

# СЕТЕВОЕ ПРОГРАММИРОВАНИЕ

СЕТЕВОЕ ПРОГРАММИРОВАНИЕ  
В NET FRAMEWORK

TCP И UDP СОКЕТЫ, UNICAST,  
BROADCAST, MULTICAST

ИСПОЛЬЗОВАНИЕ СЕТЕВЫХ  
ПРОТОКОЛОВ HTTP, SMTP, FTP

# Урок №1, 2

Введение  
в сети, сокеты.

Асинхронные  
сокеты

## Содержание

<b>1. Что такое сетевое программирование.....</b>	<b>3</b>
<b>2. Цели и задачи сетевого программирования .....</b>	<b>5</b>
<b>3. Что такое сеть .....</b>	<b>7</b>
<b>4. Модель OSI .....</b>	<b>8</b>
<b>5. Базовые термины .....</b>	<b>10</b>
<b>6. Класс Socket .....</b>	<b>17</b>
<b>7. Асинхронные сокеты .....</b>	<b>25</b>
<b>8. Домашнее задание .....</b>	<b>41</b>

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе [Adobe Acrobat Reader](#).

# 1. Что такое сетевое программирование

---

В этом уроке мы с Вами начинаем рассмотрение нового учебного курса под названием «сетевое программирование». Как видно из названия курса, технологии, которые мы здесь будем изучать, имеют прямое отношение к программированию приложений, предназначенных для взаимодействия между несколькими компьютерами по сети.

Начиная с шестидесятых годов XX века, в мире начались работы по созданию сетей передачи данных. Первая публичная демонстрация возможности взаимодействия нескольких компьютеров с использованием телефонных сетей состоялась 29 октября 1969 года в 21:00. Сеть состояла из двух терминалов, один из которых находился в Калифорнийском университете, а второй на расстоянии 600 км от него — в Стэнфордском университете.

Разработка координировалась агентством передовых оборонных исследовательских проектов США (DARPA). При этом одной из наиболее существенных особенностей созданной сети была отказоустойчивость, то есть даже при разрушении части сети (например, в результате ядерного удара по территории страны), оставшаяся часть сети могла продолжать работу.

Так, собственно, начиная с 70-х годов XX века и началось использование сетей передачи данных. Естественно, возникла необходимость писать программное обеспече-

ние, предназначенное для обеспечения взаимодействия программных продуктов между собой.

Однако в то время сеть ещё не могла легко взаимодействовать с другими сетями, построенными на других технических стандартах. К концу 1970-х годов начали бурно развиваться протоколы передачи данных, которые были стандартизированы к середине 80-х годов.

До 90-х годов XX века сеть в основном использовалась для пересылки электронной почты, тогда же появились первые списки почтовой рассылки, новостные группы и доски объявлений.

С 1 января 1983 года сеть ARPA стала использовать для работы протокол TCP/IP вместо протокола NCP. TCP/IP успешно применяется до сих пор для объединения (или, как ещё говорят, «наслоения») сетей. Именно тогда же (в 1983 году) за сетью ARPA закрепился термин «Internet».

## 2. Цели и задачи сетевого программирования

---

При таком бурном развитии сетей передачи данных появился новый класс приложений, существование которых невозможно в отрыве от сети. Такие приложения получили название **клиент-серверных** приложений.

При работе в сети один компьютер выступает в роли **сервера**, а второй подключается к нему в качестве **клиента**, при этом подразумевается, что оба компьютера должны использовать один и тот же способ подключения (например, тип кабеля, размах сигнала, способ модуляции) и использовать один и тот же «язык» для общения.

Чтобы стандартизовать способы обмена между компьютерами, используются специальные наборы правил обмена данными, называемыми **протоколами** передачи данных.

При этом задача разбивается как минимум на две части: часть логики приложения реализуется на сервере, а часть логики — на клиенте. При этом чаще всего в серверной части не предусматривается наличие пользовательского интерфейса. Более того, клиентская часть зачастую представляет собой пользовательский интерфейс со средствами взаимодействия по сети с сервером.

Клиент-серверные приложения позволяют распределять вычислительные задачи между несколькими компьютерами в сети, что позволяет оптимизировать задачи, выполняемые на нескольких машинах.

Клиент-серверные приложения также позволяют разбивать задачи на части, задавая решение одной задачи нескольким компьютерам в сети, такая технология получила название технологии **распределённых вычислений**.

Таким образом, мы можем вывести основную цель разработки сетевых приложений — это создание эффективного и безопасного взаимодействия различных компьютеров находящихся как в локальных сетях (*intranet*), так и разбросанных по всему миру (*internet*).

## 3. Что такое сеть

---

Давайте определимся с терминами, которые мы будем вводить в настоящем уроке. Итак, первое, что мы должны определить, это что понимать под словом «сеть».

**Сеть** — это группа компьютеров или устройств, соединённых между собой каналами связи.

Все эти устройства (компьютеры, маршрутизаторы, шлюзы, принтеры) называют узлами сети. Узлы между собой соединяются каналами, в качестве которых могут использоваться любые линии связи (кабельные, кабельные оптические, оптические атмосферные, радио и т.д.). По этим **каналам связи** узлы сети могут взаимодействовать между собой, передавая друг другу **сообщения**.

По своим размерам сети подразделяются на **локальные**, объединяющие узлы в пределах одного здания или в группе близкорасположенных зданий в пределах района и **глобальные**, объединяющие между собой несколько локальных сетей.

## 4. Модель OSI

В целях формализации протоколов обмена данными по каналам связи организацией ISO была принята как стандарт модель протоколов, получившая название семиуровневой модели OSI (*Open System Interconnection*). В рамках этой модели были выделены основные задачи взаимодействия узлов в сети.

У каждого уровня модели OSI есть особое назначение и каждый из них связан с уровнем находящимся выше и ниже текущего.

Вот эти семь уровней:

7	прикладной (Application),
6	представительный (Presentation),
5	сеансовый (Session),
4	транспортный (Transport),
3	сетевой (Network),
2	канальный (Data Link)
1	физический (Physical)

Проблема взаимодействия двух компьютеров по сети может решиться только в том случае, если программист, который программирует это взаимодействие, чётко понимает работу сети.

