socket.SendTo(data, remoteEP); // в метод передается массив отправляемых  
данных, а также адрес, по которому эти данные надо отправить  
        }  
    }  
}
```

```

        // закрываем сокет
        socket.Shutdown(SocketShutdown.Both);
        socket.Close();
    }
}

```

Листинг 4.

```

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

namespace SocketRecipient
{
    class Program
    {
        static void Main(string[] args)
        {
            Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
ProtocolType.Udp);

            IPEndPoint localEP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1111);

            socket.Bind(localEP);

            Console.Write("Жду новых сообщений: ");

            // 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 remoteEP = new IPEndPoint(IPAddress.Any, 0);

            do
            {
                bytes = socket.ReceiveFrom(data, ref remoteEP);
                builder.Append(Encoding.Unicode.GetString(data, 0, bytes));
            }
            while (socket.Available > 0);

            // получаем данные о подключении
            IPEndPoint fullRemoteEP = remoteEP as IPEndPoint;

            // выводим сообщение
            Console.WriteLine("{0}:{1}-{2}", fullRemoteEP.Address.ToString(),
fullRemoteEP.Port, builder.ToString());

            // закрываем сокет

```

```
        socket.Shutdown(SocketShutdown.Both);  
        socket.Close();  
  
        Console.ReadLine();  
    }  
}  
}
```

**Задание 1.** Руководствуясь описанным в лабораторной работе примером создания клиент-серверного приложения на потоковом сокете TCP выполните следующие задания:

1. В примере клиентская программа отправляет лишь одно сообщение серверу и завершает работу. Необходимо доработать пример, реализовав отправку нескольких сообщений.
2. На сервере необходимо предусмотреть возможность автоответа, то есть автоматическое отправление клиентскому приложению ответа на принятое сообщение.
3. Организовать взаимодействие типа клиент-сервер. Клиент делает запрос серверу на выполнение какой-либо команды. Сервер выполняет эту команду и возвращает результаты клиенту.

**Задание 2.** Изучите код приложений «Отправитель» и «Получатель». Сравните использование сокетов при работе с протоколами TCP и UDP.

1. Программа «Отправитель» отправляет лишь одно сообщение, а программа «Получатель» принимает это сообщение и завершает работу. Необходимо реализовать отправку и получение нескольких сообщений.
2. Добавьте возможность ответа на сообщения, используя данные полученной удаленной точки (адрес и порт).
3. Объедините функции отправки и получения сообщений в одном приложении. Запустите две копии приложения. Протестируйте работу.

### **КОНТРОЛЬНЫЕ ВОПРОСЫ:**

1. Что такое сокет?
2. Какие бывают сокеты, в чем их особенности?
3. Особенность приложения клиент – сервер, основанного на потоковом сокете?
4. Алгоритм установления связи между клиентом и сервером для взаимодействия на основе потокового сокета.
5. Алгоритм прослушивания портов на локальном компьютере.
6. Методы и свойства класса Socket.
7. Как завершить соединение между клиентом и сервером?
8. Какие сокеты участвуют при взаимодействии приложений?

**9.** Как осуществляется прием и отправка данных между клиентом и сервером?

### **Литература**

1. Кумар В., Кровчик Э и др. .NET. Сетевое программирование / Пер. с англ. –М.: «Лори», 2007, стр.112-137.

## Лабораторная работа №2. Классы TcpClient, TcpListener

**Цель работы.** Изучить технологию создания сетевых приложений используя высокоуровневые сетевые классы, предоставляемые средой .NET Framework.

### Задачи:

1. изучить протокол TCP, его архитектуру и структуру данных;
2. рассмотреть классы TcpClient и TcpListener;
3. изучить пример сетевого приложения;
4. разработать собственное клиент-серверное приложение.

**Протокол TCP** или **Transmission Control Protocol**, используется как надежный протокол, обеспечивающий взаимодействие через взаимосвязанную сеть компьютеров. TCP проверяет, что данные доставляются по назначению и правильно.

**TCP** – это ориентированный на соединения протокол, предназначенный для обеспечения надежной передачи данных между процессами, выполняемыми или на одном и том же компьютере или на разных компьютерах. Термин *«ориентированный на соединения»* означает, что два процесса или приложения, прежде чем обмениваться какими-либо данными должны установить TCP-соединение. В этом TCP отличается от протокола UDP, являющегося протоколом *«без организации соединения»*, позволяющим выполнять широковещательную передачу данных неопределенному числу клиентов.

Когда приложение отправляет данные, используя TCP, они перемещаются вниз по стеку протоколов. Данные проходят по всем уровням и в конце концов передаются через сеть как поток битов. Каждый уровень в наборе протоколов TCP/IP добавляет к данным некоторую информацию в форме заголовков.

Когда пакет прибывает на конечный узел в сети, он снова проходит через все уровни снизу доверху. Каждый уровень проверяет данные, отделяя от пакета свою информацию в заголовке и наконец данные достигают серверного приложения в той же самой форме, в какой они покинули приложение-клиент.

TCP передает данные порциями, которые называются сегментами. Чтобы гарантировать правильное и в должном порядке получение сегментов, каждому из них назначается порядковый номер. Получатель отправляет подтверждение получения сегмента. Если подтверждение не получено до истечения интервала (тайм-аута), данные отправляются еще раз. Каждому октету (восьми битам) данных назначается порядковый номер. Порядковый

номер сегмента равен порядковому номеру первого октета данных в сегменте и это число отправляется в заголовке TCP данного сегмента.

TCP использует порядковые номера, чтобы гарантировать, что дублирующие данные получающему приложению переданы не будут и данные будут доставлены в правильном порядке. Заголовок TCP содержит контрольную сумму, чтобы гарантировать корректность данных при доставке. Если получен сегмент с неверной контрольной суммой, он просто отбрасывается, и подтверждение не отправляется. Это означает, что, когда значение тайм-аута истечет, отправитель повторит передачу сегмента.

TCP управляет объемом направляемых ему данных, возвращая с каждым подтверждением «размер окна». «Окно» – это объем данных, который может принять получатель. Между прикладной программой и потоком данных в сети располагается буфер данных. «Размер окна» фактически представляет собой разность между размером буфера и объемом сохраненных в нем данных. Это число отправляется в заголовке, чтобы информировать удаленный хост о текущем размере окна. Такой прием называется «скользящим окном» (*«Sliding Window»*).

Полученные данные сохраняются в этом буфере, и приложение может обращаться к буферу и считывать из него данные со свойственной ему скоростью. По мере того как приложение считывает данные, буфер опустошается и может принимать следующие данные, поступающие из сети.

Если приложение считывает данные из буфера слишком медленно, размер окна падает до нуля, и удаленный хост получает команду прекратить передачу данных. Как только локальное приложение обработает данные в буфере, размер окна возрастает и поступление данных из сети возобновляется. Если размер окна больше размера пакета, отправитель знает, что получатель может хранить одновременно несколько пакетов, что повышает производительность.

TCP дает возможность нескольким процессам на одной машине одновременно использовать сокет TCP. Сокет TCP состоит из адреса хоста и уникального номера порта, а TCP-соединение включает два сокета на разных концах сети. Порт может использоваться для нескольких соединений одновременно – один сокет на одном конце может использоваться для нескольких соединений с разными сокетами на другом конце. Примером этой ситуации служит Web-сервер, слушающий порт 80 и отвечающий на запросы от нескольких компьютеров.

Поддержка сокетов TCP на платформе .NET значительно усовершенствована по сравнению с предыдущей моделью программирования. Раньше большинство разработчиков, использовавших Visual C++, для реализации любых типов взаимодействия сокетов, обращались к классам CSocket и CAsyncSocket или пользовались библиотеками независимых поставщиков.

Для высокоуровневого программирования TCP встроенная поддержка практически отсутствовала. В .NET для работы с сокетами предоставлено особое пространство имен System.Net.Sockets. Это пространство имен содержит не только такие низкоуровневые классы, как Socket, но и классы высокого уровня **TcpClient** и **TcpListener**, предлагающие простые интерфейсы для взаимодействия через TCP.

В отличие от класса Socket, в котором для отправки и получения данных применяется побайтовый подход, классы TcpClient и TcpListener придерживаются потоковой модели. В этих классах все взаимодействие между клиентом и сервером базируется на потоке с использованием класса NetworkStream. Однако при необходимости можно работать с байтами.

Класс TcpClient обеспечивает TCP-сервисы для соединений на стороне клиента. Он построен на классе Socket и обеспечивает TCP-сервисы на более высоком уровне – в классе TcpClient есть закрытый объект данных m\_ClientSocket, используемый для взаимодействия с сервером TCP. Класс TcpClient предоставляет простые методы для соединения через сеть с другим приложением сокетов, отправки ему данных и получения данных от него. Наиболее важные члены класса TcpClient перечислены далее:

Свойство или метод	Тип	Описание
ReceiveBufferSize()	int	Задаёт размер буфера для входящих данных (в байтах). Это свойство используется при считывании данных из сокета.
ReceiveTimeout()	int	Задаёт время в миллисекундах, которое TcpClient будет ждать получения данных после инициирования этой операции. Если это время истечёт, а данные не будут получены, возникнет исключение SocketException.
SendBufferSize()	int	Задаёт размер буфера для исходящих данных.
SendTimeout()	int	Задаёт время в миллисекундах, которое TcpClient будет ждать подтверждения числа байтов, отправленных удалённому хосту от базового сокета. При истечении времени SendTimeout порождается исключение SocketException.
Close()	int	Закрывает TCP-соединение.
Connect()	int	Соединяется с удалённым хостом TCP.
GetStream()	int	Возвращает объект NetworkStream, используемый для передачи данных между клиентом и удалённым хостом.
Active	bool	Указывает, есть ли активное соединение с удалённым хостом.

## Создание экземпляра класса TcpClient

В классе TcpClient существуют три перегруженных конструктора:

```
public TcpClient();  
public TcpClient(IPEndPoint ipEndPoint);  
public TcpClient(string hostName, int port);
```

Конструктор, используемый по умолчанию, инициализирует экземпляр TcpClient. Для установления соединения с удаленным хостом надо вызвать метод Connect().

Второй перегруженный конструктор принимает один параметр типа IPEndPoint. Он инициализирует новый экземпляр класса TcpClient, связанный с указанной конечной точкой. Заметьте, что это не удаленная, а локальная конечная точка. Если попытаться передать конструктору удаленную конечную точку, будет порождено исключение, означающее, что в данном контексте IP-адрес задан некорректно. Если использовать этот конструктор, то после создания объекта TcpClient все-таки нужно вызвать метод Connect():

```
// Создаем локальную конечную точку  
IPAddress localAddr = IPAddress.Parse("127.0.0.1");  
IPEndPoint localEndPoint = new IPEndPoint(localAddr, 5001);  
  
TcpClient client = new TcpClient(localEndPoint);  
  
// Для создания соединения с сервером надо вызвать Connect()  
client.Connect(192.168.1.52, 5002);
```

Параметр, переданный конструктору объекта TcpClient, является локальной конечной точкой, в то время как метод Connect() фактически соединяет клиента с сервером и поэтому принимает в качестве параметра удаленную конечную точку.

Последний перегруженный конструктор создает новый экземпляр класса TcpClient и устанавливает удаленное соединение с использованием в параметрах DNS-имени и номера порта:

```
TcpClient client = new TcpClient("localhost", 80);
```

Это самый удобный метод, он позволяет инициализировать TcpClient, разрешить DNS-имя и соединиться с хостом в одном простом шаге. Однако заметьте, что с помощью этого конструктора нельзя задать локальный порт, с которым желательно связаться.

## Установка соединения с хостом

Создав экземпляр класса TcpClient, следующим шагом установим соединение с удаленным хостом. Для соединения клиента с хостом TCP предоставлен метод Connect(). Если для создания экземпляра TcpClient использовать конструктор по умолчанию или локальную конечную точку, то останется лишь вызвать этот метод, иначе, если конструктору были переданы



имя хоста и номер порта, попытка вызова метода `Connect()` породит исключение. Существуют три перегруженных метода `Connect()`:

1. `public void Connect(IPEndPoint endPoint);`
2. `public void Connect(IPAddress ipAddr, int port);`
3. `public void Connect(string hostname, int port);`

Они достаточно просты, но, тем не менее, на коротких примерах продемонстрируем использование каждого перегруженного метода:

1. Передача объекта `IPEndPoint`, представляющего удаленную конечную точку, с которой надо соединиться:

```
// Создаем новый экземпляр TcpClient
TcpClient client = new TcpClient();

// Устанавливаем соединение с IPEndPoint
IPAddress ipAddr = IPAddress.Parse("127.0.0.1");
IPEndPoint endPoint = new IPEndPoint(ipAddr, 80);

// Соединяемся с хостом
client.Connect(endPoint);
```

2. Передача объекта `IPAddress` и номера порта:

```
// Создаем новый экземпляр TcpClient
TcpClient client = new TcpClient();

// Устанавливаем соединение с IPEndPoint
IPAddress ipAddr = IPAddress.Parse("127.0.0.1");

// Соединяемся с хостом
client.Connect(ipAddr, 80);
```

3. Передача имени хоста и номера порта:

```
// Создаем новый экземпляр TcpClient
TcpClient client = new TcpClient();

// Соединяемся с хостом
client.Connect("127.0.0.1", 80);
```

Если соединение будет неудачным или возникнут другие проблемы, порождается исключение `SocketException`:

```
try
{
    TcpClient client = new TcpClient();

    // Соединяемся с сервером
    client.Connect("192.168.0.1", 80);    // В этот момент сокет
                                         // порождает исключение, если
                                         // при соединении возникают проблемы
}
catch (SocketException ex)
{
    Console.WriteLine("Exception: " + ex.ToString());
}
```

## Отправка и получение сообщений

Для обработки на уровне потока, как канал между двумя соединенными приложениями, используется класс `NetworkStream`. Прежде чем отправлять и получать любые данные, нужно определить базовый поток. Класс `TcpClient` предоставляет метод `GetStream()` исключительно для этих целей. С помощью базового сокета он создает экземпляр класса `NetworkStream` и возвращает его вызывающей программе. Следующий пример кода демонстрирует, как получить сетевой поток через метод `GetStream()`.

Предположим, что `client` – это экземпляр `TcpClient`, а соединение с хостом уже установлено. Иначе будет порождено исключение `InvalidOperationException`.