Вы только что прослушали курс «Введение в сетевые технологии», в котором вы подробно изучали модель OSI, построение и принципы функционирования локальных и глобальных сетей.



Давайте кратко напомним вам о назначении того или иного уровня модели OSI и покажу вам те моменты которые в первую очередь интересуют программиста.

- **На физическом уровне** определяется среда передачи данных способы кодирования передаваемой информации, разъёмы и кабели, применяемые для построения сети. Программисту, занимающемуся решением прикладных задач на этом уровне делать нечего.
- **Канальный уровень**, во-первых, проверяет доступность среды передачи. Во-вторых, реализует механизмы обнаружения и коррекции ошибок.
- **Сетевой уровень** (*Network layer*) служит для образования единой транспортной системы, объединяющей несколько сетей. При этом эти сети могут использовать абсолютно разные принципы передачи информации и быть организованными совершенно произвольно по структуре! На этом уровне используется логическая адресация узлов (в отличие от физических адресов канального уровня). Дело в том, что физические адреса канального уровня можно использовать только внутри локальной сети.

## 5. Базовые термины

Обычно (или привычно) на сетевом уровне используется протокол IP, то есть каждому узлу в сети присваивается некий уникальный для данной сети IP адрес.

Программист при реализации сетевого приложения должен знать IP адрес узла, к которому он хочет подключиться и IP адрес узла, к которому будет произведено подключение.

На сетевом уровне программист создаёт специальный объект сетевого API — *socket* (*гнездо*), который связывается с конкретным сетевым адресом и типом транспортного протокола.

Используя сокет, программист может осуществить взаимодействие с другим сокетом в сети.

Следующий уровень — **транспортный**. И тут придется поговорить подробнее. Поскольку на транспортном уровне мы впервые сталкиваемся с приложениями. На транспортном уровне мы сталкиваемся с понятием **порта** или конечной точкой (*end point*), которая описывает приложение готовое принять сетевое соединение или подключиться к приложению, находящемуся на другом узле сети. На транспортном уровне программист должен выбрать, какой из протоколов он будет использовать. Существует несколько широко распространённых протоколов транспортного уровня. Наиболее часто используется программистами протоколы TCP (*Transmission Control Protocol*) и UDP (*User Datagram Protocol*), немного реже — протокол RTP (*Real-time Transport Protocol*) и другие.

**TCP** протокол используется когда необходима гарантия доставки пакетов к удалённому узлу. При работе через протокол TCP клиент в начале сеанса связи устанавливает соединение с сервером, а по окончании — разрывает соединение. Передача данных возможна только при наличии соединения. При этом принимающая сторона отсылает передающей стороне подтверждение получения данных, в случае искажения передаваемых данных или при их утере, передающая сторона повторяет передачу пакета.

В протоколе **UDP** передача данных осуществляется без гарантии доставки и без поддержания постоянного соединения. Обеспечение целостности передачи данных должно осуществляться протоколами верхних уровней или игнорироваться.

Протокол UDP часто используется там, где важна скорость передачи, а частью данных можно пренебречь.

Если важно сохранение последовательности передаваемых кадров, то имеет смысл воспользоваться протоколом RTP, который обычно применяется при передаче мультимедийных данных (речь, видео). Протокол RTP базируется на протоколе UDP и отличается только тем, что в составе каждого пакета присутствует информация, позволяющая восстанавливать последовательность пакетов, существовавшую при передаче.

**Сеансовый уровень** определяет порядок установки соединения, то есть порядок инициализации сессии, порядок её проведения. **Реализация этого уровня полностью ложится на плечи программиста.**

Именно на этом уровне определяется порядок обмена сообщениями между клиентским сокетом и сокетом на сервере.

На **представительном** уровне определяется форма передаваемого сообщения. Здесь программист выбирает представление своего сообщения, то есть, в каком виде он будет передавать и принимать данные, то ли в виде бинарной последовательности, то ли в виде команд в текстовом режиме (ASCII или UNICODE), а может быть имеет смысл передавать данные в XML, или сериализовать их в Soap? Применить шифрование или не применять? Эти вопросы представительского уровня нужно решать при разработке собственного стека протоколов сетевого обмена.

**Прикладной (Application)** — это самый верхний уровень модели **OSI**. Этот уровень реализует функциональность верхнего уровня, такую как передача почты (SMTP), получение почтовых сообщений от удалённого сервера (POP), передача файлов по сети (FTP и TFTP), просмотр web-страниц (HTTP и HTTPS), передачу голоса через VoIP (SIP) и другие стандартные протоколы. Программист при разработке собственных приложений для решения тривиальных должен строго следовать соответствующему протоколу.

Если в процессе работы возникнет необходимость (а она возникает очень часто) разработать собственный протокол прикладного уровня, то необходимо очень тщательно прорабатывать систему команд серверу своего протокола и систему ответов от сервера.

Итак, это все семь уровней модели OSI. как вы заметили, в модели отделены программная и аппаратная часть структуры сети. Первые два уровня — это уровни, которые работают с аппаратными средствами сети, они зависят от топологии сети, сетевого оборудования. Остальные

верхние пять уровней очень мало зависят от технических особенностей построения сети. Вы можете перейти на другую сетевую технологию, но это не потребует никаких изменений в программных средствах верхних уровней.