```
NetworkStream tcpStream = client.GetStream();
```

Получив поток, используем методы `Read()` и `Write()` класса `NetworkStream` для чтения из приложения хоста и записи к нему.

Метод `Write()` принимает три параметра: массив байтов, содержащий данные, которые надо отправить хосту, позицию в потоке, с которой хотим начать запись, и длину данных:

```
byte[] sendBytes = Encoding.UTF8.GetBytes("Тест");  
tcpStream.Write(sendBytes, 0, sendBytes.Length);
```

Метод `Read()` имеет точно такой же набор параметров – массив байтов для сохранения данных, которые считываются из потока, позицию начала считывания и число считываемых байтов:

```
byte[] bytes = new byte[client.ReceiveBufferSize];  
int bytesRead = tcpStream.Read(bytes, 0, client.ReceiveBufferSize);  
  
// Строка, содержит все поступившие данные от сокета  
string returnData = Encoding.UTF8.GetString(bytes);
```

Свойство `ReceiveBufferSize` класса `TcpClient` позволяет получить или установить размер (в байтах) буфера для чтения, поэтому используем его как размер массива байтов. Заметьте, что, устанавливая это свойство, мы не ограничиваем число байтов, которое можно считывать каждой операцией, поскольку при необходимости размер буфера будет динамически изменяться, но если задать размер буфера, это сократит накладные расходы.

## Заккрытие сокета TCP

После взаимодействия с клиентом, чтобы освободить все ресурсы, следует вызвать метод `Close()`:

```
// Закрываем клиентский сокет  
client.Close();
```

Вот и все, что нужно, чтобы использовать класс `TcpClient` для взаимодействия с сервером.

Помимо этой основной функциональности имеются другие возможности. Если требуется обратиться к экземпляру сокета, базовому для объекта `TcpClient`, например для установки опций методом `SetSocketOption()`, можно использовать свойство `Client`, получая доступ к членам соответствующего объекта `Socket`. Можно также использовать свойство `Client`, чтобы сделать существующий объект `Socket` базовым сокетом для объекта `TcpClient`. Но поскольку это защищенный член класса `TcpClient`, прежде чем его использовать, наш класс должен наследовать класс `TcpClient`.

Свойство `Client` дает возможность защищенного доступа к закрытому члену `m_ClientSocket`, о котором упоминалось ранее. Класс `TcpClient` передает сделанные на нем вызовы аналогичному методу класса `Socket` после проверки параметров и инициализации экземпляра сокета. Объект `m_ClientSocket` создается в конструкторе, который вызывает закрытый метод `initialize()`, строящий новый объект `Socket`, и затем вызывает метод `set_Client()`, чтобы назначить его свойству `Client`. Этот метод также устанавливает булево значение `m_Active`, используемое для отслеживания состояния экземпляра `Socket`. Он также проверяет наличие излишних соединений объекта `Socket` и операций, требующих установления соединения.

В общем у сокетов есть масса опций, которые класс `TcpClient` не охватывает. Если нужно установить или получить какое-либо из этих свойств, не представленных в `TcpClient` (например, `Broadcast` или `KeepAlive`), необходимо унаследовать класс от `TcpClient` и использовать его член `Client`.

## Класс `TcpListener`

Обычно приложение стороны сервера начинает работу, связываясь с локальной конечной точкой и ожидает входящие запросы от клиентов. Как только клиент достиг порта, приложение активизируется, принимает запрос и создает канал, предназначенный для взаимодействия с этим клиентом. На основном потоке приложение продолжает ожидать другие входящие запросы от клиентов.

Класс `TcpListener` делает именно это – он слушает запросы клиентов, принимает запрос и создает новый экземпляр класса `Socket` или класса `TcpClient`, которые можно использовать для взаимодействия с клиентом. Как и `TcpClient`, класс `TcpListener` также инкапсулирует закрытый объект `Socket` – `m_ServerSocket`, доступный только для производных классов.

В следующей таблице показаны важные свойства и методы класса `TcpListener`:

Свойство или метод	Тип	Описание
<i>LocalEndpoint</i>	<code>IPEndpoint</code>	Это свойство возвращает объект <code>IPEndpoint</code> , который содержит информацию о локальном сетевом интерфейсе и номере порта, используемую для

		ожидания входящих запросов от клиентов.
<i>AcceptSocket()</i>		Дает согласие на ожидающий запрос соединения и возвращает объект <code>Socket</code> , используемый для взаимодействия с клиентом.
<i>AcceptTcpClient()</i>		Дает согласие на ожидающий запрос соединения и возвращает объект <code>TcpClient</code> , используемый для взаимодействия с клиентом.
<i>Pending()</i>		Указывает, есть ли ожидающие запросы соединения.
<i>Start()</i>		Принуждает <code>TcpListener</code> начать слушать запросы соединения.
<i>Stop()</i>		Закрывает слушающий объект.
<i>Active</i>	<code>bool</code>	Указывает, слушает ли в настоящий момент <code>TcpListener</code> запросы соединения.
<i>Server</i>	<code>Socket</code>	Возвращает базовый объект <code>Socket</code> , используемый объектом <code>TcpListener</code> , чтобы слушать запросы соединения.

## Создание экземпляра класса `TcpListener`

Существуют три перегруженных конструктора `TcpListener`:

```
public TcpListener(int port);
public TcpListener(IPEndPoint endPoint);
public TcpListener(IPAddress ipAddr, int port);
```

Первый конструктор просто указывает, какой порт используется, чтобы слушать запросы. В этом случае IP-адрес равен `IPAddress.Any`, т.е. сервер принимает действия клиентов на всех сетевых интерфейсах. Это значение эквивалентно IP-адресу `0.0.0.0`.

```
int port = 5001;
TcpListener newListener = new TcpListener(port);
```

Второй конструктор принимает объект `IPEndPoint`, определяющий IP-адрес и порт, на котором хотим слушать:

```
IPAddress ipAddr = IPAddress.Parse("127.0.0.1");
IPEndPoint endPoint = new IPEndPoint(ipAddr, 11000);
TcpListener newListener = new TcpListener (endPoint);
```

Последний перегруженный конструктор принимает объект `IPAddress` и номер порта:

```
IPAddress ipAddr = IPAddress.Parse("127.0.0.1");  
int port = 5001;  
TcpListener newListener = new TcpListener(ipAddr, port);
```

### Прослушивание клиентов

На следующем шаге после создания сокета начинаем слушать запросы клиентов. В классе `TcpListener` есть метод `Start()` выполняющий такую последовательность. Сначала он связывает сокет, используя IP-адрес и порт, переданные в параметрах конструктору `TcpListener`. Затем, вызвав метод `Listen()` базового объекта `Socket`, он начинает слушать запросы соединений от клиентов:

```
IPAddress ipAddr = IPAddress.Parse("127.0.0.1");  
TcpListener newListener = new TcpListener(ipAddr, 11000);  
Listener.Start();
```

Уже приступив к прослушиванию сокета, можно вызывать метод `Pending()`, чтобы проверять, нет ли ожидающих запросов соединения в очереди. Этот метод позволяет проверять наличие ожидающих клиентов до вызова метода `Accept`, который будет блокировать выполняющийся поток:

```
...  
if (newListener.Pending())  
{  
    Console.WriteLine("Соединение добавлено в очередь");  
}
```

### Прием соединений от клиентов

Типичная серверная программа оперирует двумя сокетами: один используется классом `TcpListener`, а второй — для взаимодействия с отдельным клиентом. Чтобы дать согласие на любой запрос, ожидающий в настоящий момент в очереди, можно воспользоваться методом `AcceptSocket()` или более простым методом `AcceptTcpClient()`. Эти методы возвращают соответственно объекты `Socket` или `TcpClient` и дают согласие на запросы клиентов.

```
Socket sAccepted = newListener.AcceptSocket();  
TcpClient sAccepted = newListener.AcceptTcpClient();
```

### Отправка и получение сообщений

В зависимости от типа сокета, созданного при установлении соединения, реальный обмен данными между клиентом и сокетом сервера

выполняется методами `Send()` и `Receive()` объекта `Socket` или с помощью чтения-записи объекта `NetworkStream`.

### Остановка сервера

После завершения взаимодействия с клиентом нужно выполнить последний шаг – остановить слушающий сокет. Для этого вызывается метод `Stop()` объекта `TcpListener`:

```
listener.Stop();
```

### Пример сетевого клиент-серверного приложения. ТСП-клиент

Листинг 1

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

namespace SimpleTcpClient
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // Создаем клиента, используя конструктор по умолчанию
                TcpClient tcpClient = new TcpClient();

                // Подключаемся к серверу
                tcpClient.Connect("127.0.0.1", 5001);

                // Создаем поток, соединенный с сервером
                NetworkStream stream = tcpClient.GetStream();

                // Формируем сообщение. Преобразуем его в массив байтов
                byte[] data = Encoding.UTF8.GetBytes("Ваше сообщение или запрос");

                // Отправка сообщения
                stream.Write(data, 0, data.Length);

                // Закрываем потоки
                stream.Close();
                tcpClient.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.ToString());
            }
            finally
            {
            }
        }
    }
}
```

```
}
```

## TCP-сервер

Листинг 2

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

namespace SimpleTcpListener
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                IPAddress localAddr = IPAddress.Parse("127.0.0.1");
                TcpListener server = new TcpListener(localAddr, 5001);

                // Запускаем сервер
                server.Start();
                Console.WriteLine("Ожидание подключений... ");

                // Получаем входящее подключение
                TcpClient client = server.AcceptTcpClient();

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

                byte[] data = new byte[256];
                int bytes = stream.Read(data, 0, data.Length); // Читаем сообщение
                // Отображаем сообщение
                Console.WriteLine(Encoding.UTF8.GetString(data, 0, bytes));

                stream.Close();
                client.Close();

                // Закрываем слушающий объект
                server.Stop();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                Console.ReadLine();
            }
        }
    }
}
```

### Задание:

1. Изучите примеры листингов приведенных выше.
2. На основе примеров, напишите простое сетевое приложения эхо-сервер, функционирующий следующим образом:
  - клиент считывает строку текста из стандартного потока ввода и отправляет ее серверу;
  - сервер считывает строку из сети и отсылает эту строку обратно клиенту;
  - клиент считывает отраженную строку и помещает ее в свой стандартный поток вывода.

*Соединение клиент-сервер, отражающее вводимые строки, является корректным и в то же время простым примером сетевого приложения. На этом примере можно проиллюстрировать все основные действия, необходимые для реализации соединения клиент-сервер. Все, что Вам нужно сделать, чтобы применить его к вашему приложению, - это изменить операции, которые выполняет сервер с принимаемыми от клиентов данными.*

3. Измените сервер так, чтобы он, по-прежнему принимая текстовую строку от клиента, предполагал, что строка содержит два целых числа, разделенных пробелом, и возвращал сумму этих чисел.

### Литература

1. Кумар В., Кровчик Э и др. .NET. Сетевое программирование / Пер. с англ. –М.: «Лори», 2007, стр.



## Лабораторная работа №3

### Создание параллельного многопоточного сервера с установлением логического соединения TCP

**ЦЕЛЬ РАБОТЫ:** изучить методы создания серверных приложений на основе установления логического соединения TCP, используя алгоритм многопоточной обработки запросов.

В предыдущих лабораторных работах были показаны примеры реализации последовательных серверов как с установлением, так и без установления логического соединения. В данной лабораторной работе рассматривается пример параллельного сервера с установлением логического соединения.

**Многопоточность** – это специализированная форма многозадачности (multitasking). Что касается многозадачности, то выделяют два типа многозадачности: основанную на процессах (process-based) и основанную на потоках (thread-based). По сути, процесс (process) – это отдельно выполняющаяся программа. Таким образом, основанная на процессах многозадачность – средство, позволяющее вашему компьютеру выполнять несколько программ одновременно. Отличия основанной на процессах и многопоточной многозадачности можно сформулировать следующим образом: первая поддерживает одновременное выполнение нескольких программ, а вторая имеет дело с одновременным выполнением разных фрагментов одной и той же программы. С помощью процессов можно организовать параллельное выполнение программ. Для этого процессы клонируются с помощью вызовов функции `fork()` или функции `exec()`, а затем между ними (процессами) организуется взаимодействие средствами IPC. Это довольно дорогостоящий с точки зрения ресурсов процесс.

С другой стороны, для организации параллельного выполнения и взаимодействия части программы можно использовать механизм многопоточности. Основной единицей здесь является поток. Рассмотрим этот механизм подробнее.

#### Потоки

Последовательная реализация сервера, о которой речь шла в предыдущих лабораторных работах, может оказаться неподходящей, поскольку клиенты будут вынуждены ждать завершения обработки всех предыдущих запросов на установление соединения. Если клиент решит передать большие объемы данных (например, несколько мегабайт), последовательный сервер отложит обслуживание всех других клиентов до тех пор, пока не выполнит этот запрос.

Параллельная реализация сервера дает возможность обойтись без продолжительных задержек, так как не позволяет одному клиенту захватить все ресурсы. Вместо этого параллельный сервер поддерживает обмен данными сразу с несколькими клиентами для того, чтобы их запросы обрабатывались одновременно.

Поэтому, с точки зрения клиента, параллельный сервер обеспечивает лучшее наблюдаемое время отклика по сравнению с последовательным сервером. Распараллеливание обработки на сервере достигается созданием отдельного потока для обработки запросов одного клиента или отдельного однопотокового процесса для обработки запросов одного клиента.

**Поток** (thread) – это управляемая единица исполняемого кода. У всех процессов обязательно есть один поток, но их может быть и больше. Это означает, что в одной программе могут выполняться несколько задач одновременно.

#### Преимущества многопоточности

Если операционная система поддерживает концепции потоков в рамках одного процесса, она называется многопоточной. Многопоточные приложения имеют ряд преимуществ:

а) улучшенная реакция приложения – любая программа, содержащая много не зависящих друг от друга действий, может быть перепроектирована так, чтобы каждое действие выполнялось в отдельном потоке. Например, пользователь многопоточного интерфейса не должен ждать завершения одной задачи, чтобы начать выполнение другой;

б) более эффективное использование мультипроцессорирования – как правило, приложения, реализующие параллелизм через потоки, не должны учитывать число доступных процессоров. Производительность приложения равномерно увеличивается при наличии дополнительных процессоров. Численные алгоритмы и приложения с высокой степенью параллелизма, например перемножение матриц, могут выполняться намного быстрее;

в) улучшенная структура программы – некоторые программы более эффективно представляются в виде нескольких независимых или полуавтономных единиц, чем в виде единой монолитной программы. Многопоточные программы легче адаптировать к изменениям требований пользователя;

г) эффективное использование ресурсов системы – программы, использующие два или более процессов, которые имеют доступ к общим данным через разделяемую память, содержат более одного потока управления. При этом каждый процесс имеет полное адресное пространство и состояние в операционной системе. Стоимость создания и поддержания большого количества служебной информации делает каждый процесс более затратным, чем поток. Кроме того, разделение работы между процессами может потребовать от программиста значительных усилий, чтобы обеспечить связь между потоками в различных процессах или синхронизировать их действия.

### **Преимущества и недостатки многопоточковых процессов**

Многопоточковые процессы обладают двумя основными преимуществами по сравнению с однопоточковыми процессами: более высокая эффективность и разделяемая память. Повышение эффективности связано с уменьшением издержек на переключение контекста. Переключателем контекста называются действия, выполняемые операционной системой при передаче ресурсов процессора от одного потока выполнения к другому. При переключении с одного потока на другой операционная система должна сохранить в памяти состояние предыдущего потока (например, значение регистров) и восстановить состояние следующего потока. Потоки в одном и том же процессе разделяют значительную часть информации о состоянии процесса, поэтому операционной системе приходится выполнять меньший объем работы по сохранению и восстановлению состояния. Вследствие этого переключение с одного потока на другой в одном и том же процессе происходит быстрее по сравнению с переключением между двумя потоками в разных процессах. В частности, поскольку потоки одного и того же процесса разделяют адресное пространство памяти, то переключение между потоками процесса означает, что операционная система не должна менять отображение виртуальной памяти на физическую. Второе преимущество потоков, т.е. разделяемая память, вероятно, является для программистов еще более важным, чем повышение эффективности. Потоки упрощают разработку параллельных серверов, в которых все копии сервера должны взаимодействовать друг с другом или обращаться к разделяемым элементам данных. Кроме того, потоки упрощают разработку систем контроля и управления. В частности, поскольку ведомые потоки в сервере совместно используют память, они могут записывать в глобальную память статистическую информацию, что позволяет контролирующему потоку формировать отчеты об активности ведомых потоков сервера для системного администратора. Хотя потоки имеют свои преимущества над однопоточковыми

процессами, они не лишены также определенных недостатков. Один из наиболее важных недостатков связан с тем, что потоки не только разделяют память, но и имеют общее состояние процесса, поэтому действия, выполненные одним потоком, могут повлиять на другие потоки в том же процессе. Например, если два потока попытаются одновременно обратиться к одной и той же переменной, они могут помешать друг другу. API-интерфейс потоков предоставляет функции, которые могут использоваться потоками для координации работы. Однако многие библиотечные функции, возвращающие указатели на статические элементы данных, не являются безопасными с точки зрения потоков, а это означает, что результаты вызова таких функций могут оказаться непредсказуемыми. Еще один недостаток потоков (и отличие однопоточного процессора от многопоточного) связан с отсутствием надежности. Если одна из параллельно работающих копий однопоточного сервера вызовет серьезную ошибку (например, в ней будет выполнена ссылка на недопустимую область памяти), то операционная система завершит только тот процесс, который вызвал ошибку. С другой стороны, если серьезная ошибка будет вызвана одним из потоков многопоточного сервера, то операционная система завершит весь процесс.

### **Алгоритм работы параллельного (многопоточного) сервера с установлением логического соединения.**

Приведем обобщенный алгоритм работы параллельного сервера с установлением логического соединения:

1. Ведущий поток. Создать сокет и выполнить его привязку к локальному адресу. Оставить сокет неподключенным.
2. Ведущий поток. Перевести сокет в пассивный режим, подготовив его для использования сервером.
3. Ведущий поток. Вызывать в цикле функцию `Accept()` для получения очередного запроса от клиента и создавать новый ведомый поток или процесс для формирования ответа.
4. Ведомый поток. Работа потока начинается с получения доступа к соединению, полученному от ведущего потока (т.е. к сокету соединения).
5. Ведомый поток. Выполнять обмен данными с клиентом через соединение: принимать запрос (запросы) и передавать ответ (ответы).
6. Ведомый поток. Закрыть соединение и завершить работу. Ведомый поток завершает свою работу после обработки всех запросов от одного клиента. Сервер функционирует неопределенно долгое время, ожидая поступления новых запросов на установление соединения от клиентов. Его ведущий поток при подключении клиента создает новый ведомый поток для обработки запросов каждого нового соединения и предоставляет каждому ведомому потоку возможность взять на себя весь обмен данными с клиентом.

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ** по созданию параллельного многопоточного сервера с установлением логического соединения ТСП.

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

Создадим проект для клиента:

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

```

namespace ConsoleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            TcpClient client = null;
            try
            {
                client = new TcpClient("127.0.0.1", 1234);
                NetworkStream stream = client.GetStream();

                while (true)
                {
                    // Запрос на ввод сообщения
                    Console.Write("Введите сообщение: ");
                    string message = Console.ReadLine();

                    // Преобразуем сообщение в массив байтов
                    byte[] data = Encoding.Unicode.GetBytes(message);

                    // Отправляем сообщение
                    stream.Write(data, 0, data.Length);

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

                    message = response.ToString();
                    Console.WriteLine("Ответ сервера: {0}", message);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                client.Close();
            }
        }
    }
}

```

Создадим проект сервера. В проект сервера добавим новый класс ClientObject, который будет представлять отдельное подключение:

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

                string message = response.ToString();

                Console.WriteLine(message);

                // Отправляем сообщение обратно
                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 отправляем его обратно клиенту. То есть класс `ClientObject` заключает в себе все действия по работе с отдельным подключением.

В главном классе проекта сервера определим следующий код:

```
static TcpListener listener;
static void Main(string[] args)
{
    try
    {
        listener = new TcpListener(IPAddress.Parse("127.0.0.1"), 1234);
        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();
    }
}
```

### ЗАДАНИЕ

Разработать приложение, реализующее архитектуру «клиент-сервер». Реализовать параллельный многопоточный сервер с установлением логического соединения (TCP). Логика взаимодействия клиента и сервера реализовать следующим образом.

На сервере хранится список студентов. Каждая запись списка содержит следующую информацию:

1. ФИО студента;
2. номер группы.

Таких записей должно быть не менее пяти. Клиент вводит с клавиатуры букву алфавита, по которой он хотел бы посмотреть информацию о студентах, и посылает ее на сервер. Назад он получает список только тех студентов, фамилии которых начинаются на эту букву. Аналогично с номером группы.

## Лабораторная работа № 4

### Порядок выполнения работы

1. Консольный TCP-чат
  - 1.1 Класс ClientObject
  - 1.2 Класс ServerObject
  - 1.3 Класс Program
  - 1.4 Изменение стандартного класса Program
2. Задание

### 1. Консольный TCP-чат

Создадим консольный проект сервера, который назовем ChatServer. В этот проект добавим два новых класса **ClientObject** и **ServerObject**.

#### 1.1 Класс ClientObject

##### Листинг 1

```
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`.

## 1.2 Класс `ServerObject`

### Листинг 2

```

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.

### 1.3 Изменим класс Program

#### Листинг 3

```

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.

#### **1. 4 Изменим его стандартный класс Program следующим образом**

##### **Листинг 4**

```

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
```

## 2. Задание

1. Напишите консольный чат для двух удаленных клиентов с использованием протокола TCP.

## Лабораторная работа № 5. Класс `UdpClient`

**Цель работы.** Реализация протокола UDP в .NET с использованием класса `UdpClient`.

### Задачи:

1. изучить основы протокола UDP;
2. рассмотреть класс `UdpClient`;
3. изучить пример сетевого приложения;
4. разработать сетевое приложение, с помощью класса `UdpClient`.

**UDP** (User Datagram Protocol) – это простой, ориентированный на дейтаграммы протокол без организации соединения, предоставляющий быстрое, но необязательно надежное транспортное обслуживание. Он поддерживает взаимодействия «один со многими» и поэтому часто применяется для широковещательной и групповой передачи дейтаграмм.

### Класс `UdpClient`

Среда Microsoft .NET Framework предоставляет класс `UdpClient` для реализации в сети протокола UDP. Как и классы `TcpClient` и `TcpListener`, этот класс построен на классе `Socket`, но скрывает излишние члены, которые не требуются для реализации приложения, базирующегося на UDP.

Применять класс `UdpClient` довольно просто. Во-первых, создайте экземпляр `UdpClient`. Далее через вызов метода `Connect()` соединитесь с удаленным хостом. Эти два шага можно сделать в одной строке, если указать в конструкторе `UdpClient` удаленный IP-адрес и удаленный номер порта. Ранее было сказано, что протокол UDP не ориентирован на установление соединений, поэтому может возникнуть вопрос: так зачем же этот `Connect`? В действительности метод `Connect()` до отправки или получения данных не устанавливает соединение с удаленным хостом. Когда вы отправляете дейтаграмму, то пункт назначения должен быть известен, для этого нужно указать IP-адрес и номер порта.

Третий шаг состоит в отправке и получении данных с использованием метода `Send()` или `Receive()`. Наконец метод `Close()` закрывает соединение UDP.

### Создание экземпляра класса `UdpClient`

Экземпляр класса `UdpClient` можно создать несколькими способами, которые отличаются передаваемыми параметрами. Использование объекта `UdpClient` зависит от того, как он создавался.

Простейший способ состоит в вызове конструктора по умолчанию. Когда экземпляр класса создается таким образом, нужно вызвать метод



Connect() и установить соединение, или задать информацию о соединении при отправке данных.

Можно также создать объект `UdpClient`, указав в качестве параметра номер порта. В этом случае `UdpClient` будет слушать все локальные интерфейсы (т. е. использовать IP-адрес 0.0.0.0). Если номер порта находится вне пределов, указанных полями `MinPort` и `MaxPort` класса `IPEndPoint`, порождается исключение `ArgumentOutOfRangeException` (производное от `ArgumentException`). Если указанный порт уже занят, порождается исключение `SocketException`:

```
// Создаем UdpClient, используя номер порта
try
{
    UdpClient udpClient = new UdpClient(8001);
}
catch (ArgumentOutOfRangeException ex)
{
    Console.WriteLine("Некорректный номер порта");
}
catch (SocketException ex)
{
    Console.WriteLine("Порт уже используется");
}
```

Следующий способ заключается в использовании объекта `IPEndPoint`, представляющего локальный IP-адрес и номер порта, которые хотим выбрать для соединения. В этом случае первый шаг состоит в создании экземпляра класса `IPEndPoint`, а это можно сделать, используя длинный IP-адрес или объект `IPAddress`. IP-адрес должен принадлежать одному из интерфейсов локальной машины, иначе будет порождено исключение `SocketException` с ошибкой «The requested address is not valid in it's context» (Запрошенный адрес в этом контексте неприменим).

Если конструктору передается пустой объект `IPEndPoint`, порождается исключение `ArgumentNullException`:

```
IPAddress ipAddr = IPAddress.Parse("127.0.0.1");
IPEndPoint ipEndPoint = new IPEndPoint(ipAddr, 5001);

// Создаем UdpClient, используя экземпляр IPEndPoint
try
{
    UdpClient udpClient = new UdpClient(ipEndPoint);
}
catch (ArgumentNullException ex)
{
    Console.WriteLine(ex.ToString());
}
```

Последний способ состоит в передаче конструктору имени хоста и номера порта. В этом случае конструктор инициализируется именем хоста и

номером порта удаленного хоста. Это позволяет исключить шаг с вызовом метода `Connect()`, поскольку он вызывается из конструктора.

```
// Создаем объект UdpClient, используя имя удаленного хоста и номер порта
try
{
    UdpClient udpClient = new UdpClient("remoteHostName",5001);
}
catch (ArgumentNullException ex)
{
    Console.WriteLine(ex.ToString());
}
```

## Определение информации о соединении

После создания объекта `UdpClient` переходим ко второму шагу – подготовке информации о соединении, которая будет использоваться, когда потребуется отправить данные удаленному хосту. Вспомните, что протокол UDP не ориентирован на соединения и эта информация не нужна для получения данных, она используется, только чтобы указать, куда мы хотим отправить данные.

Эту информацию можно указать в любом из следующих трех мест: как мы уже видели, ее можно задать в конструкторе `UdpClient`, можно явно вызвать метод `Connect()` класса `UdpClient` или ее можно включить в метод `Send()` при фактической передаче данных.

Существуют три перегруженных метода `Connect()`:

- использующий объект `EndPoint`;
- устанавливающий соединение, используя IP-адрес и номер порта удаленного хоста;
- использующий имя DNS или машины и номер порта удаленного хоста.