Приведу простой пример отправки почтового сообщения через сервер SMTP. Это можно сделать, используя программу telnet — очень полезную утилиту, позволяющую установить TCP соединение (транспортный уровень) с удалённым узлом (сетевой уровень) и обеспечить передачу команд (прикладной уровень) в виде ASCII строки (представительский уровень). Telnet клиент инициализирует соединение (сеансовый уровень) с SMTP сервером при помощи команды: `telnet smtp.mail.ru`:

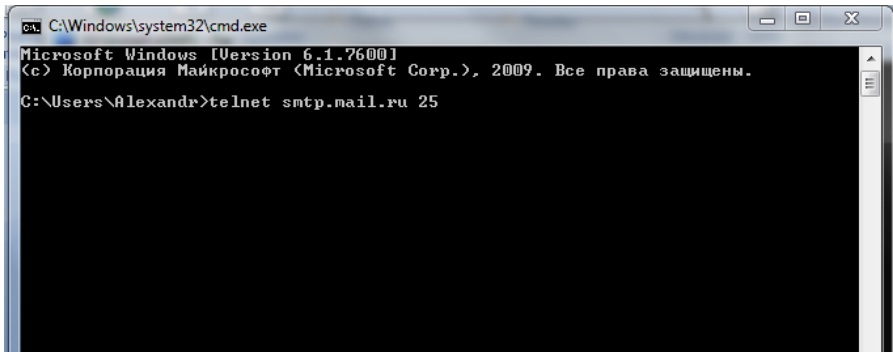


Рисунок 1

где `smtp.mail.ru` — имя узла, а `25` — стандартный TCP порт SMTP сервера.

Кстати обладатели Windows 7 pro должны установить себе Telnet клиент через оснастку «программы и компоненты» -> «включение или отключение компонентов windows» панели управления, Telnet нам очень понадобится на стадии

тестирования сетевых приложений, кроме того, включите простые службы TCP/IP, они нам также понадобятся для тестирования простых клиентских приложений:

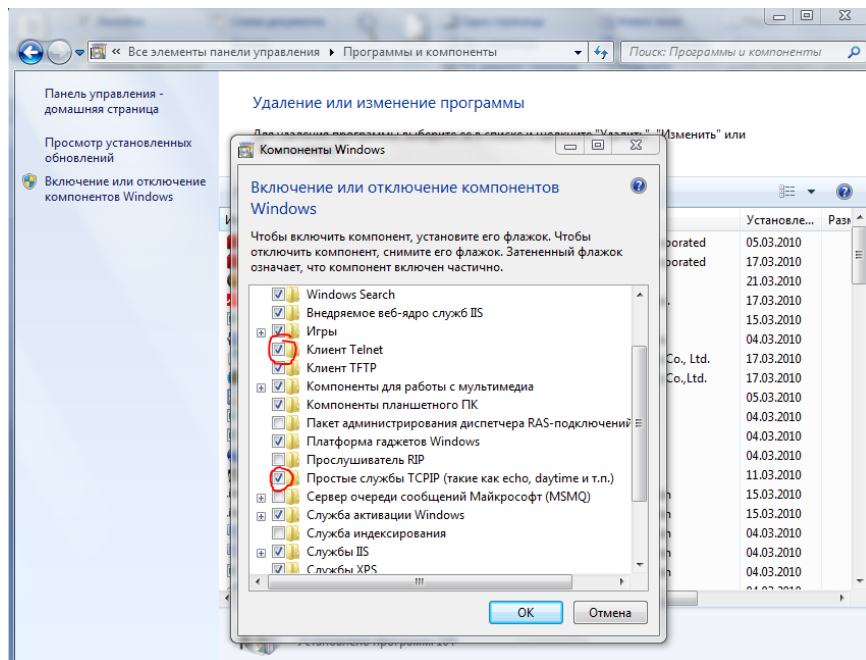


Рисунок 2

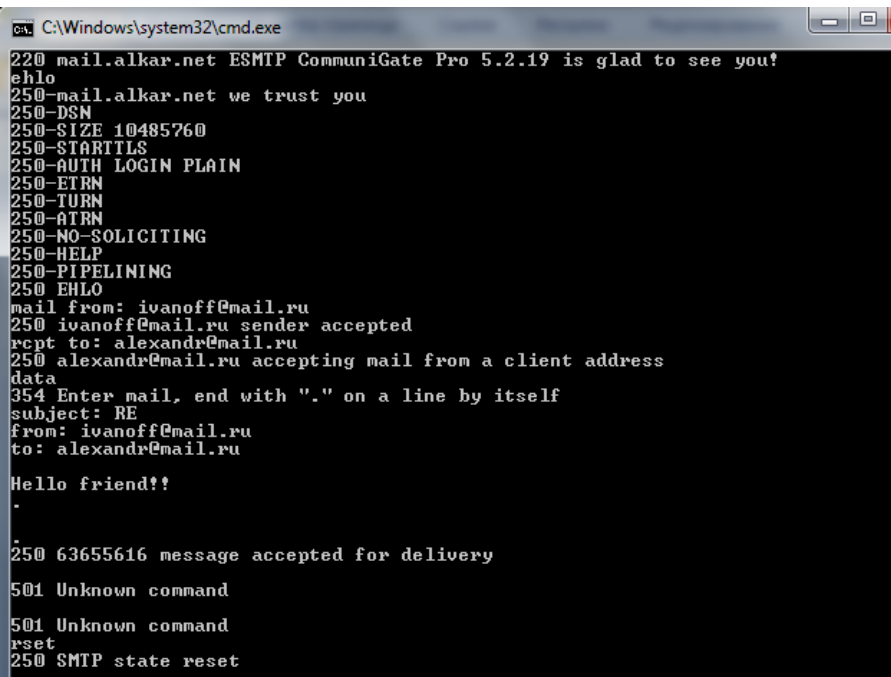
После установки соединения с smtp сервером сервер отправляет клиенту строку, содержащую код 220 (готовность) и информацию о себе. Клиент посылает серверу приветствие «ehlo» (то есть сервер первым передаёт клиенту, а клиент отвечает серверу — порядок общения определяется сеансовым уровнем).

Обмен данными происходит в виде строк в семибитной кодировке ASCII, что определяется представительским уровнем протокола SMTP.

Команды, отдаваемые серверу, и получаемые от сервера ответы определяются прикладным уровнем и описаны в соответствующем документе RFC–821 (RFC–2821 для ESMTP).

Ознакомиться с соответствующими документами можно на сайте <http://ietf.org>.