### 1. Использование объекта `EndPoint`

Первый перегруженный метод `Connect()` для соединения с удаленным хостом использует экземпляр класса `EndPoint`, поэтому перед вызовом метода `Connect()` создаем объект `EndPoint`.

```
Создаем объект UdpClient
UdpClient udpClient = new UdpClient();

Получаем IP-адрес удаленного хоста.
IPAddress ipAddress = IPAddress.Parse("224.56.0.1");

Создаем объект EndPoint, используя IPAddress и номер порта.
EndPoint endPoint = new EndPoint(ipAddress,1234);

try
{
```

```

        Соединяемся, используя объект IPEndPoint.
        udpClient.Connect(ipEndPoint);
    }
    catch (Exception e)
    {
        Console.WriteLine("Ошибка при подключении:" + e.ToString());
    }

```

## 2. Использование IPAddress и номера порта

Второй перегруженный метод принимает объект IPAddress и номер порта удаленного хоста. Если известны удаленный IP-адрес и удаленный UDP-порт, можно создать соединение с удаленным UDP-хостом следующим образом.

```

        Создаем объект UdpClient.
        UdpClient udpClient = new UdpClient();

        Получаем IP-адрес удаленного хоста.
        IPAddress ipAddress = IPAddress.Parse("224.56.0.1");

        try
        {
            Соединяемся, используя созданный объект IPAddress и удаленный порт.
            udpClient.Connect(ipAddress,1234);
        }
    }

```

## 3. Использование имени хоста и номера порта

В последнем перегруженном методе используется DNS-имя и номер порта удаленного хоста. Это довольно простой метод, поскольку не нужно создавать ни объект IPAddress, ни объект IPEndPoint.

```

        Создаем экземпляр класса UdpClient
        UdpClient udpClient = new UdpClient();

        try
        {
            Соединяемся, используя имя и номер порта удаленного хоста.
            udpClient.Connect("remoteHostName",1234);
        }
    }

```

## Отправка данных через объект UdpClient

Получив экземпляр класса UdpClient и подготовив (необязательную) информацию о соединении, можно приступить к отправке данных. Неудивительно, что для этого вызывается метод Send(), который используется, чтобы послать дейтаграмму от клиента удаленному хосту. Важный момент, характеризующий протокол UDP, состоит в том, что после отправки данных удаленному хосту он не получает никаких подтверждений. Как и метод Connect(), метод Send() представлен несколькими

перегруженными методами. `Send()` возвращает длину данных, которую можно использовать для проверки, правильно ли были отправлены данные.

Основная процедура отправки данных для класса `UdpClient` показана на следующей схеме:

Отправка данных с использованием UDP включает следующие четыре шага:

1. Создать экземпляр `UdpClient`.
2. Соединиться с удаленным хостом (необязательно).
3. Отправить данные.
4. Закрыть соединение.

#### **Пример отправки данных по UDP (общий процесс):**

```
IPAddress remoteAddress = IPAddress.Parse("127.0.0.1");
int remotePort = 5001;

try
{
    // Создаем объект UdpClient
    UdpClient sender = new UdpClient();

    // Соединяемся с удаленным хостом
    sender.Connect(remoteAddress, remotePort);

    // Посылаем данные удаленному хосту
    byte[] bytes = Encoding.UTF8.GetBytes("Test");
    sender.Send(bytes, bytes.Length);

    // Закрываем соединение
    sender.Close();
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

#### **Получение данных с использованием объекта `UdpClient`**

Естественно, что для получения данных от удаленного хоста через UDP вызывается метод `Receive()`. Этот метод принимает один ссылочный параметр, экземпляр класса `IPEndPoint`, и возвращает в массиве байтов принятые данные. Обычно рекомендуется выполнять этот метод в отдельном потоке, поскольку он опрашивает базовый сокет на предмет поступления дейтаграмм и блокирует поток, пока данные не будут получены. Если он выполняется в основном потоке, выполнение программы приостанавливается, пока не будет получен пакет дейтаграммы.

*Если объекту `UdpClient` уже указана информация о соединении в конструкторе или вызван метод `Connect()`, то метод `Receive()` будет принимать и возвращать нашему приложению только данные от указанной удаленной точки, а соединения от других источников будут отбрасываться. Если никакая информация о соединении не задавалась, будут приниматься все входящие соединения с локальной конечной точкой.*

После получения дейтаграммы этот метод возвращает данные в массиве байтов (удалив информацию заголовка) и заполняет объект `IPEndPoint`, на который ссылается параметр, информацией об удаленном хосте, отправившем данные. Процесс получения данных от удаленного хоста очень похож на отправку данных:

- создать экземпляр класса `UdpClient`;
- получить данные `Receive()`;
- закрыть `UdpClient` – `Close()`.

Все три шага показаны в следующем примере:

#### **Пример получения данных по UDP:**

```
try
{
    // Создаем объект UdpClient
    UdpClient receiving = new UdpClient(5001);

    // Создаем переменную IPEndPoint, чтобы передать ссылку на нее в Receive()
    IPEndPoint RemoteIPEndPoint = null;

    // Получение данных
    byte[] bytes = receiving.Receive(ref RemoteIPEndPoint);
    string returnData = Encoding.UTF8.GetString(bytes);

    Console.WriteLine(returnData); // Отображаем полученную информацию

    // Закрываем соединение
    receiving.Close();
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

#### **Пример приложения отправитель:**

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

namespace sender
{
    class Program
    {
```

```

static void Main(string[] args)
{
    UdpClient udpClient = new UdpClient();

    // Формируем сообщение
    byte[] sendBytes = Encoding.Unicode.GetBytes("Ваше сообщение");

    // Отправляем сообщение, указывая IP-адрес и номер порта получателя
    udpClient.Send(sendBytes, sendBytes.Length, "127.0.0.1",5001);

    udpClient.Close();
}
}
}

```

### Пример приложения получатель:

```

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

namespace receiver
{
    class Program
    {
        static void Main(string[] args)
        {
            // Указываем локальный порт для прослушивания входящих сообщений
            UdpClient receivingUdpClient = new UdpClient(5001);

            // Адрес входящего подключения
            IPEndPoint remoteIPEndPoint = null;

            try
            {
                while (true)
                {
                    // Получаем данные
                    byte[] receiveBytes = receivingUdpClient.Receive(ref remoteIPEndPoint);
                    string returnData = Encoding.Unicode.GetString(receiveBytes);
                    Console.WriteLine("Входящее сообщение: {0}, адрес входящего
подключения: {1}", returnData, remoteIPEndPoint.ToString());
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                receivingUdpClient.Close();
            }
        }
    }
}

```

}

### **Задание:**

1. Разработать консольное приложение UDP-отправитель, задав информацию о получателе дейтаграмм в конструкторе `UdpClient`.
2. Разработать консольное приложение UDP-отправитель, задав информацию о получателе дейтаграмм в методе `Connect`.
3. Разработать консольное приложение UDP-отправитель, задав информацию о получателе дейтаграмм в методе `Send`.
4. Разработать консольное приложение UDP-приемник, который принимает данные только от указанной удаленной точки и отбрасывает все остальные.
5. Разработать консольное приложение UDP-приемник, который принимает все входящие соединения.

### **Литература**

1. Кумар В., Кровчик Э и др. .NET. Сетевое программирование / Пер. с англ. –М.: «Лори», 2007, стр.199-208.
2. <http://metanit.com/sharp/net/5.1.php>
3. [https://professorweb.ru/my/csharp/web/level4/4\\_6.php](https://professorweb.ru/my/csharp/web/level4/4_6.php)

# Широковещательная, групповая рассылка

## Задачи:

1. Сравнить однонаправленные, широковещательные и групповые передачи.
2. Исследовать архитектуру групповой рассылки.
3. Создать приложение группового интерактивного форума.

## Однонаправленные, широковещательные и групповые передачи.

Internet Protocol поддерживает три вида IP-адресов:

1. Однонаправленные – сетевые пакеты посылаются в один пункт назначения.
2. Широковещательные – дейтаграммы отправляются всем узлам подсети.
3. Групповые дейтаграммы – отправляются всем узлам, находящимся, быть может, в нескольких подсетях, которые принадлежат одной группе.

Протокол TCP обеспечивает ориентированное на соединение взаимодействие, при котором две системы обмениваются между собой, но с этим протоколом можно отправлять только однонаправленные сообщения. Если несколько клиентов соединяются с одним сервером, каждый из них поддерживает отдельное соединение с сервером. Серверу требуются ресурсы для всех этих одновременных соединений, и он должен взаимодействовать отдельно с каждым клиентом.

Протокол UDP также можно использовать для посылки однонаправленных сообщений и он в отличие от TCP не устанавливает соединения, делая обмен более быстрым, хотя и не таким надежным, как при использовании TCP.

Широковещательные сообщения *всегда* передаются при взаимодействии без организации соединений с использованием протокола UDP. Широковещательная передача может осуществляться только внутри конкретной подсети.

Широковещательная передача полезна, если несколько узлов одной подсети должны получать информацию одновременно.

Диапазон групповых IPv4-адресов – от 224.0.0.0 до 239.255.255.255. (они могут использоваться только в качестве адреса назначения, адрес источника обязательно должен быть индивидуальным адресом узла).

## Модели приложений с групповой рассылкой.

Взаимодействие многих со многими (каждой станции в группе нужно, чтобы ее данные отправлялись всем другим станциям).

Групповая рассылка означает, что ни одной станции не требуется создавать соединение с каждой другой станцией, а вместо этого можно пользоваться групповым адресом. Одноранговое приложение интерактивного форума, только выиграет от такой схемы. В этом приложении автор сообщения передает его в сеть один раз, и оно поступает в каждый узел группы рис. 1.



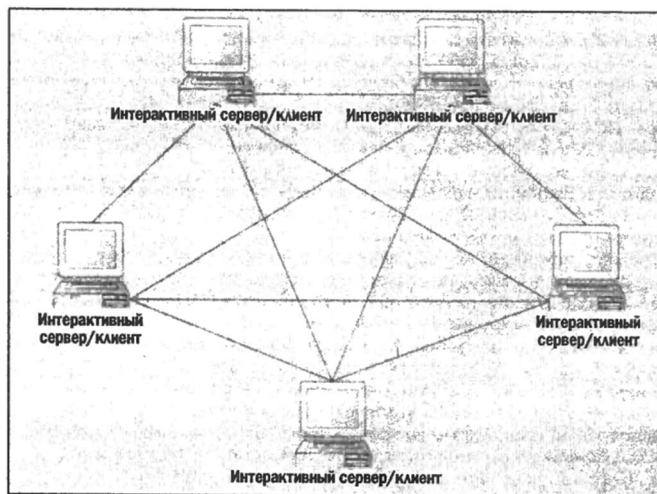


Рис. 1

Взаимодействие одного со многими (сценарий в котором одной станции требуется отправлять данные группе станций). Сервер отправляет данные только один раз по групповому адресу, а большое число станций может слушать его сообщения рис. 2. Эта технология может использоваться в локальной сети для одновременной установки приложений на сотнях PC, и серверу не требуется отправлять большой установочный пакет каждой клиентской станции в отдельности.

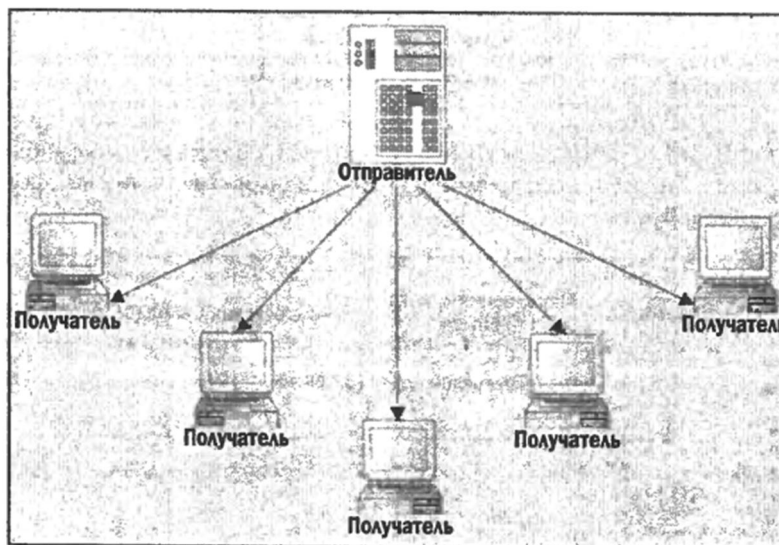


Рис. 2

Создадим приложение группового интерактивного форума, используя модель «Взаимодействие многих со многими».

## Отправитель

У приложения-отправителя нет никаких особых задач, которые не были бы рассмотрены в предыдущих лабораторных работах, поэтому для отправки групповых сообщений просто используем класс `UdpClient`. Единственное различие в том, что теперь нужно использовать групповой адрес. Объект `remoteEP` класса `IPEndPoint` указывает на адрес группы и номер порта, которые будут использоваться группой:

```
IPAddress groupAddress = IPAddress.Parse("234.5.6.11");
```

```
Int remotePort = 7777;  
Int localPort = 7777;  
IPEndPoint remoteEP = new IPEndPoint(groupAddress, remotePort);  
UdpClient server = new UdpClient(localPort);  
Server.Send(data, data.Length, remoteEP);
```

Адрес групповой рассылки нужно сделать известным для клиентов, присоединяющихся к группе. Сделаем это, используя фиксированный адрес в конфигурационном файле, к которому могут получать доступ клиенты.

## Получатель

Клиенты должны присоединиться к группе рассылки. В методе `JoinMulticastGroup()` класса `UdpClient` эта возможность уже реализована. В первом параметре метода `JoinMulticastGroup()` указывается IP-адрес группы рассылки, а второй параметр представляет значение TTL (число маршрутизаторов, которые должны пересылать сообщение).

```
UdpClient udpClient = new UdpClient();  
udpClient.JoinMulticastGroup(groupAddress, 50);
```

Чтобы выйти из группы, вызываем метод `UdpClient.DropMulticastGroup()`, принимающий параметр, в котором задается тот же самый адрес групповой рассылки, указанный в методе `JoinMulticastGroup()`:

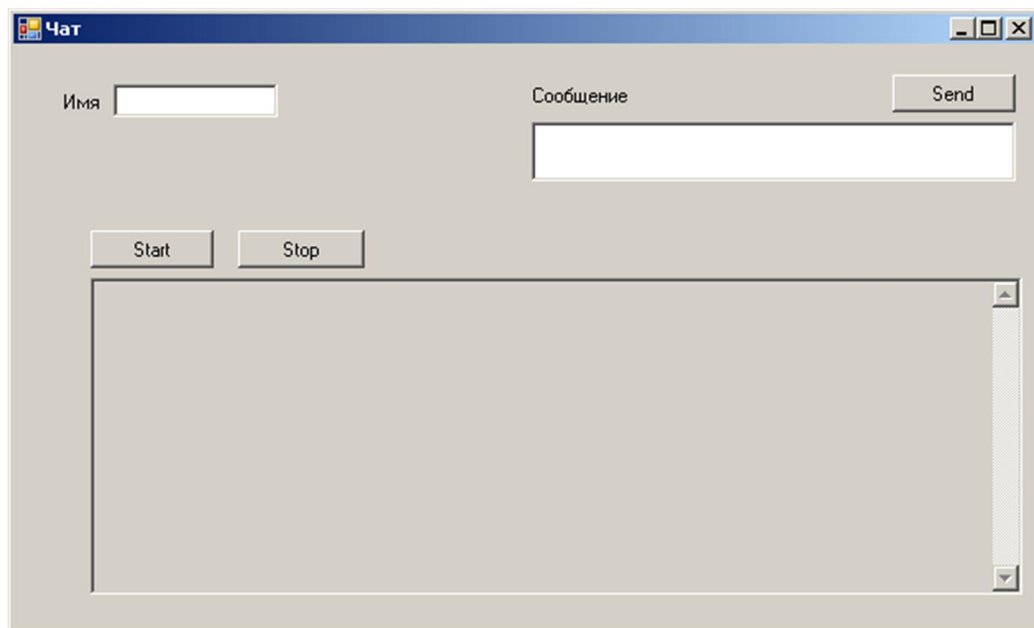
```
UdpClient.DropMulticastGroup(groupAddress);
```

## Создание приложения интерактивного форума

Приступим к разработке приложения групповой рассылки. Создадим простое приложение интерактивного форума, к которому могут обратиться несколько пользователей, чтобы отправить сообщения всем остальным клиентам. В этом приложении каждая станция действует и как клиент, и как сервер. Каждый пользователь может ввести сообщение, отправляемое всем участникам форума.

Для приложения форума строится проект Windows-формы с именем `MulticastChat`. Класс основной формы в этом приложении называется `ChatForm.cs`.

Пользовательский интерфейс приложения интерактивной переписки позволяет ввести имя разговора и присоединится к сетевым взаимодействиям, нажав кнопку `Start`. Нажимая эту кнопку, пользователь присоединяется к группе рассылки, и приложение начинает слушать адрес группы. Сообщения вводятся в текстовом поле под меткой `Message` и отправляются группе, когда нажимается кнопка `Send`:



В следующей таблице показаны основные элементы управления формы с их именами и значениями свойств по умолчанию:

Тип элемента управления	Имя	Свойства
Текстовое поле	textName	Text = ""
Кнопка	buttonStart	Text = "Start"
Кнопка	buttonStop	Enabled = false
		Text = "Stop"
Кнопка	buttonSend	Enabled = false
		Text = "Send"
Текстовое поле	textMessage	Multiline = true
		Text = ""
Текстовое поле	textMessages	Multiline = true
		ReadOnly = true
		Scrollbars = Vertical
		Text = ""
Полоса состояния	statusBar	-

ChatForm – это основной класс приложения.

```
using System;
using System.Windows.Forms;
using System.Configuration;
using System.Collections.Specialized;
using System.Net;
using System.Net.Sockets;
```

```

using System.Text;
using System.Threading;

namespace MulticastChat
{
    public partial class ChatForm : Form
    {
        private bool done = true;
        private UdpClient client;
        private IPAddress groupAddress;
        private int localPort;
        private int remotePort;
        private int ttl;

        private IPEndPoint remoteEP;
        private UnicodeEncoding encoding = new UnicodeEncoding();
        private string name;
        private string message;
        //...
    }
}

```

## Параметры конфигурации

Групповые адреса и номера портов должны быть легко конфигурируемыми, поэтому создадим XML-файл конфигурирования приложения с именем MulticastChat.exe.config и следующим содержанием. Этот конфигурационный файл нужно поместить в тот же каталог, где находится исполнимый файл.

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="GroupAddress" value="234.5.6.11"/>
    <add key="LocalPort" value="7777"/>
    <add key="RemotePort" value="7777"/>
    <add key="TTL" value="32"/>
  </appSettings>
</configuration>

```

Этот конфигурационный файл считывается конструктором класса ChatForm с использованием класса System.Configuration.ConfigurationSettings. Если конфигурационный файл не существует или имеет неверный формат, то порождается исключение, которое мы перехватываем и отображаем сообщение об ошибке.

```

public ChatForm()
{
    InitializeComponent();
    try
    {
        //Считываем конфигурационный файл приложения
        NameValueCollection configuration = ConfigurationSettings.AppSettings;
        groupAddress = IPAddress.Parse(configuration["groupAddress"]);
        localPort = int.Parse(configuration["LocalPort"]);
    }
}

```

```

        remotePort = int.Parse(configuration["RemotePort"]);
        ttl = int.Parse(configuration["TTL"]);
    }
    catch {
        MessageBox.Show(this, "Ошибка в конфигурационном файле!", "Ошибка",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
        buttonStart.Enabled = false;
    }
}

```

## Присоединение к группе, получающей рассылку

В обработчике щелчка по кнопке Start считываем имя, введенное в текстовом поле textName, и записываем его в поле name. Далее создаем объект UdpClient и присоединяемся к группе, получающей рассылку, вызывая метод JoinMulticastGroup(). Затем создаем новый объект IPEndPoint, ссылающийся на адрес групповой рассылки и удаленный порт. Этот объект мы будем использовать в методе Send() для отправки данных группе:

```

private void buttonStart_Click(object sender, EventArgs e)
{
    name = textName.Text;
    textName.ReadOnly = true;

    try {
        // Присоединяемся к группе рассылки
        client = new UdpClient(localPort);
        client.JoinMulticastGroup(groupAddress, ttl);

        remoteEP = new IPEndPoint(groupAddress, remotePort);
    }
}

```

Создаем новый поток, который будет получать сообщения, отправленные по групповому адресу. После запуска потока отправляем группе представляющее нас сообщение. Для преобразования строки в массив байтов, как того требует метод Send(), вызываем метод UnicodeEncoding.GetBytes():

```

// Запускаем поток, получающий сообщения
Thread receiver = new Thread(new ThreadStart(Listener));
receiver.IsBackground = true;
receiver.Start();

//Отправляем первое сообщение группе
byte[] data = encoding.GetBytes("Пользователь " + name + " присоединился к чату");
client.Send(data, data.Length, remoteEP);

```

Последнее действие выполняемое в методе buttonStart\_Click(), должно подключить кнопки Stop и Send и запретить доступ к кнопке Start. Кроме того, напишем обработчик исключения SocketException, которое может возникнуть, если приложение, слушающее один и тот же порт, запущено второй раз:

```

buttonStart.Enabled = false;

```

```
buttonStop.Enabled = true;
buttonSend.Enabled = true;
catch(SocketException ex){
    MessageBox.Show(this, ex.Message, "Ошибка", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
}
```

## Получение сообщений, адресованных группе

В методе слушающего потока, ждем сообщение `client.Receive()`. С помощью класса `UnicodeEncoding` полученный массив байтов преобразуем в строку. Теперь сообщение нужно отобразить в пользовательском интерфейсе.

```
// Основной метод слушающего потока, который получает данные
private void Listener()
{
    done = false;
    try
    {
        while (!done)
        {
            IPEndPoint ep = null;
            byte[] buffer = client.Receive(ref ep);
            message = encoding.GetString(buffer);

            this.Invoke(new MethodInvoker(DisplayReceivedMessage));
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show(this, ex.Message, "Ошибка!", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
    }
}
```

В реализации `DisplayReceivedMessage()` запишем полученное сообщение в текстовом поле `textMessages` и поместим некоторую информацию в полосу состояния:

```
private void DisplayReceivedMessage()
{
    string time = DateTime.Now.ToString("t");
    textMessages.Text = time + " " + message + "\r\n" + textMessages.Text;
    //statusBar.Text = "Последнее сообщение " + time;
}
```

## Отправка групповых сообщений

Следующая задача состоит в реализации функциональной возможности отправки сообщения в обработчике щелчка по кнопке `Send`.

```

private void buttonSend_Click(object sender, EventArgs e)
{
    try
    {
        //Отправляем сообщение группе
        byte[] data = encoding.GetBytes(name + ": " + textMessage.Text);
        client.Send(data, data.Length, remoteEP);
        textMessage.Clear();
        textMessage.Focus();
    }
    catch (Exception ex)
    {
        MessageBox.Show(this, ex.Message, "Ошибка!", MessageBoxButtons.OK,
        MessageBoxIcon.Error);
    }
}

```

## Прекращение членства в группе

Обработчик события щелчка по кнопке Stop, метод OnStop(), останавливает ожидание клиентом сообщений групповой рассылки, вызывая метод DropMulticastGroup(). Прежде чем клиент прекращает получать данные групповой рассылки, группе отправляется заключительное сообщение, информирующее о выходе пользователя из форума:

```

private void buttonStop_Click(object sender, EventArgs e)
{
    StopListener();
}

```

```

private void StopListener()
{
    //Отправляем группе сообщение о выходе
    byte[] data = encoding.GetBytes(name + " покинул чат");
    client.Send(data, data.Length, remoteEP);

    //Покидаем группу
    client.DropMulticastGroup(groupAddress);
    client.Close();
    //Останавливаем поток, получающий сообщения
    done = true;
    buttonStart.Enabled = true;
    buttonStop.Enabled = false;
    buttonSend.Enabled = false;
}

```

Поскольку членство в группе надо прекратить в любом случае – не только когда пользователь щелкает по кнопке Stop, но и когда он завершает приложение, - обрабатываем событие Closing для формы в методе OnClosing(). Если к этому моменту еще не остановлен приемный сервер, нужно опять вызвать метод StopListener() и тем самым после отправки группе заключительного сообщения прекратить слушать группу:

```
private void ChatForm_FormClosing(object sender, FormClosingEventArgs e)
{
    if (!done)
        StopListener();
}
```

### **Задание:**

Скомпилируйте приложение. Запустите экземпляр одного и того же приложения на разных компьютерах. Протестируйте работу.



# Лабораторная работа № 6

## Порядок выполнения работы

1. Протокол HTTP
  - 1.1 HTTP-заголовки
    - 1.1.2 Заголовки ответов
    - 1.1.3 Общие заголовки
2. Классы `HttpRequest` и `HttpResponse`
  - 2.1 Работа с HTTP-заголовками
3. Класс `WebClient`
  - 3.1 Загрузка файлов
4. Аутентификация и разрешения
5. Web-прокси
6. Класс `HttpListener`
7. Задание

## 1. Протокол HTTP

**HTTP** (HyperText Transfer Protocol) – это протокол передачи данных (изначально гипертекстовых данных в формате HTML, потом произвольных данных) прикладного уровня, используемый в World Wide Web (WWW).

Протокол HTTP работает по технологии клиент-сервер. HTTP-сервер – это программа, слушающая на порте машины входящие HTTP-запросы. Клиентом обычно выступает веб-браузер. Клиент формирует и делает запрос к серверу, и тот, обработав запрос, возвращает ответ клиенту.

Объектом, над которым происходит работа протокола HTTP, является ресурс, на который указывает URI в запросе клиента. URI (Uniform Resource Identifier) – унифицированный идентификатор ресурса, простыми словами, это то, что указывается в строке браузера – имя запрашиваемого ресурса (страница, изображение, и т.д.).

**Структура HTTP-сообщения** имеет следующий вид:

[стартовая-строка]

[заголовков-сообщения] (или заголовки)

...

[тело-сообщения]

Стартовая строка запроса и ответа отличаются.

Стартовая строка запроса: [Метод] [URI] HTTP/[Версия].

Здесь метод – название запроса, одно слово заглавными буквами, наиболее часто используются GET, POST, HEAD.

URI – идентификатор запрашиваемого ресурса.

Версия – цифры, разделенные точкой (например 1.1).

Стартовая строка ответа: HTTP/Версия [Код Состояния] [Пояснение].

**Код Состояния** – это трехзначный цифровой код, который определяет результат запроса. Например, если клиент запросил при помощи метода GET некий ресурс, и сервер его смог предоставить, такое состояние имеет код 200. Если же на сервере нет запрашиваемого ресурса, он вернет код состояния 404. Есть и много других состояний.

**Пояснение** – это текстовое отображение кода состояния, для упрощенного понимания человека. Для кода 200 пояснение имеет вид «ОК».

В таблице 1 приведены распространенные коды состояния:

Код	Описание
200	Хорошо. Успешный запрос
301	Запрошенный ресурс был окончательно перенесен на новый URI
302	Запрошенный ресурс был временно перенесен на другой URI
400	Неверный запрос - запрос не понят сервером из-за наличия синтаксической ошибки
401	Несанкционированный доступ — у пользователя нет прав для доступа к запрошенному документу.
404	Не найдено - сервер понял запрос, но не нашёл соответствующего ресурса по указанному URI
408	Время ожидания сервером передачи от клиента истекло
500	Внутренняя ошибка сервера—ошибка помешала HTTP-серверу обработать запрос

## 1.1 HTTP-заголовки

**Заголовки HTTP** – это строки в HTTP-сообщении в формате «имя: значение». Другими словами их можно назвать метаданными (информация об используемых данных) HTTP- сообщения.