Ну, в общем, то приблизительно так:



```

C:\Windows\system32\cmd.exe
220 mail.alkar.net ESMTP CommuniGate Pro 5.2.19 is glad to see you!
ehlo
250-mail.alkar.net we trust you
250-DSN
250-SIZE 10485760
250-STARTTLS
250-AUTH LOGIN PLAIN
250-ETRN
250-TURN
250-ATRN
250-NO-SOLICITING
250-HELP
250-PIPELINING
250 EHLO
mail from: ivanoff@mail.ru
250 ivanoff@mail.ru sender accepted
rcpt to: alexandr@mail.ru
250 alexandr@mail.ru accepting mail from a client address
data
354 Enter mail, end with "." on a line by itself
subject: RE
from: ivanoff@mail.ru
to: alexandr@mail.ru
Hello friend!!
.
250 63655616 message accepted for delivery
501 Unknown command
501 Unknown command
rset
250 SMTP state reset
  
```

Рисунок 3

Правда, я в конце не отправил письмо, а то получатель, адрес которого я выдумал, очень бы удивился, получив письмо от неизвестного ему адресата. Кроме того сервер *smtp.mail.ru* ещё требует процедуры аутентификации, которая отсутствует в приведенном выше фрагменте.

И не забывайте, что в заголовке письма будет записан IP адрес отправителя.

Возможность взаимодействия с другими системами по сети в операционных системах семейства Windows NT обеспечивается при помощи сокетов — совместимого и почти полного аналога библиотеки Berkeley Sockets — разработанной в 1983 году для упрощения реализации сетевого взаимодействия в операционных системах семейства UNIX.

Сам термин SOCKET переводиться как гнездо или розетка (этим термином называли гнезда на телефонных станциях с ручной коммутацией).

В операционных системах, начиная с Windows NT, применяется вторая версия библиотеки Windows Sockets. Эта библиотека полностью доступна разработчику на WinAPI, но она содержит неуправляемый код. Естественно, что создатели .Net Framework создали управляемую библиотеку, которая инкапсулирует в себе вызовы неуправляемого кода, предоставляя разработчику удобный инструмент для написания сетевых приложений.

При этом, поскольку этот инструмент полностью совместим с неуправляемой библиотекой WinSock2 и с Berkeley Sockets UNIX систем, то это позволяет строить распределённые системы, в которых узлы могут функционировать на разных платформах.



## 6. Класс Socket

Классы для работы с сетью находятся в пространствах имён [System.Net](#) и [System.Net.Sockets](#);

### **Класс [System.Net.Sockets.Socket](#)**

Итак, класс [Socket](#) — это класс, реализующий на платформе *Microsoft.Net Framework* интерфейс сокетов [Berkeley](#). Как указывается в MSDN, [Socket](#) имеет набор методов и свойств для реализации сетевого взаимодействия. Класс [Socket](#) позволяет выполнять передачу и приём данных с использованием любого из коммуникационных протоколов, имеющих в перечислении [ProtocolType](#), таких как:

**Таблица 1**

Имя члена	Описание
IP	Протокол IP
Icmp	Протокол ICMP
Igmp	Протокол IGMP
Ggp	Протокол GGP
IPv4	Протокол IPv4
Tcp	Протокол TCP
Pup	Протокол PUP
Udp	Протокол UDP
Idp	Протокол IDP
IPv6	Протокол IPv6
Raw	Протокол Raw IP
Ipx	Протокол IPX
Spx	Протокол SPX
Spxll	Протокол SPXII

Полный список поддерживаемых протоколов вы можете увидеть в MSDN.

В зависимости от назначения объекта класса [Socket](#), сокеты делятся на активные и пассивные.

- **Активный сокет** предназначен для установления соединения с удалённым сервером со стороны клиента.
- **Пассивный сокет** — это серверный сокет, ожидающий соединения с ним от клиентов.

В зависимости от назначения сокета (активный или пассивный) отличается и алгоритм работы с ним.

Кроме того, существует понятие синхронного и асинхронного сокета.

Дело в том, что при организации обмена сообщениями с использованием сокетов мы должны предусмотреть механизм позволяющий обслуживать всех подключившихся клиентов. При использовании синхронных сокетов для решения этой проблемы обычно используется механизм обеспечения многозадачного выполнения платформы.

В отличие от синхронных, в асинхронных сокетах присутствует встроенный механизм многопоточной обработки событий, что позволяет производить обмен данными в асинхронном режиме.

### ***Установка соединения со стороны клиента синхронного соединения с использованием протокола TCP***

Для того чтобы установить соединение со стороны клиента необходимо сделать следующее:

1. Создать объект типа [Socket](#), указав ему тип сети (в приведенном ниже примере [AddressFamily.InterNetwork](#)

- (IPv4), тип транспортного протокола `SocketType.Stream` (TCP) и `ProtocolType`);
2. Вызвать метод `Connect`, передав ему в качестве параметра объект класса `EndPoint`, инкапсулирующий в себе IP адрес удалённой машины и порт, к которому необходимо подключиться.
  3. В случае успешного соединения можно начинать обмен сообщениями в соответствии с протоколами прикладного, представительского и сеансового уровней, используя метод `Send` для отправки сообщений и `Receive` для получения.

```
IPAddress ip=IPAddress.Parse("207.46.197.32");
EndPoint ep = new EndPoint(ip, 80);
Socket s = new Socket(AddressFamily.InterNetwork,
    SocketType.Stream, ProtocolType.IP);
try{
    s.Connect(ep);
    if (s.Connected)
    {
        String strSend="GET\r\n\r\n";
        s.Send(System.Text.Encoding.ASCII.
            GetBytes(strSend));
        byte[] buffer = new byte[1024];
        int l;
        do
        {
            l = s.Receive(buffer);
            textBox1.Text +=
                System.Text.Encoding.ASCII.
                    GetString(buffer, 0, l);
        } while (l > 0);
    }
    else
```

```

        MessageBox.Show("Error");
    }
    catch (SocketException ex)

        MessageBox.Show(ex.Message);
}

```

В приведенном выше примере мы подключились к порту 80 (стандартный порт прикладного протокола http) удалённого хоста **207.46.197.32** (*Microsoft.com*) и отправили туда стандартный **GET** запрос (в соответствии со спецификацией HTTP1.0). Сервер вернул нам содержимое запрошенной страницы по умолчанию (протокол HTTP1.0 не поддерживает работу с виртуальными хостами).

Обратите внимание, что нам пришлось перекодировать строки в массивы байтов в кодировке ASCII, поскольку формат передаваемых данных определяется на представительском уровне протокола HTTP.

После работы с сокетом его необходимо корректно закрыть, для этого последовательно вызывается метод **Shutdown** для блокировки передачи данных и **Close** для освобождения управляемых и неуправляемых ресурсов, использующихся сокетом.

Метод **Shutdown** принимает в качестве параметра перечисление **SocketShutdown**, в зависимости от значения которого запрещается дальнейшая отправка или (и) получение сообщений.

Для корректного завершения работа с сокетом добавляем следующий блок кода:

```
finally{
    s.Shutdown(SocketShutdown.Both);
    s.Close();
}
```

### **Принимаем соединение со стороны сервера в синхронном режиме с установлением соединения (TCP): (работа с пассивным сокетом в синхронном режиме)**

Для того чтобы создать сокет, принимающий TCP соединение на стороне сервера и обеспечить обмен данными с подключившимся клиентом необходимо сделать следующее

1. Создать объект типа `Socket`, указав ему тип сети (в приведенном ниже примере `AddressFamily.InterNetwork` (`IPv4`), тип транспортного протокола `SocketType.Stream` (`TCP`) и `ProtocolType`);
2. Связать полученный сокет с IP адресом и портом на сервере, вызвав метод `Bind` сокета. В качестве параметра `Bind` принимает объект класса `EndPoint`, инкапсулирующий в себе IP адрес сервера и порт, к которому будут подключиться клиенты.
3. Установить сокет в состояние прослушивания, вызвав метод `Listen` и передав ему в качестве параметра размер очереди ожидающих обработки подключений.
4. Внутри цикла вызывается метод `Accept`, вызов которого является блокирующим для данного потока выполнения. Метод `Accept` при подключении клиента разблокирует поток и возвращает порождённый им

новый объект `Socket`, через который происходит обмен сообщениями с удалённым клиентом. При этом номер порта у порождённого сокета отличается от номера порта, через который осуществляется прослушивание.

5. После завершения обмена сообщениями через сокет, который был получен из метода `Accept`, этот сокет закрывается, и вновь вызывается `Accept` до подключения нового клиента.

В приведенном ниже примере происходит ожидание подключения к порту `1024` на `IP 127.0.0.1 (localhost)`, после подключения клиента ему сбрасывается строка, содержащая текущее время и дату, а в окно консоли выводится IP адрес клиента и номер порта удалённого клиента:

```
Socket s = new Socket(AddressFamily.InterNetwork,
    SocketType.Stream,
    ProtocolType.IP);
IPAddress ip = IPAddress.Parse("127.0.0.1");
IPEndPoint ep = new IPEndPoint(ip, 1024);

s.Bind(ep);
s.Listen(10);

try
{
    while (true)
    {
        Socket ns = s.Accept();
        Console.WriteLine(ns.RemoteEndPoint.ToString());
        ns.Send(System.Text.Encoding.ASCII.
            GetBytes(DateTime.Now.ToString()));
        ns.Shutdown(SocketShutdown.Both);
        ns.Close();
    }
}
```

```

    }
}
catch (SocketException ex)
{
    Console.WriteLine(ex.Message);
}

```

Проверить работу данной программы можно с помощью утилиты **telnet**, выполнив подключение на **localhost** (127.0.0.1) к порту 1024.

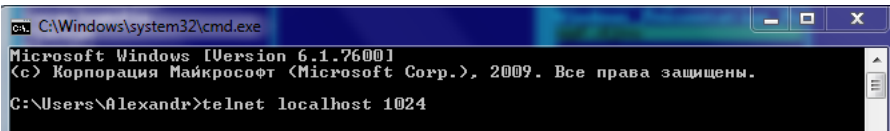


Рисунок 4

Примерный вид работающего сервера и клиента ниже.

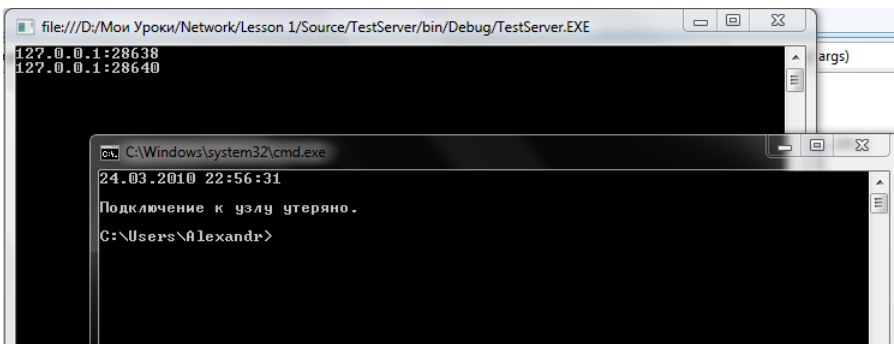


Рисунок 5

Естественно, если вы укажете в методе **Bind** IP адрес вашего сетевого интерфейса, то и подключаться необходимо будет, также используя этот адрес.

## 7. Асинхронные сокеты

Такой простой метод взаимодействия между клиентом и сервером, как применён в примере выше конечно имеет право на существование, но он не лишён ряда недостатков, а именно, если клиенту необходимо достаточно долго поддерживать соединение с сервером, то пока клиент не освободит сокет, никто другой не сможет подключиться к серверу.