Заголовки делятся на те, которые используются в запросах, на те которые включаются в ответы, и на те, которые могут быть как запросе, так и в ответе.

### 1.1.1 Заголовки запросов

**Заголовок Referer** – URI ресурса, с которого клиент сделал запрос на сервер. Перейдя со страницы 1 на страницу 2, Referer будет содержать адрес страницы 1. Этот заголовок может быть полезным, например, для того, чтобы отследить по каким поисковым запросам посетители попали на ваш сайт.

Или сервер может как либо иначе обрабатывать запрос, если Referer не тот, который ожидается.

**Заголовок Accept** используется для того, чтобы клиент сообщил серверу, какие типы контента он поддерживает. Типы указываются в формате тип/подтип через запятую (Accept: text/html, text/plain, image/jpeg). Если тип не может быть обработан, возвращается HTTP-код 406 «Not acceptable».

**Заголовок User-Agent** содержит информацию о клиентском приложении. Обычно это имя браузера, его версия и платформу. В первую очередь этот заголовок нужен для корректного отображения страницы. Браузеров есть много, много версий и не все могут одинаково отображать контент, web-программисты учитывают эту информацию и выдают различным браузерам различные скрипты/стили.

**Заголовок Content-length** содержит длину передаваемых данных в байтах при использовании метода передачи POST. При методе GET он не устанавливается.

### 1.1.2 Заголовки ответов

Заголовок Server содержит информацию о программном обеспечении, которое использует сервер (Server: Microsoft-IIS/7.0).

Заголовок Content-Type указывает тип данных, которые отправляются клиенту, или которые могли бы отправиться, используя метод HEAD 9 (Content-Type: text/html; charset=utf-8).

### 1.1.3 Общие заголовки

Заголовок Date указывает дату и время создания сообщения (Date: Mon, 16 Nov 2015 09:09:39 GMT).

Заголовок Content-Language содержит список языков, для которых предназначается контент (Content-Language: en, ru).

Тело HTTP сообщения Тело (message-body) используется для передачи тела объекта, связанного с запросом или ответом. Обычно это сгенерированный html-код, который браузер потом будет отображать.

Тело обязательно отделяется от заголовков пустой строкой.

## 2. Классы HttpRequest и HttpResponse

Платформа .NET Framework для работы с протоколом HTTP предоставляет два основных класса. Один из них отвечает за запрос – класс HttpRequest, а другой за ответ – HttpResponse.

Рассмотрим пример простой программы, с использованием данных классов, которая получает из интернета указанную страничку:

### Листинг 1

```
static void Main()
{
    string uri = "адрес web-страницы";
    HttpWebRequest request = (HttpWebRequest)WebRequest.Create(uri);
    HttpWebResponse response = (HttpWebResponse)request.GetResponse();
    StreamReader reader = new StreamReader(response.GetResponseStream(),
    Encoding.UTF8);
    Console.WriteLine(reader.ReadToEnd());
    Console.ReadLine();
}
```

Сначала создается объект запроса при помощи метода `Create()` класса `WebRequest`, в который передается адрес запрашиваемого ресурса. При этом возвращаемый объект необходимо привести к типу `HttpWebRequest`, потому как метод `Create()` возвращает различный объект запроса, основываясь на том, какой URI ему передали, это может быть, как в нашем случае, HTTP запрос, так и запрос к файловой системе (при этом возвращается объект `FileWebRequest`).

Дальше мы создаем объект ответа `HttpWebResponse` путем вызова метода `GetResponse` у объекта запроса (при этом приводим его к типу `HttpWebResponse`). При вызове этого метода, на сервер отправляется запрос, и в результате мы получаем объект `HttpWebResponse` который содержит HTTP-заголовки, а также поток, с которого мы можем считать тело ответа.

Дальше создается объект чтения потока `StreamReader`, в его конструктор передается поток ответа, который возвращает метод `GetResponseStream`, указываем необходимую кодировку, и выводим данные на экран. В результате мы увидим HTML код запрашиваемой страницы.

### 2.1 Работа с HTTP-заголовками

Для работы с заголовками у обоих классов `HttpWebRequest` и `HttpWebResponse` есть свойство `Headers` (объект типа `WebHeaderCollection`) которое предоставляет информацию о заголовках запроса и ответа соответственно. Для добавления или установки заголовков используются методы `Add` и `Set`. Метод `Add` принимает один строковый аргумент – заголовок полностью в формате "имя: значение" - `request.Headers.Add("Content-Language: en, ru")`.

Метод Set принимает два строковых аргумента – имя и значение:

```
request.Headers.Set("Content-Language", "en, ru");
```

Использовать можно любой из этих методов, но кроме такой "ручной" установки заголовков есть еще возможность задать значения некоторых распространенных заголовков, используя соответствующие свойства в классах `HttpRequest` и `HttpResponse`.

Например, установка заголовка Referer:

```
request.Referer = "http://google.com";
```

Доступ к конкретному заголовку осуществляется при помощи той же коллекции Headers (коллекция пар имя-значение `WebHeaderCollection`). В квадратных скобках указываем имя заголовка, и получаем его значение:

```
Console.WriteLine(response.Headers["date"]); // Tue, 16 Nov 2015  
09:09:39 GMT
```

Считать все заголовки можно так:

```
foreach (string header in response.Headers)  
Console.WriteLine("{0}: {1}", header, response.Headers[header]);
```

В примерах выше все запросы выполнялись методом "GET". У классов `HttpRequest` и `HttpResponse` есть свойство `Method`. Для запроса оно по умолчанию установлено как "GET", но его можно изменить, например на "POST".

### 3. Класс WebClient

Если необходимо только запросить файл с определенного URI (Uniform Resource Identifier — унифицированный идентификатор ресурса), то простейшим в использовании классом .NET, который подходит для этого, будет `System.Net.WebClient`. Этот исключительно высокоуровневый класс предназначен для выполнения базовых операций с помощью всего одной или двух команд. В настоящее время в .NET Framework поддерживаются URI, начинающиеся с идентификаторов `http:`, `https:` и `file:`.

Важно отметить, что термин URL (Uniform Resource Locator — универсальный локатор ресурсов) больше не используется в новых технических спецификациях, а вместо него отдается предпочтение URI. URI

имеет приблизительно тот же смысл, что и URL, но немного более общий, потому что в URL не подразумевается обязательное применение одного из знакомых протоколов, таких как HTTP или FTP.

### 3.1 Загрузка файлов

Для загрузки файлов с использованием WebClient доступны два метода. Выбор метода зависит от того, как должно обрабатываться содержимое файла. Если необходимо просто сохранить файл на диске, следует применять метод DownloadFile(). Этот метод принимает два параметра: URI файла и местоположение (путь и имя файла) для сохранения запрошенных данных:

#### Листинг 2

```
using System;
using System.Net;
using System.IO;
namespace NetConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            WebClient client = new WebClient();
            client.DownloadFile("http://www.polessu.by", "01.txt");
            Console.WriteLine("Файл загружен");
        }
    }
}
```

Часто приложение должно обрабатывать данные, извлеченные с веб-сайта. Это обеспечивает метод OpenRead(), возвращающий ссылку на Stream, которую можно использовать для извлечения данных в память.

В следующем примере демонстрируется применение метода WebClient.OpenRead(). Содержимое загруженной страницы будет отображено в элементе управления TextBox. Для начала создайте новый проект как стандартное приложение WPF и добавьте элемент управления TextBox по имени txb. В начало файла к списку директив using потребуется добавить ссылки на пространства имен System.Net и System.IO. Затем добавьте обработчик клика по кнопке:

### Листинг 3

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    WebClient client = new WebClient();
    Stream stream = client.OpenRead("http://www.polessu.by");
    StreamReader sr = new StreamReader(stream);
    string newLine;
    while ((newLine = sr.ReadLine()) != null)
        txb.Text += newLine;
    stream.Close();
}
```

### Листинг 4

```
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://www.polessu.by/01.txt"))
            {
                using (StreamReader reader = new StreamReader(stream))
                {
                    string line = "";
                    while ((line = reader.ReadLine()) != null)
                    {
                        Console.WriteLine(line);
                    }
                }
            }
            Console.WriteLine("Файл загружен");
            Console.Read();
        }
    }
}
```

## 4. Аутентификация и разрешения

Если Web-сервер требует аутентификации пользователя, можно создать удостоверение личности пользователя и передать его Web-запросу. При этом полезны следующие интерфейсы и классы: ICredentials, NetworkCredential и CredentialCache.

Для аутентификации пользователя создадим объект типа NetworkCredential. Этот класс обеспечивает информацию с целью удостоверения личности пользователя для базовой аутентификации, аутентификации на основе дайджестов, NTLM и Kerberos.

Конструктору класса NetworkCredential можно передать имя пользователя, пароль и дополнительно домен, разрешающий доступ пользователя:

```
NetworkCredential credential = new NetworkCredential("логин", "пароль",  
"www.polessu.by");
```

Для авторизации пользователя эту информацию удостоверения личности можно установить в свойстве Credentials класса WebRequest:

```
WebRequest request = WebRequest.Create("http://www.polessu.by/");  
request.Credentials = credential;
```

### Листинг 5

```
using System;  
using System.Net;  
using System.IO;  
namespace NetConsoleApp  
{  
    class Program  
    {  
  
        static void Main(string[] args)  
        {  
  
            string query = "http://www.polessu.by";  
            WebRequest request = (HttpWebRequest)WebRequest.Create(query);  
  
            request.Credentials = new NetworkCredential("логин", "пароль");  

```



```

        HttpResponseMessage response =
(HttpWebResponse)request.GetResponse();

        StreamReader reader = new
streamReader(response.GetResponseStream());
        Console.WriteLine(reader.ReadToEnd());
        Console.Read();
        response.Close();
        reader.Close();

    }
}

```

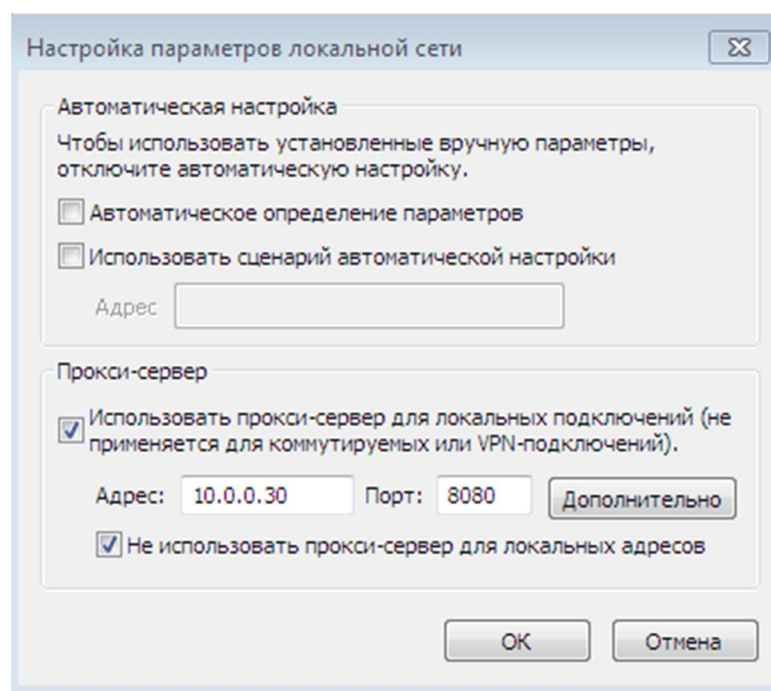
## 5. Web-прокси

В локальной сети можно использовать прокси-сервер, чтобы направить интернет-доступ к конкретным серверам. Прокси-сервер может сократить число передач и сетевых соединений из Интернета и повысить благодаря кэшированию ресурсов производительность локальных клиентов.

Прокси-сервер выполняет активное и пассивное кэширование:

- При пассивном кэшировании Web-ресурсы сохраняются в кэше прокси-сервера, как только клиент запрашивает ресурс. Если второй клиент запрашивает тот же самый ресурс, получать его снова от Web-сервера в Интернете не нужно, поскольку Web-прокси может ответить непосредственно из кэша, созданного при первом запросе.
- С помощью активного кэширования системный администратор может сконфигурировать конкретные Web-серверы и каталоги, которые должны кэшироваться автоматически в соответствии со специальным расписанием, например в ночные часы. Этим способом пропускная способность, необходимая для Интернета, днем может быть сокращена, чтобы увеличить производительность для часто используемых страниц.

Прокси-сервер, настроенный по умолчанию, устанавливается из Internet Options в Control Panel. Кроме того, к средствам конфигурирования можно также получить доступ из Internet Explorer (Tools --> Internet Options --> Connections --> LAN Settings):



В данном случае Web-прокси-сервер имеет IP-адрес 10.10.0.30 и слушает порт 8080. Этот прокси-сервер не должен использоваться для Web-серверов в интрасети. Через кнопку "Дополнительно" можно сконфигурировать разные прокси-серверы для разных протоколов (HTTP, HTTPS или FTP) и выбрать конкретные Web-сайты, к которым прокси-сервер не должен обращаться.

## 5.1 Класс WebProxy

Класс WebProxy используется для определения прокси-сервера. Свойства этого класса аналогичны настройкам, которые были рассмотрены вместе с конфигурированием прокси-сервера:

*Класс WebProxy*

Свойства WebProxy	Описание
Address	Свойство Address имеет тип Uri и определяет URI прокси-сервера, IP-адрес или имя и номер порта.
BypassList	В свойстве BypassList можно получать и устанавливать в массиве строк URI, которые не должны использовать прокси-сервер.

BypassArrayList	BypassArrayList — это свойство только для чтения, возвращающее объект типа ArrayList, представляющий URI, которые устанавливаются в свойстве BypassList.
BypassProxyOnLocal	BypassProxyOnLocal — это логическое свойство, указывающее, должны ли с прокси-сервером использоваться локальные адреса.
Credentials	Если прокси-сервер требует аутентификации пользователя, в свойстве Credentials можно передать удостоверение личности пользователя.

Вместо того, чтобы использовать установленный по умолчанию Web-прокси для всех запросов, можно выделить другой прокси для конкретных запросов. Для выбора другого прокси нужно лишь установить свойство Proxy класса WebRequest:

### Листинг 6

```
using System;
using System.Net;
using System.IO;
namespace NetConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            WebProxy myProxy = new WebProxy("10.0.0.30", 8080);
            myProxy.BypassProxyOnLocal = true;
            string query = "http://www.polessu.by/";
            HttpWebRequest request =
            HttpWebRequest.Create(query);
            request.Proxy = myProxy;
        }
    }
}
```

## 6. Класс 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.

### Листинг 7

```
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.