Выход из этой ситуации есть и он очевиден, нужно применить многопоточную обработку, то есть взаимодействие с каждым из клиентов производить в отдельном потоке выполнения, то есть применить асинхронную обработку. Конечно, можно реализовать многопоточную обработку вручную, как показано в примере ниже:

```
class Server
{
    delegate void ConnectDelegate (Socket s);
    delegate void StartNetwork(Socket s);
    Socket socket;
    IPEndPoint endP;

    public Server(string strAddr, int port)
    {
        endP = new IPEndPoint(IPAddress.
            Parse(strAddr), port);
    }
    void Server_Connect(Socket s)
    {
        s.Send(System.Text.Encoding.ASCII.
            GetBytes(DateTime.Now.ToString()));
    }
}
```



```

        s.Shutdown(SocketShutdown.Both);
        s.Close();
    }
    void Server_Begin(Socket s)
    {
        while (true)
        {
            try
            {
                while (s!=null)
                {
                    Socket ns = s.Accept();
                    Console.WriteLine(ns.
                        RemoteEndPoint.ToString());
                    ConnectDelegate cd = new
                        ConnectDelegate(
                            Server_Connect);
                    cd.BeginInvoke(ns, null, null);
                }
            }
            catch (SocketException ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
    public void Start()
    {
        if (socket != null)
            return;
        socket = new Socket(AddressFamily.
            InterNetwork, SocketType.Stream,
            ProtocolType.IP);
        socket.Bind(endP);
        socket.Listen(10);
        StartNetwork start =
            new StartNetwork(Server_Begin);
    }

```

```

        start.BeginInvoke(socket, null, null);
    }
    public void Stop()
    {
        if (socket != null)
        {
            try
            {
                socket.Shutdown(SocketShutdown.Both);
                socket.Close();
                socket = null;
            }
            catch (SocketException ex)
            {
            }
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Server s = new Server("127.0.0.1", 1024);
        s.Start();
        Console.Read();
        s.Stop();
    }
}

```

В данном примере при выполнении подключения со стороны клиента создаётся делегат для обработки соединения с клиентом. Причём этот делегат запускается асинхронно, что обеспечивает выполнение взаимодействия с подключившимся клиентом в отдельном потоке. Сам слушающий сокет также запущен в отдельном потоке.

Вам приведенный выше код не показался слишком сложным? Надеюсь, что нет. Но, тем не менее, его можно значительно упростить. Дело в том, что сокеты сами по себе умеют работать асинхронно, без написания дополнительного кода

Для обеспечения возможности асинхронной работы в классе `Socket` предусмотрен набор соответствующих методов:

Таблица 2

Имя	Описание
<code>AcceptAsync</code>	Начинает асинхронную операцию, чтобы принять попытку входящего подключения.
<code>BeginAccept</code>	Перегружен. Начинает асинхронную операцию, чтобы принять попытку входящего подключения.
<code>BeginConnect</code>	Перегружен. Начинает выполнение асинхронного запроса для подключения к удаленному узлу.
<code>BeginDisconnect</code>	Начинает выполнение асинхронного запроса для отключения от удаленной конечной точки.
<code>BeginReceive</code>	Перегружен. Начинает выполнение асинхронного приема данных с подключенного объекта <code>Socket</code> .
<code>BeginReceiveFrom</code>	Начинает выполнение асинхронного приема данных с указанного сетевого устройства.
<code>BeginReceiveMessageFrom</code>	Начинает асинхронный прием заданного числа байтов данных в указанное место буфера данных, используя заданный объект <code>SocketFlags</code> , а также сохраняет конечную точку и информацию пакета.
<code>BeginSend</code>	Перегружен. Выполняет асинхронную передачу данных на подключенный объект <code>Socket</code> .

Имя	Описание
BeginSendFile	Перегружен. Выполняет асинхронную передачу файла на подключенный объект <b>Socket</b> .
BeginSendTo	Выполняет асинхронную передачу данных на указанный удаленный узел.
ConnectAsync	Начинает выполнение асинхронного запроса для подключения к удаленному узлу.
DisconnectAsync	Начинает выполнение асинхронного запроса для отключения от удаленной конечной точки.
EndAccept	Перегружен. Асинхронно принимает попытку входящего подключения.
EndConnect	Завершает ожидающий асинхронный запрос на подключение.
EndDisconnect	Завершает ожидающий асинхронный запрос на разъединение.
EndReceive	Перегружен. Завершает отложенное асинхронное чтение.
EndReceiveFrom	Завершает отложенное асинхронное чтение с определенной конечной точки.
EndReceiveMessageFrom	Завершает отложенное асинхронное чтение с определенной конечной точки. Этот метод также показывает больше информации о пакете, чем метод <b>EndReceiveFrom</b> .
EndSend	Перегружен. Завершает отложенную операцию асинхронной передачи.
EndSendFile	Завершает отложенную операцию асинхронной передачи файла.
EndSendTo	Завершает отложенную операцию асинхронной отправки в определенное местоположение.
ReceiveAsync	Начинает выполнение асинхронного запроса, чтобы получить данные с подключенного объекта <b>Socket</b> .

Имя	Описание
<code>ReceiveFrom</code>	Перегружен. Получает датаграмму и сохраняет конечную точку источника.
<code>ReceiveFromAsync</code>	Начинает выполнение асинхронного приема данных с указанного сетевого устройства.
<code>ReceiveMessageFrom</code>	Получает указанное число байтов данных в указанное место буфера данных, используя заданный объект <code>SocketFlags</code> , а также сохраняет конечную точку и информацию пакета.
<code>ReceiveMessageFromAsync</code>	Начинает асинхронный прием заданного числа байтов данных в указанное место буфера данных, используя заданный объект <code>SocketAsyncEventArgs</code> , <code>SocketFlags</code> , а также сохраняет конечную точку и информацию пакета.
<code>SendAsync</code>	Выполняет асинхронную передачу данных на подключенный объект <code>Socket</code> .
<code>SendPacketsAsync</code>	Выполняет асинхронную передачу набора файла или буферов данных в памяти на подключенный объект <code>Socket</code> .
<code>SendToAsync</code>	Выполняет асинхронную передачу данных на указанный удаленный узел.

**Принимаем соединение со стороны сервера  
в асинхронном режиме с установлением соединения (TCP):  
(работа с пассивным сокетом в асинхронном режиме)**

Для того чтобы создать сокет, принимающий TCP соединение на стороне сервера и обеспечить обмен данными с подключившимся клиентом в асинхронном режиме необходимо сделать следующее

1. Создать объект типа `Socket`, указав ему тип сети (в приведенном ниже примере `AddressFamily.InterNetwork`

- (IPv4), тип транспортного протокола `SocketType.Stream` (TCP) и `ProtocolType.IP`);
2. Связать полученный сокет с IP адресом и портом на сервере, вызвав метод `Bind` сокета. В качестве параметра `Bind` принимает объект класса `EndPoint`, инкапсулирующий в себе IP адрес сервера и порт, к которому будут подключиться клиенты.
  3. Установить сокет в состояние прослушивания, вызвав метод `Listen` и передав ему в качестве параметра размер очереди ожидающих обработки подключений. Первые три пункта полностью соответствуют алгоритму работы в синхронном режиме, а дальше будут отличия.
  4. Необходимо создать делегат типа `AsyncCallback` и вызвать метод `BeginAccept`, передав ему в качестве параметра делегат и наш слушающий сокет.
  5. При подключении клиента будет вызван делегат, которому в свойстве `AsyncState` параметра типа `IAsyncResult` придет наш слушающий сокет.
  6. У полученного сокета вызывается метод `EndAccept`, который возвращает новый объект `Socket`, через который происходит обмен сообщениями с удалённым клиентом.
  7. Снова вызывается `BeginAccept`, цикл работы повторяется.

Аналогично можно организовать и асинхронную отправку сообщения клиенту, используя метод `BeginSend`, передав ему делегат, который будет вызван по завершению операции отправки и сокет для обмена данными с клиентом.

Приведенный ниже пример запускает прослушивание соединения в асинхронном режиме и асинхронную отправку сообщения клиенту:

```
class AsyncServer
{
    IPEndPoint endP;
    Socket socket;

    public AsyncServer(string strAddr, int port)
    {
        endP = new IPEndPoint(IPAddress.
            Parse(strAddr), port);
    }

    void MyAcceptCallbakFunction(IAsyncResult ia)
    {
        //получаем ссылку на слушающий сокет
        Socket socket=(Socket)ia.AsyncState;
        //получаем сокет для обмена данными с клиентом
        Socket ns = socket.EndAccept(ia);

        //выводим в консоль информацию о подключении
        Console.WriteLine(ns.RemoteEndPoint.ToString());

        //отправляем клиенту текущее время асинхронно,
        //по завершении операции отправки будет
        //вызван метод MySendCallbackFunction
        byte[] sendBufer =
            System.Text.Encoding.ASCII.
            GetBytes(DateTime.Now.ToString());

        ns.BeginSend(sendBufer, 0, sendBufer.Length,
            SocketFlags.None,
            new AsyncCallback(MySendCallbackFunction),
            ns);
    }
}
```

```

        //возобновляем асинхронный Асепт
        socket.BeginAccept(new
            AsyncCallback(MyAcceptCallbakFunction),
            socket);
    }
    void MySendCallbackFunction(IAsyncResult ia)
    {
        //по завершению отправки данных на клиента
        //закрываем сокет (если бы нам понадобился
        //дальнейший обмен данными, мы могли бы его
        //здесь организовать)
        Socket ns=(Socket)ia.AsyncState;
        int n = ((Socket)ia.AsyncState).EndSend(ia);
        ns.Shutdown(SocketShutdown.Send);
        ns.Close();
    }

    public void StartServer()
    {
        if (socket != null)
            return;
        socket = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream,
            ProtocolType.IP);

        socket.Bind(endP);
        socket.Listen(10);
        //начинаем асинхронный Асепт, при подключении
        //клиента вызовется обработчик
        //MyAcceptCallbakFunction
        socket.BeginAccept(new AsyncCallback
            (MyAcceptCallbakFunction),
            socket);
    }
}
class Program
{
    static void Main(string[] args)

```



```

{
    AsyncServer server = new AsyncServer("127.0.0.1",
                                          1024);

    server.StartServer();
    Console.Read();
}
}

```

Асинхронное программирование TCP сокета на стороне клиента может производиться с использованием метода **BeginConnect** для создания соединения со стороны клиента в асинхронном режиме. Также на стороне клиента можно использовать асинхронную передачу и приём сообщений точно также как и при программировании на стороне сервера.

### **Использование класса *Socket* для работы по протоколу UDP в синхронном режиме**

Если используется протокол UDP, то есть протокол без установления соединения, то для отправки сообщений можно воспользоваться методами **SendTo**, а для получения дейтаграмм можно применить методы **ReceiveFrom**.

Алгоритм работы с UDP слушающим сокетом следующий:

1. Создать объект типа **Socket**, указав ему тип сети (в приведенном ниже примере **AddressFamily.InterNetwork** (*IPv4*), тип транспортного протокола **SocketType.Dgram** (*UDP*) и **ProtocolType.IP**);

```

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

```

2. Связать полученный сокет с IP адресом и портом на сервере, вызвав метод `Bind` сокета. В качестве параметра `Bind` принимает объект класса `IPEndPoint`, инкапсулирующий в себе IP адрес сервера и порт, к которому будут подключиться клиенты.

```
socket.Bind(new IPEndPoint(IPAddress.  
Parse("10.2.21.129"), 100));
```

3. Вызвать у сокета метод `ReceiveFrom`, при этом выполнение текущего потока будет приостановлено до появления данных во входном буфере. `ReceiveFrom` возвращает количество считанных байт. Если размер буфера для чтения окажется недостаточным, то может возникнуть исключение `SocketException`;

```
int l=rs.ReceiveFrom(buffer, ref ep);  
  
String strClientIP=((IPEndPoint)ep).Address.ToString();  
  
String str = String.Format("\nПолучено от {0}\r\n{1}\r\n",  
    strClientIP,  
    System.Text.Encoding.Unicode.GetString(buffer,  
        0, 1));
```