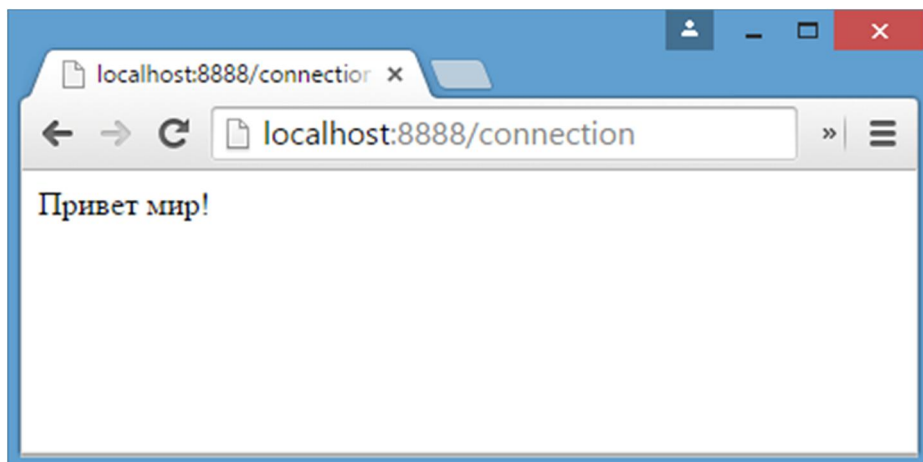
Для получения запроса используется свойство `context.Request`, которое возвращает объект **HttpRequest**, а для получения объекта ответа - свойство `context.Response`. После получения ответа получаем выходной поток и пишем в него массив байтов:

```

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:



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

### Листинг 8

```
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();
    }
}
```

### 7. Задание

1. Пользуясь объектами `HttpRequest` и `HttpResponse` напишите приложение, в котором эти объекты используются для преобразования валюты одного вида в другую, с помощью текущих валютных курсов с Web-сайта.
2. Написать приложение для прослушивания подключений по протоколу HTTP и ответа на HTTP-запросы с кодировкой html страницы в кириллицу
3. Написать данное приложение с использование windows form используя toolbox инструмент webBrowser
4. Написать приложение для прослушивания подключений по протоколу HTTP и ответа на HTTP-запросы с применением метода `WebClient.OpenRead()`

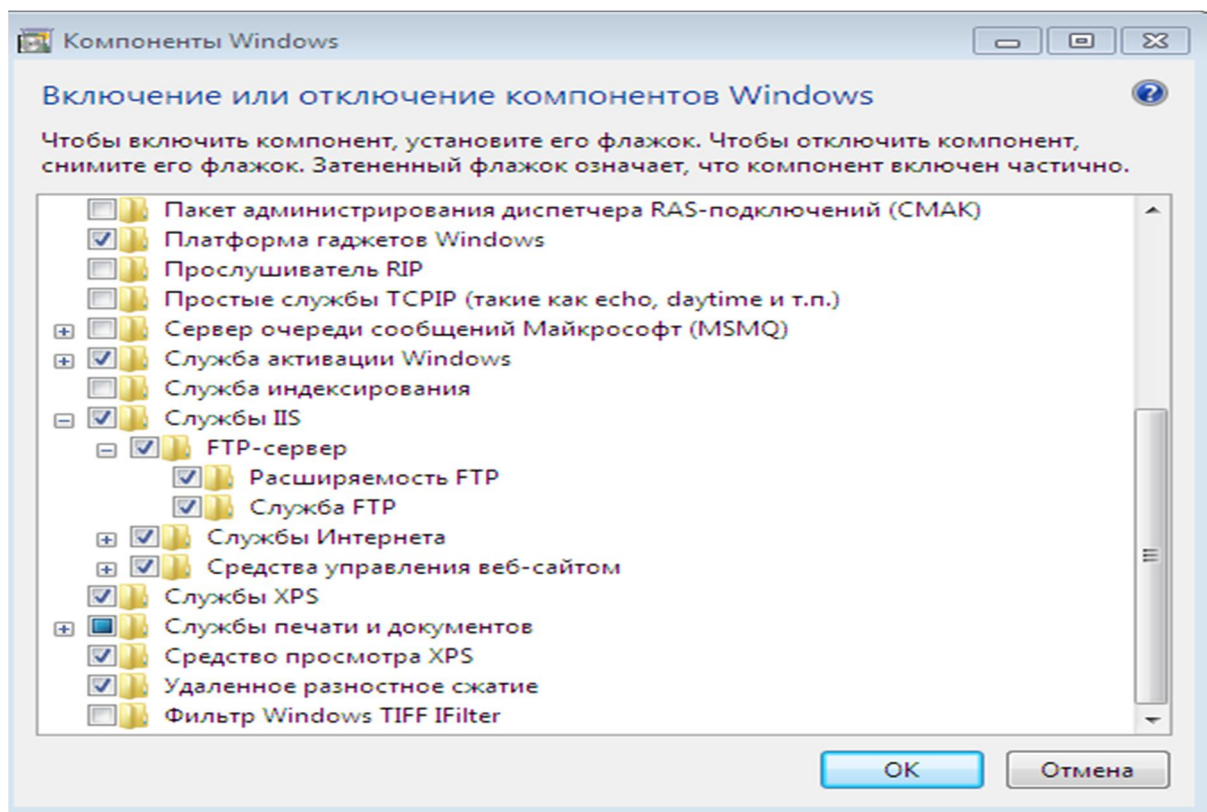
# Лабораторная работа № 7

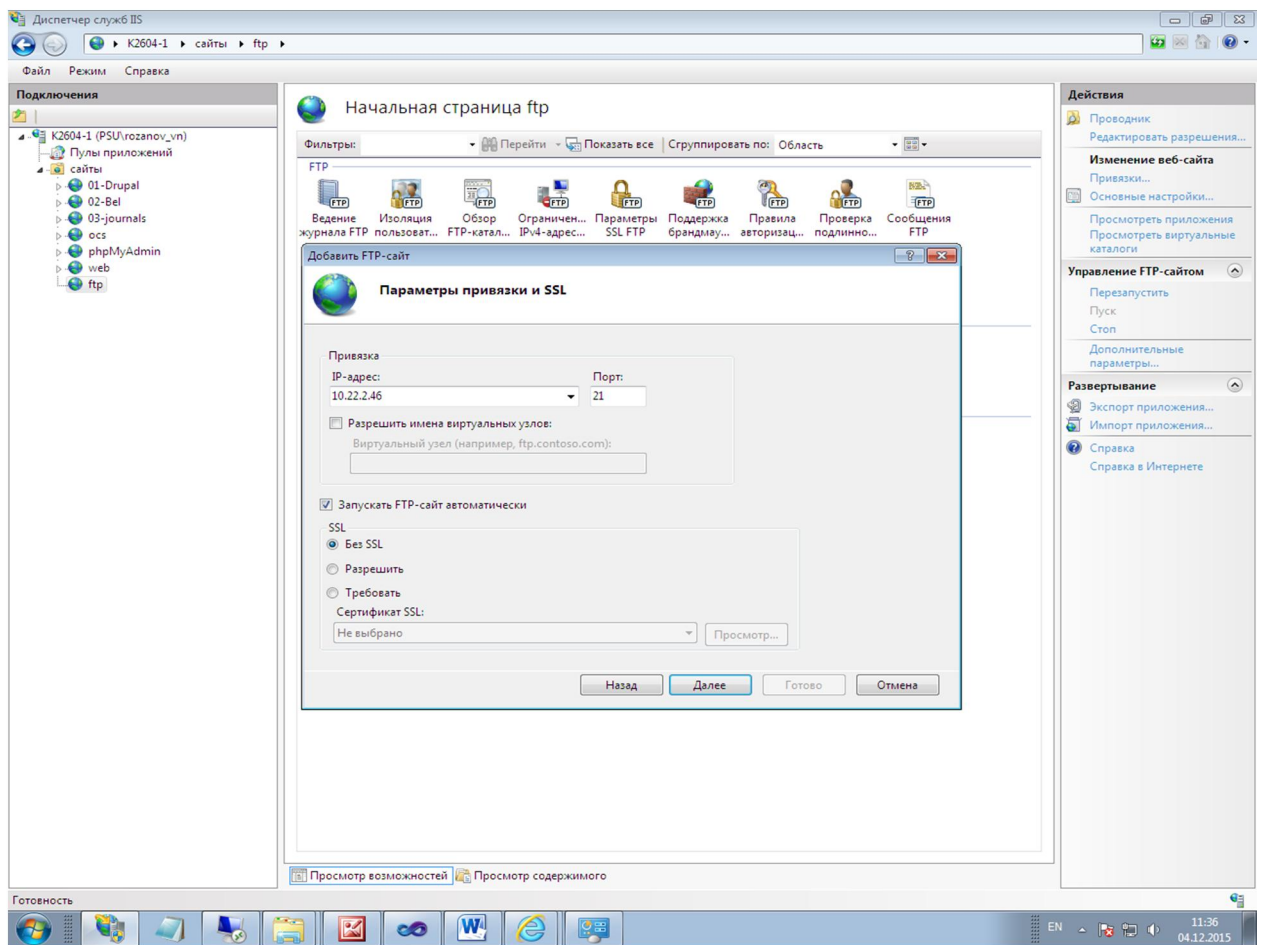
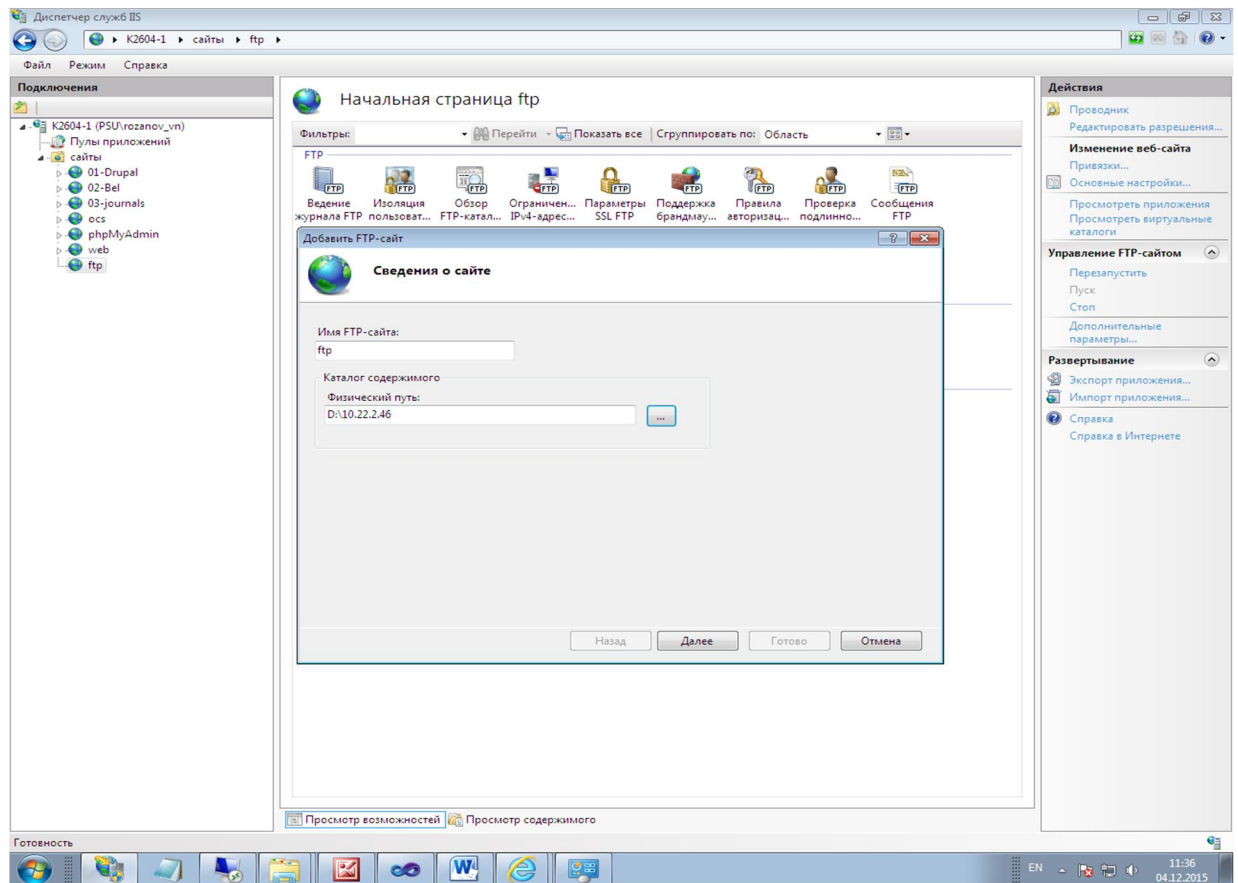
## Порядок выполнения работы

1. Протокол FTP
- 1.2 Создание FTP-сайта
- 1.2 Классы FtpWebRequest и FtpWebResponse
2. Команды протокола FTP
3. Задание

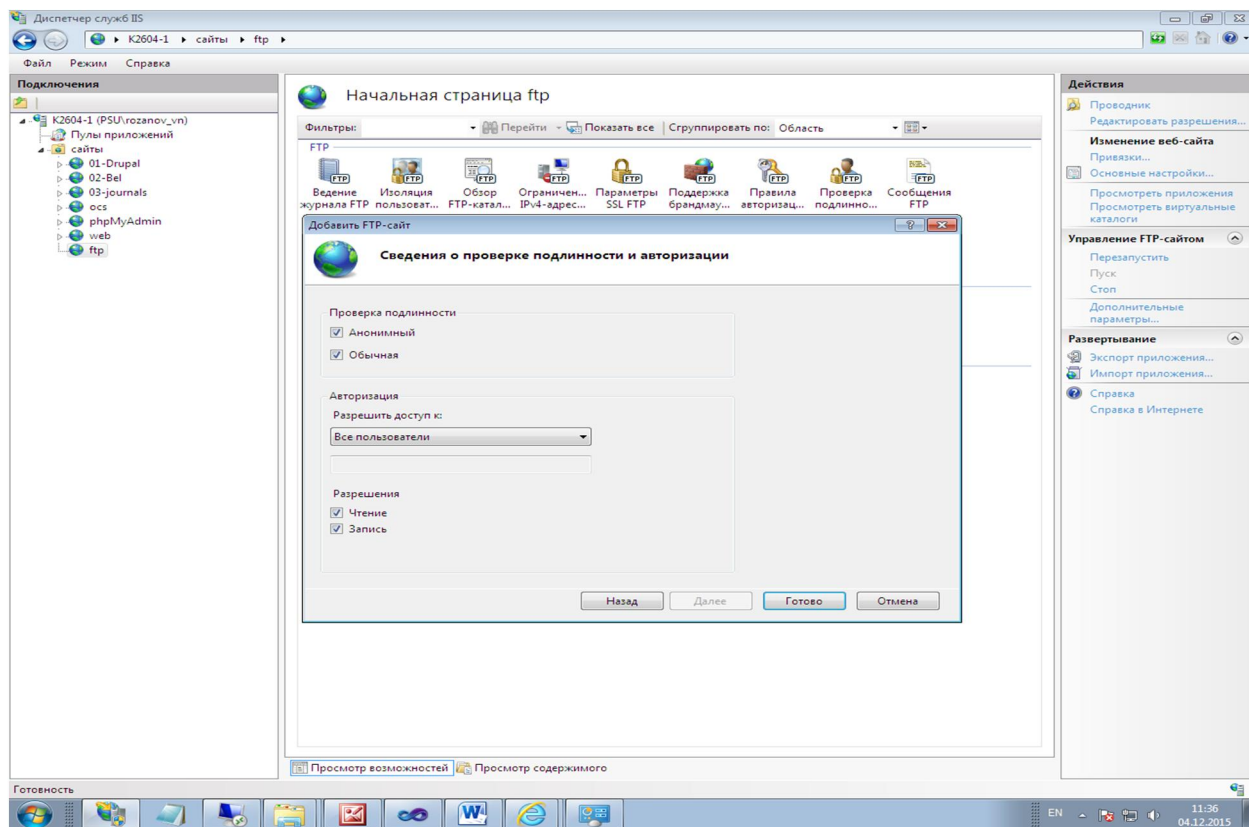
### 1. Протокол FTP

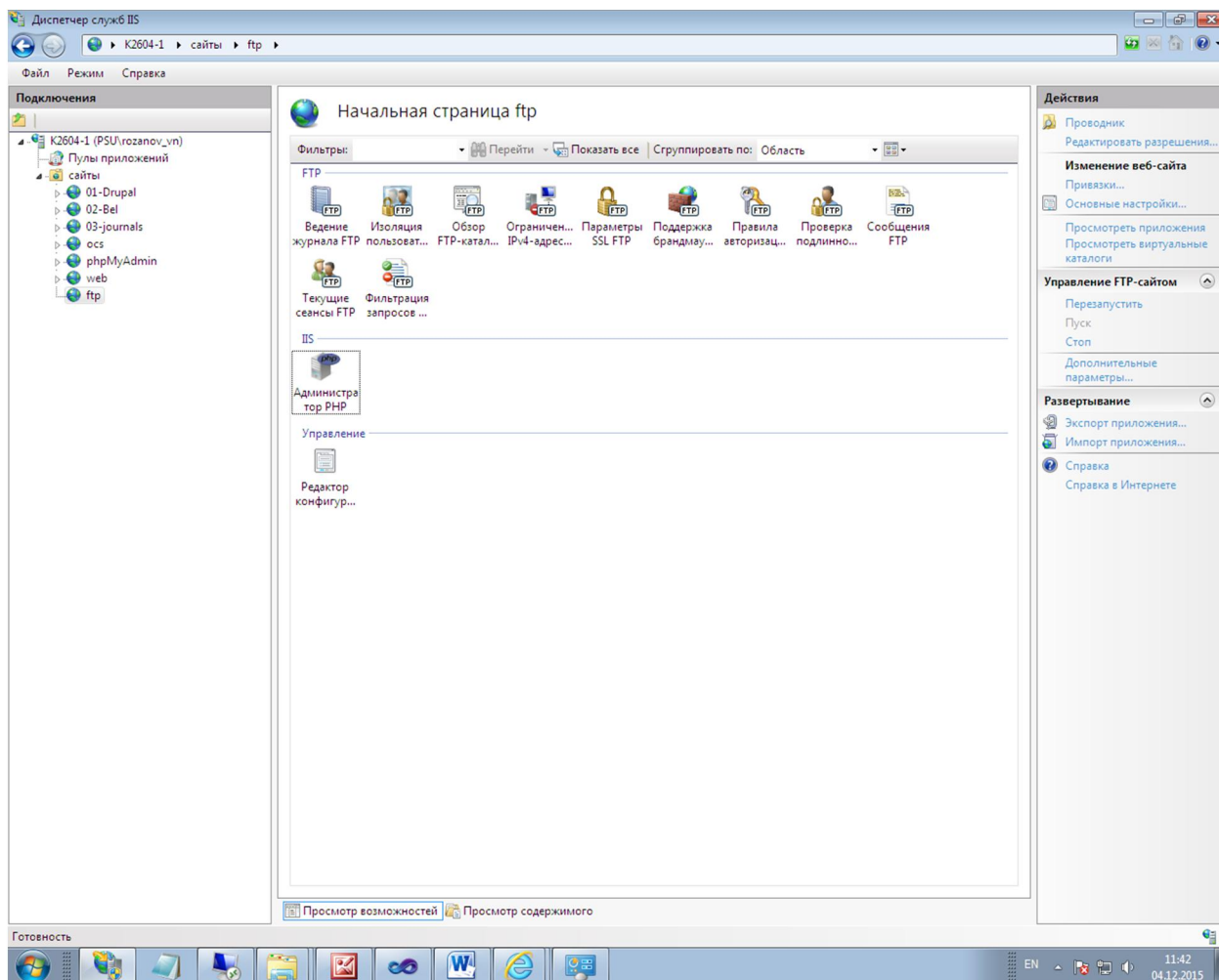
#### 1.2 Создание FTP-сайта











## 1.2 Классы 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 ftp-сервера:**

### Листинг 1