4. Для передачи данных используется метод `SendTo` (описание этого метода я приведу позже, при разборе клиентского UDP сокета);  
Поскольку вызов `ReceiveFrom` является блокирующим, то работу с ним можно вынести в отдельный поток выполнения, как это сделано в примере ниже.

```

delegate void AddTextDelegate(String text);
System.Threading.Thread thread;
Socket socket;

public Form1()
{
    InitializeComponent();
}
void AddText(String text)
{
    textBox1.Text+=text;
}
void RecivFunction(object obj)
{
    Socket rs=(Socket) obj;
    byte[] buffer = new byte[1024];
    do
    {
        EndPoint ep = new IPEndPoint(0x7F000000, 100);
        int l=rs.ReceiveFrom(buffer, ref ep);
        String strClientIP= ((IPEndPoint)ep).
            Address.ToString();
        String str = String.Format(
            "\nПолучено от {0}\r\n{1}\r\n",
            strClientIP,
            System.Text.Encoding.Unicode.
                GetString(buffer, 0, l));
        textBox1.BeginInvoke(new
            AddTextDelegate(AddText), str);
    } while (true);
}

private void button1_Click(object sender, EventArgs e)
{
    if (socket != null &&thread!=null)
        return;
    socket=new Socket(AddressFamily.InterNetwork,
        SocketType.Dgram, ProtocolType.IP);

```

```

        socket.Bind(new IPEndPoint(
            IPAddress.Parse("10.2.21.129"), 100));
        thread = new System.Threading.Thread(RecvFunction);
        thread.Start(socket);
    }

    private void button2_Click(object sender, EventArgs e)
    {
        if (socket != null)
        {
            thread.Abort();
            thread = null;
            socket.Shutdown(SocketShutdown.Receive);
            socket.Close();
            socket = null;
            textBox1.Text = " ";
        }
    }

```

Отправки сообщений по протоколу UDP не вызывает никаких вопросов:

```

private void button4_Click(object sender, EventArgs e)
{
    Socket socket = new Socket(AddressFamily.
        InterNetwork, SocketType.Dgram,
        ProtocolType.IP);

    socket.SendTo(System.Text.Encoding.Unicode.
        GetBytes(textBox2.Text),
        new IPEndPoint(IPAddress.
            Parse("10.2.21.255"), 100));

    socket.Shutdown(SocketShutdown.Send);
    socket.Close();
}

```

Здесь мы просто создаём сокет и отправляем синхронное сообщения с использованием `SendTo` на целевой IP адрес.

Обратите внимание на то, что в качестве целевого узла я указал адрес широковещания своей подсети, то есть мои сообщения адресованы всем узлам в подсети.

### ***Работа с UDP в асинхронном режиме***

Кроме синхронной отправки сообщений, UDP сокет также поддерживает возможность работы в асинхронном режиме.

Для асинхронного чтения из UDP сокета используется метод `BeginReceiveFrom`, а для асинхронной отправки `BeginSendTo`.

Ниже я приведу пример асинхронного приёма сообщений через UDP сокет:

```
private void button1_Click(object sender, EventArgs e)
{
    if (socket != null)
        return;
    socket=new Socket(AddressFamily.InterNetwork,
        SocketType.Dgram, ProtocolType.IP);
    socket.Bind(new IPEndPoint(IPAddress.
        Parse("10.2.21.129"), 100));

    state.workSocket = socket;
    RcptRes = socket.BeginReceiveFrom(state.buffer,
        0,
        StateObject.BufferSize,
        SocketFlags.None,
        ref ClientEP,
        new AsyncCallback(Receive_Completed), state);
```

```

}
void Receive_Completed(IAsyncResult ia)
{
    try
    {
        StateObject so = (StateObject)ia.AsyncState;
        Socket client = so.workSocket;
        if (socket == null)
            return;
        int readed = client.EndReceiveFrom(RcptRes,
            ref ClientEP);

        String strClientIP = ((IPEndPoint)ClientEP).
            Address.ToString();
        String str = String.Format(
            "\nПолучено от {0}\r\n{1}\r\n",
            strClientIP,
            System.Text.Encoding.Unicode.
                GetString(so.buffer, 0, readed));

        textBox1.BeginInvoke(new
            AddTextDelegate(AddText), str);

        RcptRes = socket.BeginReceiveFrom(state.buffer,
            0,
            StateObject.BufferSize,
            SocketFlags.None,
            ref ClientEP,
            new AsyncCallback(Receive_Completed),
            state);
    }
    catch (SocketException ex)
    {
    }
}

```

Отправка асинхронных сообщений через UDP сокет также не представляет труда:

```

private void button4_Click(object sender, EventArgs e)
{
    Socket socket = new Socket(
        AddressFamily.InterNetwork,
        SocketType.Dgram,
        ProtocolType.IP);

    byte[] buffer=System.Text.Encoding.Unicode.
        GetBytes(textBox2.Text);
    SendRes = socket.BeginSendTo(buffer, 0,
        SocketFlags.None,
        (EndPoint)new IPEndPoint(
            IPAddress.Parse("10.2.21.255"), 100),
        new AsyncCallback(Send_Completed),
        socket);
}

void Send_Completed(IAsyncResult ia)
{
    Socket socket = (Socket)ia.AsyncState;
    socket.EndSend(SendRes);
    socket.Shutdown(SocketShutdown.Send);
    socket.Close();
}

```

И напоследок, мы привыкли обращаться к именам узлов не по IP адресу, а по DNS имени узла. Для разрешения имени узла в IP адрес (и наоборот) можно воспользоваться статическим классом `System.Net.Dns`, обязательно ознакомьтесь с ним в MSDN;

Так метод этого класса `GetHostAddresses` возвращает массив IP адресов, связанных с данным доменным именем.

## 8. Домашнее задание

---

1. Написать сетевые часы с использованием дейтаграмного протокола. На сервере устанавливается служба отправки пакетов, содержащих информацию о текущем времени в локальную сеть. Клиенты отображают текущее время. Опционально можете сделать систему звонков.
2. Напишите систему обмена сообщениями между клиентами. Система должна использовать центральный сервер, к которому подключаются клиенты и оставляют на нём сообщения. Клиент, которому адресованы сообщения, получает их с сервера по запросу. Система должна использовать протокол, ориентированный на соединение (TCP).