```
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://10.22.2.46/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 ftp-серверу "ftp:// 10.22.2.46", но это может быть любой адрес рабочего ftp-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 ftp-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.

## 2. 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 NLST, 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 ftp-сервер:**

### Листинг 3

```
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://10.22.2.46/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();
        }
    }
}
```

```
}
```

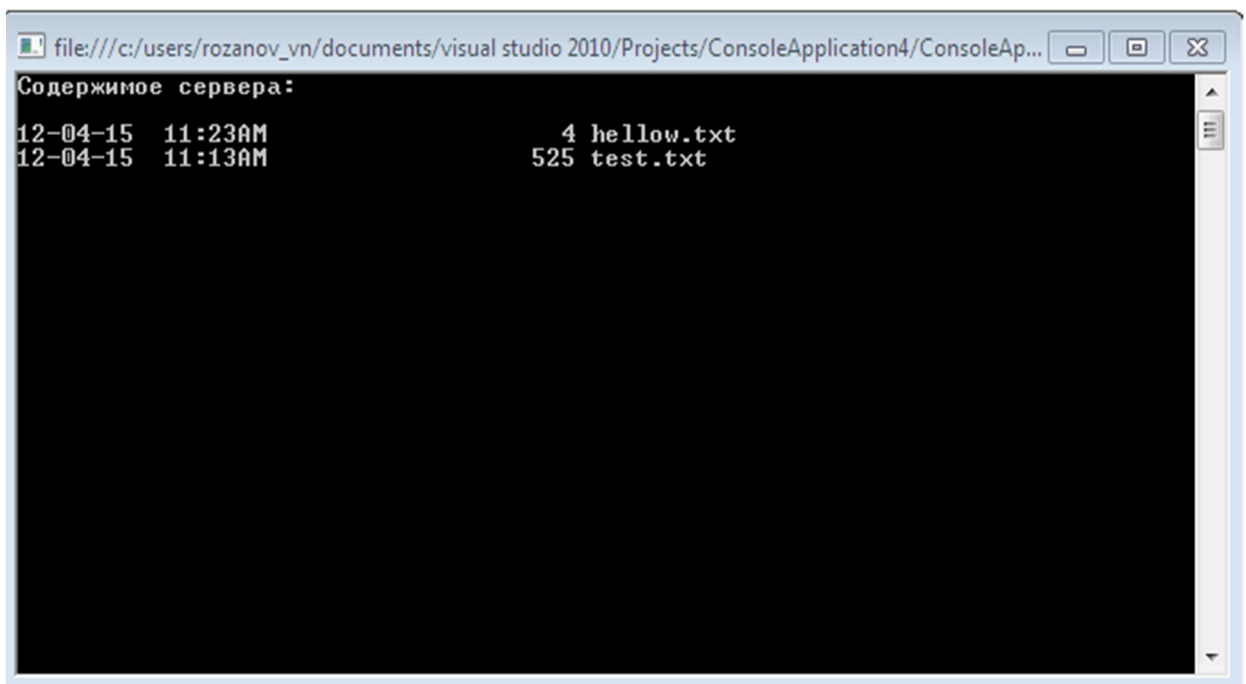
Для загрузки на сервер у нас, естественно, должны быть права на запись. Возможно, потребуется установить логин и пароль для доступа к серверу.

**С помощью другой команды получим содержимое сервера:**

#### **Листинг 4**

```
FtpWebRequest request =  
(FtpWebRequest)WebRequest.Create("ftp://10.22.2.46/");  
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:



```
file:///c:/users/rozanov_vn/documents/visual studio 2010/Projects/ConsoleApplication4/ConsoleAp...  
Содержимое сервера:  
12-04-15 11:23AM          4 hellow.txt  
12-04-15 11:13AM       525 test.txt
```

### **3. Задание**

1. Выполнить листинг 1, 2, 3, 4 в Visual Studio 2010



## Лабораторная работа № 8

### Порядок выполнения работы

1. Протокол SMTP (SmtpClient, MailMessage)
2. Задание

#### 1. Протокол SMTP (SmtpClient, MailMessage)

**SMTP** (англ. Simple Mail Transfer Protocol – простой протокол передачи почты) – это широко используемый сетевой протокол, предназначенный для передачи электронной почты в сетях TCP/IP.

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:

<i>Host</i>	SMTP-сервер, с которого производится отправление почты. Например, smtp.gmail.com
<i>Port</i>	Порт, используемый SMTP-сервером. Если не указан, то по умолчанию используется 25 порт.
<i>EnableSsl</i>	Указывает, будет ли использоваться протокол SSL при отправке.
<i>Credentials</i>	Аутентификационные данные отправителя.

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

<i>From</i>	Адрес отправителя. Представляет объект MailAddress.
<i>To</i>	Адрес получателя. Также представляет объект MailAddress.
<i>Subject</i>	Тема или заголовок письма типа String.
<i>Body</i>	Тело письма, т.е. само сообщение.
<i>Attachments</i>	Содержит все прикрепления к письму.
<i>IsBodyHtml</i>	указывает, представляет ли письмо содержимое с кодом html

Используя эти классы, выполним отправку письма:

#### Листинг 1

```

void SendMail(string smtpServer,
    string from,
    string password,
    string to,
    string caption,
    string message,
    string attachFile = null
)
{
    MailMessage mail = new MailMessage();
    mail.From = new MailAddress(from);
    mail.To.Add(new MailAddress(to));
    mail.Subject = caption;
    mail.Body = message;
    if (!string.IsNullOrEmpty(attachFile))
        mail.Attachments.Add(new Attachment(attachFile));
    SmtpClient client = new SmtpClient();
    client.Host = smtpServer;
    client.Port = 25;
    client.EnableSsl = true;
    client.Credentials = new NetworkCredential(from.Split('@')[0], password);
    client.DeliveryMethod = SmtpDeliveryMethod.Network;
    client.Send(mail);
    mail.Dispose();
}

```

**SendMail**("smtp.gmail.com", "mymail@gmail.com", "myPassword",  
"yourmail@gmail.com", "Тема письма", "Тело письма", "C:\\1.txt");

## 2. Задание

С помощью классов пространства имен **System.Net.Mail** реализовать функции:

- отправка почты, в том числе нескольким адресатам;
- отправка копии и скрытой копии письма;
- вложение файла к письму;
- использование HTML-кода в теле письма.